Quelques erreurs

- Quelques erreurs :
  - division par 0,
  - accès à un indice d'un tableau supérieur ou égal à la longueur de celui-ci,
  - appel d'une méthode sur récepteur `null`,
  - tentative d'affecter une valeur à une variable d'un type incompatible : `int[] t = "toto"`.
- Grâce au typage statique de Java, la dernière erreur est détectée dès la compilation... mais, dans cette liste d'exemples, c'est la seule !
- Que faire des autres erreurs ?

Autre scénario embêtant...

Que se passe-t-il d'indésirable quand on exécute le programme suivant ?

```
static int factorielle(int n) {
  if (n==0) return 1;
  else return n * factorielle(n-1);
}

static void main(String[] args) { System.out.println(factorielle(-2)); }
```

Quelles solutions voyez-vous, quels sont leurs propres inconvénients ?

- tester $n < 0$ et retourner... quoi ?
  - un code d'erreur ? quelle valeur ? -1 ? [1]
  - et si par mégarde on appelle `factorielle(-12)`, ne risque-t-on pas d'utiliser la valeur -1 comme si c'était réellement une factorielle correcte et provoquer une autre erreur bien plus tard dans l'exécution ? [2]

→ il existe des façons plus « propres » et plus sûres de traiter ce cas !

1. ou **`null`**, si on devait retourner une référence
2. ou, dans le cas des références, appeler une méthode sur la valeur **`null`** → crash sur `NullPointerException` !

Qu'est-ce qu'une erreur ?

Les « erreurs » se manifestent à plusieurs niveaux :

1. erreur à la compilation (syntaxe, typage, ...) : le compilateur refuse de compiler le programme, qui ne sera jamais exécuté tel quel
2. crash [1] à l'exécution, avec explication (pile d'appel)
3. comportement incorrect : le programme continue à tourner ou bien termine sans signaler d'erreur mais ne fait pas ce qu'il est censé faire.

Ces catégories sont en fait du moins grave au plus grave.

1. Les erreurs à la compilation se corrigent avant de livrer le produit et ne provoqueront jamais aucun dégât.
2. Les crashs sont plus graves (se produisent dans des programmes déjà en production) mais, quand ça arrive, on sait qu'il y a un bug à corriger.
3. Pire cas : le programme peut fonctionner très longtemps en faisant mal son travail sans qu'on ne s'en aperçoive (parfois, conséquences désastreuses, cf. Ariane 5).

1. correspond à une exception non rattrapée

Gérer les erreurs

En quoi cela consiste-t-il ?

- Avoir une « hygiène » qui tend à faire ressortir les erreurs de programmation dès la compilation (notamment via le typage fort).
- Utiliser les options du compilateur (`-Xlint`) et des outils complémentaires d'analyse statique pour détecter plus d'erreurs avant exécution.
- Savoir quand le composant qu'on programme génère ses propres erreurs. Dans ce cas, il faut les signaler de façon propre aux client du composant et/ou fournir des façons de les éviter.
- Savoir réagir quand un composant qu'on utilise remonte une erreur :
  - prendre en compte l'erreur et proposer un comportement (correct) alternatif,
  - ou bien propager l'erreur (si possible en ajoutant de l'information ou en la traduisant en tant qu'erreur du composant courant) pour qu'un client la traite.

Techniques de gestion des erreurs

Techniques proposées dans ce chapitre :

- lancer (et rattraper) des **exceptions** (nous allons voir ce que c'est) ;
- ajouter une méthode auxiliaire pour pré-valider un appel de méthode ;
- utiliser un type de retour « enrichi ».

Exceptions

Description du mécanisme

Le mécanisme des « exceptions » :

- consiste à gérer un comportement « exceptionnel » du programme, sortant du flot de contrôle « normal » ;
- sert quand il n'y a pas de valeur sensée à passer dans un `return` (la méthode est dans l'incapacité de terminer normalement [1])
  $\rightarrow$ on ne veut pas redonner la main à l'appelant comme si rien d'anormal ne s'était passé ;
- concerne des événements « exceptionnels » = « rares » (l'exécution de ce mécanisme est en fait coûteuse).

1. Que répondriez-vous si on vous demandait : « Quel nombre réel vaut dix divisé par zéro ? »

Exceptions

Factorielle corrigée grâce aux exceptions


```java
class FactorielleNegativeException extends Exception {}

public class Test {
    static int fact(int x) throws FactorielleNegativeException {
        if (x < 0) throw new FactorielleNegativeException();
        else if (x == 0) return 1;
        else return x * fact(x - 1);
    }

    public static void main(String args[]) {
        System.out.println("Entrez un entier");
        int x = (new Scanner(System.in)).nextInt();
        try {
            System.out.println("La factorielle est : " + fact(x));
        } catch (FactorielleNegative e) {  // erreur détectée !
            System.out.println("Votre nombre était négatif !");
        }
    }
}
```

Exceptions

Explication rapide

- Signaler une exception (grâce à l'instruction **`throw`**) fait sortir de la méthode sans exécuter de **`return`**.[1]
- L'exception peut ensuite être rattapée ou non.
- Non-rattrapée $\rightarrow$ le programme se quitte en affichant un message d'erreur
- Rattrapée, si
  - exécution sous la portée dynamique du **`try`** d'un groupe **`try ... catch ...`**
  - l'exception a le type donné entre **`( )`** après le **`catch`**

  $\rightarrow$ exécution du bloc d'instruction de ce **`catch`**.
- La méthode contenant **`try ... catch ...`** n'est pas nécessairement celle qui appelle directement la méthode qui fait **`throw`** (traitement non local de l'erreur).

1. **`throw`** = sortie exceptionnelle ; **`return`** = sortie normale

Exceptions

Non-localité du mécanisme

Supposons :

- que `main` appelle la méthode `f1` qui appelle `f2` qui appelle ... qui appelle `fn` ($\rightarrow$ pile d'appel de méthodes avec `main` en bas de la pile, `fn` en haut)
- et que `fn` signale une exception `exn`

alors

- si `fn` rattrape `exn`, on retrouve un fil d'exécution « normal » [1], sinon, on sort de `fn` de façon « exceptionnelle » $\rightarrow$ alors c'est comme si l'exception se produisait dans `fn-1` (**propagation** de l'exception).
- si `fn-1` la rattrape, exécution normale du **`catch`**, sinon propagation à `fn-2` etc.
- si l'exception est propagée jusqu'à `main` et `main` ne la rattrape pas, le programme se quitte en affichant l'exception [2].

1. On exécute le **`catch`**, le **`finally`**, puis les instructions d'après.
2. Dont les informations très utiles qu'elle contient.

Exceptions

Les mots-clés et leur syntaxe complète

- `throw new MonException(...);` : instruction pour signaler une exception.
- `try { -1- } catch ( -2- ){ -3- } ... catch ( ... ){ ... } finally { -4- }` : bloc (instruction) de traitement des exceptions.
  - On essaye d'abord d'exécuter le bloc `{ -1- }` du `try` (bloc protégé),
  - Si une exception se produit, pour chaque `catch`, on regarde si l'exception a le type donné dans les `( -2- )` et on exécute le bloc `{ -3- }` du premier `catch` qui correspond.
  - Enfin on exécute le bloc `{ -4- }` du `finally` (dans tous les cas).

  Le `try` doit toujours être suivi d'au moins une clause `catch` ou `finally`[1].
- `... maMethode(...)throws ExceptionType { ... }` : clause indiquant que la méthode déclarée peut signaler[2] une exception.[3]

1. Sauf construction *try-with-resource*, voir plus loin.
2. signaler ou lever (*raise* : mot-clé utilisé en OCaml) ou lancer/jeter (*throw*)
3. Cette clause est parfois obligatoire, parfois pas, détails plus loin.

Le bloc try/catch

Détails

- Paramètre de `catch` = déclaration de variable d'un sous-type de `Throwable`.
- **Attention :** en cas de plusieurs clauses `catch`, exceptions doivent être traitées dans l'ordre de sous-typage (les sous-types avant les supertypes) !

```
public class Exception1 extends Exception { }
public class Exception2 extends Exception1 { }
try { .. }
catch (Exception2 e2) { ... }
catch (Exception1 e1) { ... }
```

Si on inverse les deux `catch`, erreur de compilation.

```
error: exception Exception2 has already been caught
```

Raison : `Exception2` est un cas particulier de `Exception1`, donc déjà traité dans le premier `catch`. Le deuxième `catch` est alors inutile. [1]

- Variante (« *multi-catch* ») : `catch (Exception1 | Exception2 e){ ... }`

1. Or le compilateur considère que si on écrit du code inutile, c'est involontaire et donc une erreur.

La construction *try-with-resource* [1]

```
try (Scanner sc = new Scanner(System.in)) {
    System.out.println("Nom ?"); String nom = sc.nextLine();
    System.out.println("Prénom ?"); String prenom = sc.nextLine();
    identite = new Personne(nom, prenom);
}
```

- syntaxe : `try (Resource r = ? /*expr */){ /*instr */? }`
- équivalent : `Resource r = ?; try { ? } finally { r.close(); }`
- Assure que la ressource utilisée sera libérée après usage (avec ou sans exception).
- `Resource` doit implémenter :
  `interface AutoCloseable { void close(); }` (c'est le cas de `Scanner`).
- `r` doit être une variable finale ou effectivement finale.
- on peut aussi écrire juste `try (r){ ... }` si `r` est une variable `AutoCloseable` (effectivement) finale déjà déclarée, ce qui autorise à l'initialiser séparément.

1. Construction introduite dans Java 7 ; initialisation de resource séparée introduite dans Java 9.

Exceptions

Mais au fait qu'appelle-t-on une exception ?

Remarque : on a utilisé le nom « exception » pour parler de plusieurs choses différentes...

- l'événement qui se produit quand on quitte une méthode via le mécanisme qu'on vient de décrire : « une exception vient d'être signalée » ;
  (dans l'exemple, c'est ce qui se produit quand on exécute `throw new FactorielleNegative();`)
- l'objet qui est passé du point de programme où le problème a lieu au point du programme où on gère le problème ;
  (dans l'exemple, l'objet instancié en faisant `new FactorielleNegative();`, récupéré dans la variable `e` quand on fait `catch (FactorielleNegative e)`)
- à la classe d'un tel objet.
  (dans l'exemple : la classe `FactorielleNegativeException`)

Parlons maintenant des objets et des classes !

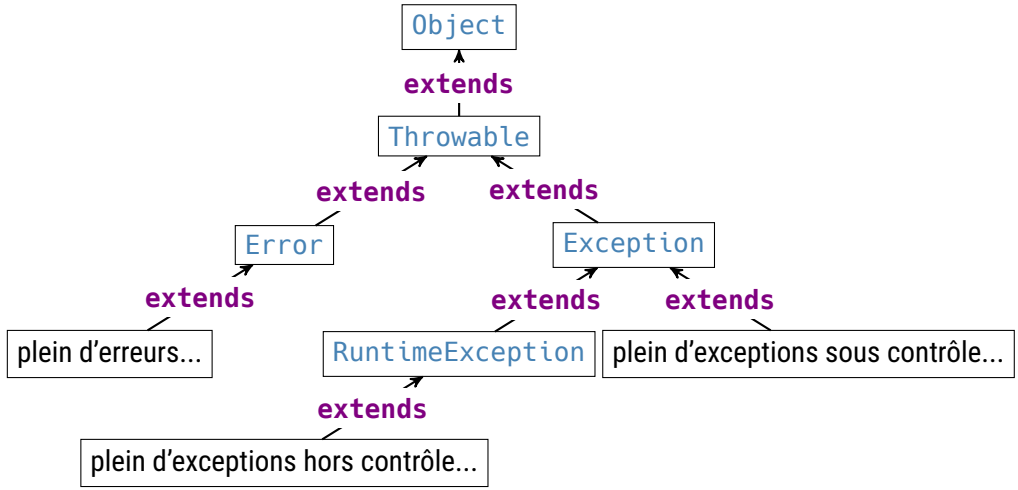
Les objets exception

- Objet exception = objet passé en paramètre de **`throw`**, récupérable par **`catch`**.
- Instance (indirecte) de la classe `Throwable`.
- Contient de l'information utile pour récupérer l'erreur (bloc **`catch`**) ou bien déboguer un crash, notamment :
  - sa classe : le fait d'appartenir à l'une sous-classe d'exception ou une autre est déjà une information très utile.
  - message d'erreur, expliquant les circonstances de l'erreur → `String getMessage()`
  - cause, dans le cas où l'exception est elle-même causée par une autre exception → `Throwable getCause()`
  - trace de la pile d'appels, qui donne la liste des appels imbriqués successifs (numéro de ligne et méthode) qui ont abouti à ce signalement d'exception → `StackTraceElement[] getStackTrace()`

Attention : génération de la trace → coûteuse en temps et en mémoire.
→ Une raison pour réserver le mécanisme des exceptions aux cas exceptionnels.

Classes d'exceptions

Hiérarchie d'héritage



Classes d'exceptions

Description des classes proposées (1)

- Classe `Throwable` :
  - Rôle = marqueur syntaxique pour **`throw`**, **`throws`** et **`catch`** → c'est le type de tout ce qui peut être « lancé » et rattrapé.
  - Après **`throw`** l'expression doit être de (super-)type `Throwable`.
  - Après **`throws`** n'apparaissent que des sous-classes de `Throwable`.
  - Le paramètre déclaré dans un **`catch`** a pour type un sous-type de `Throwable`.
- Classe `Error` :
  - Indique les erreurs tellement graves qu'il n'y a pas de façon utile de les rattraper.
  - On a le droit de les passer en argument d'un **`catch`**... mais ce n'est pas conseillé !
  - Ex. : dépassement de la capacité de la pile d'exécution (`StackOverflowError`),
- Classe `Exception` :
  - Erreurs possiblement récupérables.
  - Elles ont vocation à être passées en argument d'une clause **`catch`**.
  - Exemples :
    - opération bloquante interrompue (`InterruptedException`).

Classes d'exceptions

Description des classes proposées (2)

- Classe `RuntimeException` :
  - Erreurs se produisant au cours d'une exécution normale [1] du « *runtime* » (i.e. : la JVM).
  - Très souvent : erreurs évitables par de simples vérifications à l'exécution (`if`s)
    → faire en sorte qu'elles ne se produisent pas, **ne pas** rattraper dans un `catch` !
    Exemples :
    - division par 0 (`ArithmeticException`),
    - appel de méthode sur `null` (`NullPointerException`),
    - accès à une case qui n'existe pas dans un tableau (`ArrayOutOfBoundException`),
    - tentative de *cast* illégale (`ClassCastException`)
  - Mais, de plus en plus utilisées à la place des `Exception` normales, afin d'éviter les contraintes des exceptions sous-contrôle. (`throws` non requis).
    Dans ce cas : vocation a être rattrapée dans un `catch` malgré tout.

Remarque : l'usage qui est fait de `RuntimeException` est fondamentalement différent des autres sous-classes de `Exception`. Ainsi, « moralement », il est hasardeux de considérer `RuntimeException` comme sous-type de `Exception` (même si c'est vrai).

1. Traduction : si erreur alors que la JVM s'exécute normalement, c'est que le programme a un bug !

Classes d'exceptions

Créer ses propres exceptions

- **Attention :** avant de créer une exception, vérifiez qu'une classe d'exception standard n'est pas déjà définie dans l'API Java pour ce cas d'erreur. [1]
- On crée une classe d'exception personnalisée (le plus souvent) comme sous-classe d'`Exception` ou de `RuntimeException`.
- Le `String` passé au constructeur est le message retourné par `getMessage()`.
- Le `Throwable` passsé au constructeur est la valeur retournée par `getCause()`. Cf. chaînage causal des exceptions.
- **Avertissement :** on ne peut pas déclarer de sous-classe générique de `Throwable`. En effet, `catch` (comme `instanceof`) doit tester le type de l'exception à l'exécution. Or à l'exécution, la valeur du paramètre n'est plus disponible [2]. Donc déclarer une telle classe est inutile. Donc le compilateur l'interdit.

1. Très souvent, c'est en fait `IllegalArgumentException` ou `IllegalStateException` qui conviendrait...
2. voir effacement de type

Chaîner les exceptions

```
    static void f1() { try { f2(); } catch (E2 e) { throw new E1(e); } }
    static void f2() { try { f3(); } catch (E3 e) { throw new E2(e); } }
    static void f3() { throw new E3(); }
    public static void main(String args[]) { f1(); }
```

Quand on exécute, la JVM crashe et affiche l'exception non rattrapée (instance de E1), ainsi que toutes ses causes sucessives.

```
Exception in thread "main" exceptions.E1: exceptions.E2: exceptions.E3
    at exceptions.Exn.f1(Exn.java:25)
    at exceptions.Exn.main(Exn.java:42)
Caused by: exceptions.E2: exceptions.E3
    at exceptions.Exn.f2(Exn.java:33)
    at exceptions.Exn.f1(Exn.java:23)
    ... 1 more
Caused by: exceptions.E3
    at exceptions.Exn.f3(Exn.java:38)
    at exceptions.Exn.f2(Exn.java:31)
    ... 2 more
```

Exceptions sous contrôle vs. hors contrôle

La clause **throws**

- Accolée à la déclaration d'une méthode, la clause **`throws`** signale qu'une exception non rattrapée peut être lancée lors de son exécution.

```
public static void f() throws MonException {
    // Code pouvant générer une exception de type MonException.
}
```

- Cette clause est obligatoire pour les exceptions dites « sous contrôle »[1].
- Conséquence : si `MonException` est sous-contrôle et que `f()` (ci-dessus) est appelée dans une autre méthode `g()` qui ne la rattrape pas, alors `g()` doit elle-même avoir **`throws`** `MonException`, et ainsi de suite.
- Du coup un programme ne peut pas crasher sur une exception sous contrôle, sauf si elle est déclarée dans la signature de `main()`.[2]

1. voir page suivante
2. En fait si... on peut tricher (ce n'est pas évident), mais c'est déconseillé !

Exceptions sous contrôle vs. hors contrôle

Qui est quoi et pourquoi ?

Deux cas :

- Les exceptions sous contrôle (*checked*) doivent toujours être déclarées (`throws`).

  Lesquelles ? toutes les sous-classes de `Exception` sauf celles de `RuntimeException`.

  Raison : les concepteurs de Java ont pensé souhaitable [1] d'inciter fortement à récupérer toute erreur récupérable.
- Les exceptions hors contrôle (*unchecked*) n'ont pas besoin d'un tel signalement.

  Lesquelles ? uniquement les sous-classes de `Error` et de `RuntimeException`.

  Raisons :
  - Les `Error` sont considérées comme fatales : aucun moyen de continuer le programme de façon utile.
  - Les `RuntimeException` peuvent se produire, par exemple, dès qu'on utilise le « . » d'appel de méthode, autant dire tout le temps ! Si elles étaient sous contrôle, presque toutes les méthodes auraient `throws NullPointerException` !

1. Ce point est très controversé. Aucun langage notable, conçu après Java, n'a gardé ce mécanisme.

Exceptions

Remarque sur la non-localité

- Non-localité = point fort des exceptions : en effet l'erreur n'est mentionnée que là où elle se produit et là où elle est traitée. [1]
- Mais la non-localité peut être contradictoire avec l'encapsulation.

→ si un composant B utilise un composant A, B doit « masquer » les exceptions de A. En effet : si C utilise B mais pas A, C ne devrait pas avoir affaire à A. [2]

- Bonne conduite : traiter toutes les erreurs de A dans B. À défaut, traduire les exceptions de A en exceptions de B (en utilisant le chaînage) :

```
class A { public void f() { throw new AException(); } }
class B {
    private A a;
    public void g() {
        try { a.f(); } catch (AException e) { throw new BException(e); }
    }
}
```

1. Pas tout à fait vrai avec les exceptions sous contrôle.
2. Les exceptions sous contrôle rendent le problème particulièrement visible vu qu'elles sont affichées dans la signature des méthodes, mais il existe aussi avec les exceptions hors contrôle.

Pré-valider les appels de méthode

Description de la technique

Vous avez déjà vu cette technique mise à l'œuvre dans l'API Java. Exemples :

- avec les scanners : avant de faire `sc.nextInt()`, il faut faire `sc.hasNextInt()` pour être sûr que le prochain mot lu est interprétable comme entier
- avec les itérateurs : avant l'appel `it.next()`, on vérifie `it.hasNext()`.

En résumé : une méthode dont le bon fonctionnement est soumis à une certaine précondition peut être assortie d'une méthode booléenne retournant la validité de la précondition. En général, les deux méthodes ont des noms en rapport :

- `void doSomething()`/`boolean canDoSomething()`
- `Foo getFoo()`/`hasFoo()`

Le même test doit malgré tout être laissé au début de `doSomething()` :

```
if (!canDoSomething()) throw new IllegalStateException(); // ou IllegalArgumentException
```

(→ si on oublie de tester `canDoSomething` au pire, crash, au lieu de résultat absurde)

Pré-valider les appels de méthode

Exemple

Pour la factorielle, on peut créer une méthode de validation du paramètre, mais le plus simple c'est encore de penser à tester $x \geq 0$ avant de l'appeler.

La méthode factorielle s'écrira elle-même avec ce test et une exception hors contrôle :

```
static int fact(int x) {  // pas de throws
        if (x < 0)
                throw new IllegalArgumentException(); // hors contrôle
        else if (x == 0)
                return 1;
        else
                return x * fact(x - 1);
}
```

Enrichir le type de retour

Il s'agit d'une amélioration de la technique du « code d'erreur » : au lieu de réserver une valeur dans le type, on prend un type somme, « plus large »

- contenant, sans ambiguïté, les vraies valeurs de retour et les codes d'erreur,
- et tel qu'on ne puisse pas utiliser la valeur de retour directement sans passer par un *getter* « fait pour ça ».

Possibilités :

- programmer un « type somme » à la main
- s'il n'y a qu'un seul code d'erreur (ou bien si on ne veut pas distinguer les différentes erreurs), utiliser la classe `Optional<T>` (Java 8).
- si la méthode qui produit l'erreur devait être **`void`**, retourner à la place un **`boolean`** (si on ne souhaite pas distinguer les erreurs) ou mieux, une valeur de type énuméré (chaque constante correspond à un code d'erreur).

Enrichir le type de retour

Exemple 1

Toujours avec la factorielle, avec le type `Optional` :

```
static Optional<Integer> fact(int x) {  // pas de throws
        if (x < 0) return Optional.empty();
        else if (x == 0) return Optional.of(1);
        else return Optional.of(x * fact(x - 1).get());
}
```

Comme cette méthode ne retourne pas un **`int`** (ou `Integer`), on n'est pas tenté d'utiliser la valeur de retour directement.

Pour utiliser la valeur de retour :

```
Optional<Integer> result = fact(x);
if (result.isPresent()) System.out.println(x + "! = " + result.get());
else System.out.println(x + " n'a pas de factorielle !'");
```

Autre possibilité : `result.ifPresent(f -> System.out.println(f));`[1]

1. opérateur « –> » introduit dans Java 8, sert à dénoter une valeur fonctionnelle. *Hors programme !*

Enrichir le type de retour

Exemple 2


```java
public class Mail {

    /* attributs, constructeurs, etc. */

    public static enum SendOutcome { OK, AUTH_ERROR, NO_NETWORK, PROTOCOL_ERROR }

    /*
     send() : envoie le mail. Sans les erreurs, void suffirait...
     ... mais évidemment des erreurs sont possbles.
     Différents codes sont prévus dans l'enum SendOutCome
    */

    public SendOutcome send() {
        /*
            essaye d'envoyer le mail, mais interrompt le traitement
            avec "return error.TRUC;" dès qu'il y a un souci
        */
        return SendOutcome.OK;
    }
}
```

Points forts et limites de chaque approche

Exceptions (1)

- Toutes les exceptions :
  - L'instanciation d'un `Throwable` est coûteux (génération de la pile d'appel, etc.)
  - L'exécution de **`throw`** est coûteuse (proportionnelle à la hauteur de pile d'appel à remonter pour arriver au **`catch`**).
    Cependant : ce coût est inévitable pour pouvoir obtenir un traitement non local [1].
- Exceptions hors contrôle :
  - on peut oublier de les prendre en compte ($\rightarrow$ crash, alors que le programme n'aurait pas dû compiler du tout si l'exception avait été sous contrôle)

$\rightarrow$ idéales, soit pour les erreurs à traitement non local [2], soit pour les erreurs pour lesquelles il est acceptable de laisser crasher le programme.

1. En effet : quel que soit le mécanisme utilisé, un traitement « non local » signifie qu'on doit remonter un nombre arbitraire de niveaux dans la pile d'appels.
2. À condition de ne pas oublier de toutes les traiter ! À cet effet, penser à documenter les méthodes qui lèvent de telles exceptions $\rightarrow$ mot-clé `@throws` de la JavaDoc.

Points forts et limites de chaque approche

Exceptions (2)

- Exceptions sous contrôle :
  - « Pollution syntaxique » : obligation d'ajouter au choix des **`throws`** ou des **`try`** dans toute méthode ou une telle exception peut se produire.
    - Si on sait traiter l'exception localement, pas de problème. [1]
    - Sinon on se retrouve à ajouter une clause **`throws`** à l'appelant (qui vient s'ajouter aux autres s'il y en avait déjà...), puis à l'appelant de l'appelant, puis ... [2]
  - Incitation à faire des **`try catch`** juste pour éviter d'ajouter un **`throws`** : nuisible si on fait taire un problème sans le traiter. P. ex. :

```
void f() throws Ex1 { throw new Ex1(); }
void g() {
  try { f(); } catch (Ex1 e) { /* rien ! <-- ouh, pas bien ! */ }
}
```

→ la pratique moderne semble éviter de plus en plus l'utilisation de ce mécanisme. Cependant l'API Java l'utilise beaucoup et il faut donc savoir comment faire avec.

1. Mais pour un traitement local, à quoi bon utiliser les exceptions ?
2. Modification de signature des appelants qui finalement revient à peu près à changer les types de retour par des types enrichis. → critique courante, dans ce cas, qu'apportent les exceptions sous-contrôle en plus ?

Points forts et limites de chaque approche

Méthodes auxiliaires

- Si vérifier les paramètres est aussi coûteux qu'effectuer l'opération, un appel sécurisé devient deux fois plus long.
  → réserver aux cas où la vérification a priori est peu coûteuse
- Le compilateur ne sait pas vérifier qu'on a appelé et testé `canDoSomething()` avant d'appeler `doSomething()`.
  → problèmes potentiellement reportés à l'exécution.
- Parfois (rarement), on ne sait pas quoi faire quand l'appel à `canDoSomething()` retourne **`false`**. Peut-être que seul l'appelant de l'appelant de l'appelant de... l'appelant connait suffisament de contexte pour traiter le cas d'erreur.
  → (relativement) peu adapté au traitement d'erreur non local [1]

1. On peut propager de la façon suivante : l'opération **`void`** `doFoo()` appelle **`void`** `doBar()`, mais `doBar()` a une méthode de vérification **`boolean`** `canBar()`, alors on peut écrire une méthode de vérification **`boolean`** `canFoo()` pour l'opération `doFoo()`, qui appellera `canBar()`.
Mais ce n'est quand-même pas pratique !

Points forts et limites de chaque approche

Type de retour enrichi

- « Pollution syntaxique » : il est bien plus concis et clair de travailler sur des `int` que des `Optional<Integer>` (déclarations plus courtes, pas de méthodes spéciales pour accéder aux valeurs).
- La « pollution syntaxique » apparait dans les signatures de toutes les méthodes de la pile jusqu'à celle qui traite le cas d'erreur

  $\rightarrow$ à réserver de préférence pour erreurs à traitement local.
- Perte d'efficacité par rapport à valeur directe : on crée un nouvel objet à chaque fois qu'on retourne de la méthode.

  $\rightarrow$ à réserver pour situation où cas d'erreur aussi fréquent que cas « normal » (en comparaison : `throw new E();` coûteux mais ok si rare).

$\rightarrow$ cette technique pose les mêmes problèmes que les exceptions sous contrôle pour les erreurs traitées non localement, mais peut se révéler plus efficace localement.

Points forts et limites de chaque approche

Synthèse des critères

Pas toujours de choix unique et idéal quant à la stratégie de gestion d'une erreur. Les critères suivants, avec leurs exigences contradictoires, peuvent être considérés :

- localité du traitement de l'erreur (local, non local ou pas de traitement) ;
- coût de la vérification de la validité de l'opération avant de l'effectuer ;
- sûreté, c.-à-d. obligation de traiter le cas d'erreur (contre-partie : pollution syntaxique) ;
- fréquence de l'erreur (totalement évitable en corrigeant le programme, rare, fréquente, ...) ;
- qualité de l'encapsulation requise.