ISEN
ALL IS DIGITAL!
OUEST
yncréa

Java

Exceptions

Motivations

Retour sur la classe Fraction

- Imaginons que la méthode setDenominator renvoie un booléen pour indiquer si la modification du dénominateur a bien eu lieu.
- Si l'on souhaite gérer les erreurs, il faudrait tester le retour de setDenominator à chaque appel pour savoir si l'on peut continuer, ce qui donnerait un code comme ici :

```
d1Ok = f1.setDenominator(42);
if(d1OK){
        d2Ok = f2.setDenominator(0);
        if(d2Ok){
                d3Ok = f3.setDenominator(10);
        ...
        }
}
```

Retour sur la classe Fraction

- Il serait bien plus pratique (et lisible!) de pouvoir simplement écrire :

```
f1.setDenominator(42);
f2.setDenominator(0);
f3.setDenominator(10);
...
```

- et que l'exécution du code s'arrête automatiquement si l'une des lignes échoue (ici à la ligne rouge).
  - C'est justement ce que permettent les exceptions !

Principe

Principe

- On modifie le code de la méthode setDenominator() pour qu'elle « lance » (lève) une exception en cas d'erreur

```
public class Fraction {
        public void setDenominator(int denominator){
                if(denominator == 0){
                        throw new IllegalArgumentException("denominator =0");
                }
                this.denominator = denominator;
        }
        ...
}
```

Principe

- Si l'argument denominator vaut 0, alors l'exécution de setDenominator( ) s'arrête brutalement au niveau de la ligne rouge.

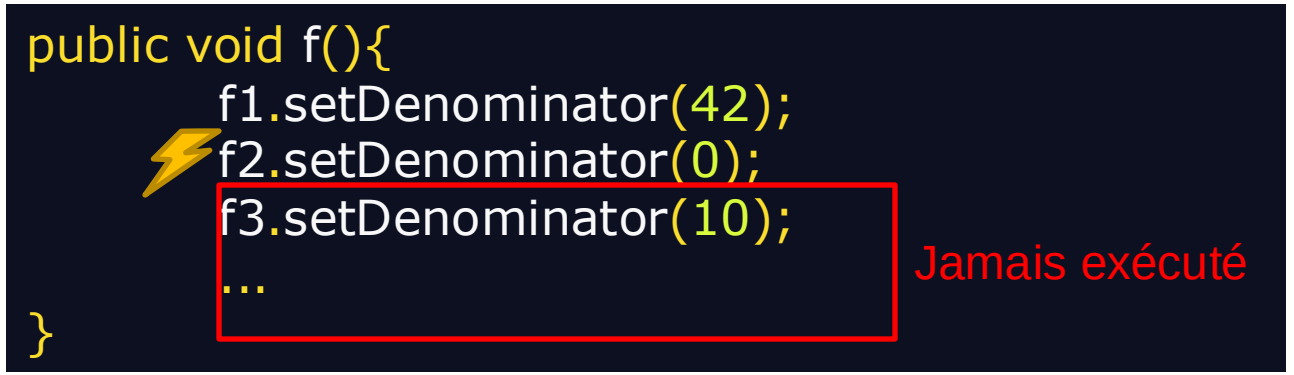
```
public class Fraction {
        public void setDenominator(int denominator){
                if(denominator == 0){
                        throw new IllegalArgumentException("denominator =0");
                }
                this.denominator = denominator;
        }
        ...
}
```

Vaut 0

Jamais exécuté

Principe

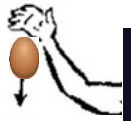
- Le code de la méthode appelante s'arrête également au niveau de l'appel qui a posé problème.



- Mais ce n'est pas tout !

Principe

- L’exception lancée par setDenominator() va remonter la pile



setDenominator()

f()

b()

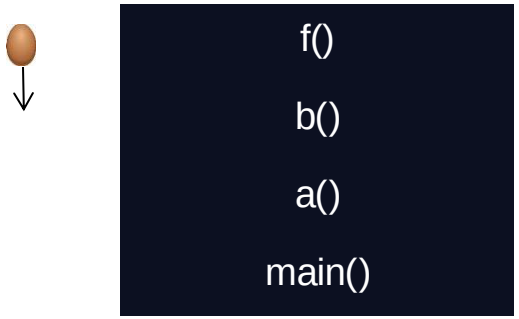
a()

main()

Dans chaque méthode, le code est interrompu et on revient dans le code de la fonction appelante.

Principe

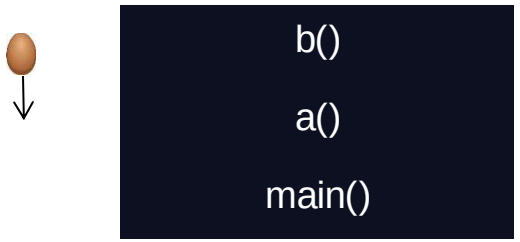
- L’exception lancée par setDenominator() va remonter la pile



Dans chaque méthode, le code est interrompu et on revient dans le code de la fonction appelante.

Principe

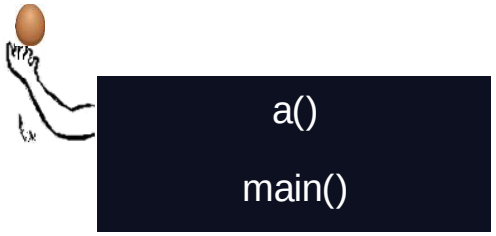
- L’exception lancée par setDenominator() va remonter la pile



Dans chaque méthode, le code est interrompu et on revient dans le code de la fonction appelante.

Principe

- L’exception lancée par `setDenominator()` va remonter la pile



La remontée de pile s'arrête quand l'exception lancée est «attrappée» (*catch*) et traitée de manière à ce que tout rentre dans l'ordre.

Principe

- L'exception lancée par setDenominator() va remonter la pile.

main()

Si l'exception n'est toujours pas traitée dans le main(), le programme s'arrête et le détail sur le cheminement de l'exception (***stack trace***) est affiché.

Affichage de la trace de la pile d'appel :

```
Exception in thread "main" java.lang.IllegalArgumentException: Denominator=0
    at Fraction.setDenominator(Fraction.java:32)
    at Fraction.f(Fraction.java:38)
    at Fraction.b(Fraction.java:42)
    at Fraction.a(Fraction.java:43)
    at Fraction.main(Fraction.java:49)

Process finished with exit code 1
```

Type de l'exception et message d'erreur

pile d'appel

Avantages des exceptions

- On sépare clairement le flux d'exécution normal du programme de la gestion d'erreur. La logique du programme reste claire.
- Les retours des fonctions ne servent qu'à renvoyer les valeurs attendues. Pas besoin d'utiliser des valeurs arbitraires pour modéliser des erreurs (et ce n'est pas toujours possible !)
- Les exceptions remontent automatiquement la pile, nous n'avons pas à propager les erreurs à la main.

Traitement d’une exception

Principe


Levée
d'exception
Flux d'exécution
normal
Gestion de l'exception

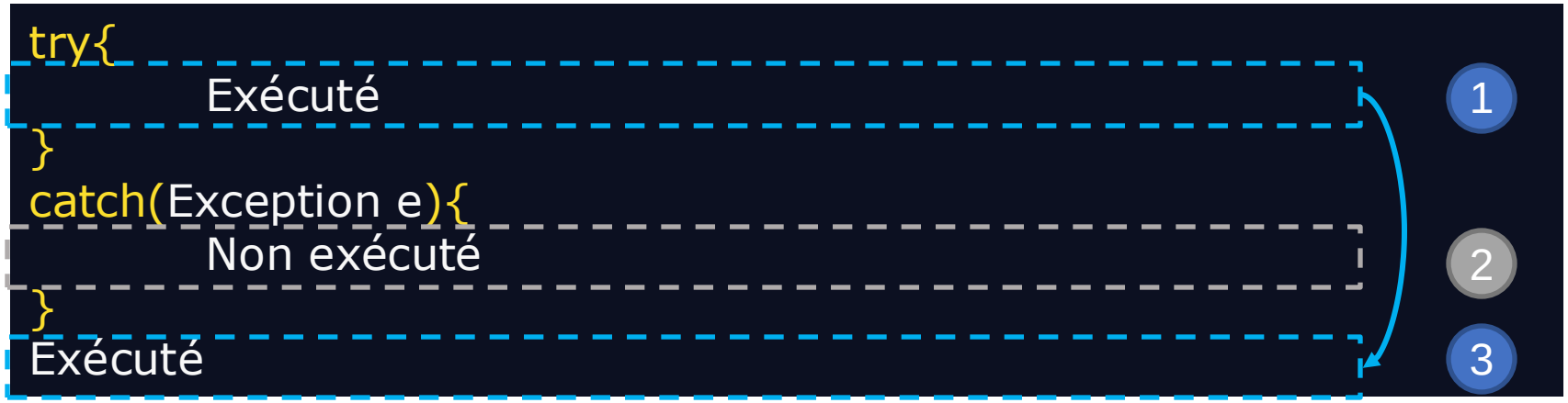
Syntaxe

- Le bloc try/catch

```
try{
        Code susceptible de lancer des exceptions
}
catch(Exception e){
        Code de gestion d'erreur
}
Suite du code
```

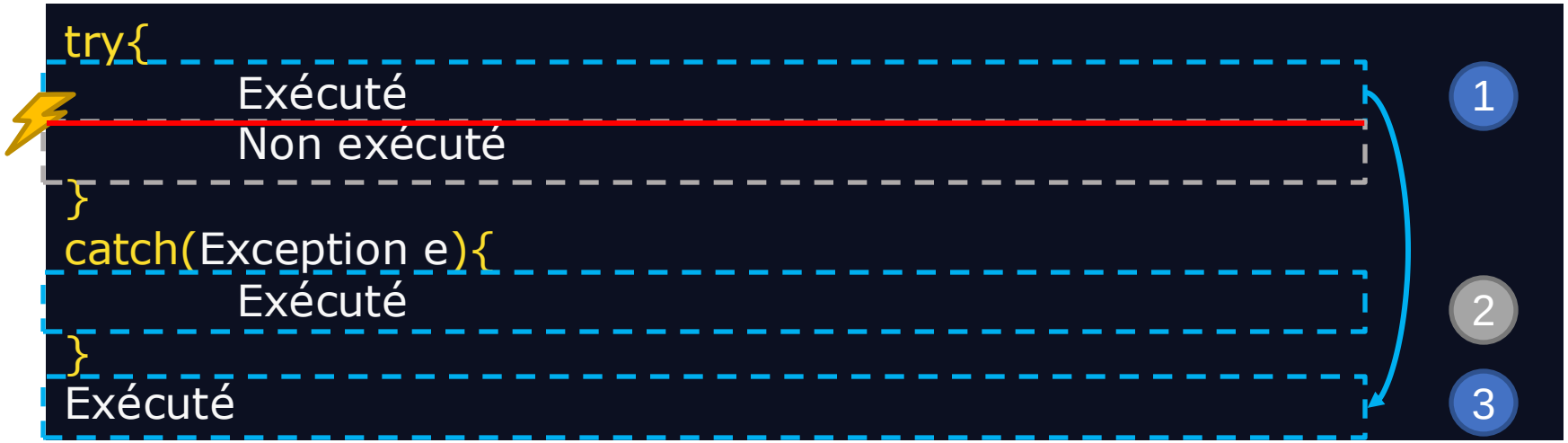
Flux d'exécution

- Exécution si aucune exception n'est levée



Flux d'exécution

- Exécution si aucune exception n'est levée



Exemple

- Code

```
try {
    Fraction f = new Fraction(1,2);
    System.out.println("1");
    f.setDenominator(0);
    System.out.println("2");
}
catch(Exception e){
        System.out.println("3 :" + e.getMessage());
}
System.out.println("4");
```

Sortie standard

```
1
3 : denominator=0
4
```

Sortie d'erreur

```
denominator=0
```

Toutes les exceptions ont un message d'erreur fourni à la construction, que l'on peut afficher à l'aide de la méthode getMessage()

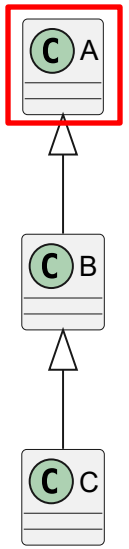
Traitement spécifique des exceptions

Plusieurs gestionnaires

- Il est possible de traiter les exceptions par type, écrivant plusieurs blocs catch
- L'exception sera traitée par le premier bloc catch compatible. Les blocs suivants seront ignorés.

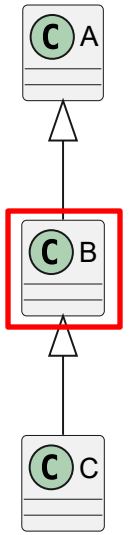
```
try {
        ...
}
catch(A a){
        // capture les classes A et dérivées
}
catch(B b){
        // capture les classes B et dérivées
}
catch(C c){
        // capture les classes C et dérivées
}
```

Exemple



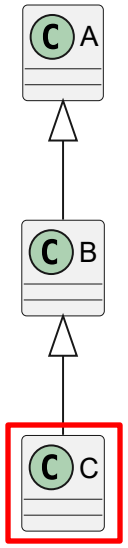

```
try {
        // Exception de type A lancée
}
catch(A a){
        // Exécuté
}
catch(B b){
        // Non exécuté (exception déjà traitée)
}
catch(C c){
        // Non exécuté (exception déjà traitée)
}
```

Exemple



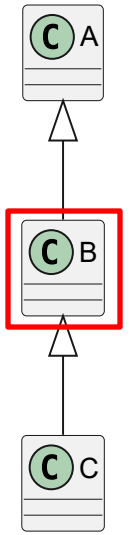

```
try {
        // Exception de type B lancée
}
catch(A a){
        // Exécuté
}
catch(B b){
        // Non exécuté (exception déjà traitée)
}
catch(C c){
        // Non exécuté (exception déjà traitée)
}
```

Exemple




```
try {
        // Exception de type C lancée
}
catch(A a){
        // Exécuté
}
catch(B b){
        // Non exécuté (exception déjà traitée)
}
catch(C c){
        // Non exécuté (exception déjà traitée)
}
```

Exemple




```
try {
        // Exception de type C lancée
}
catch(C c){
        // Non exécuté (incompatible)
}
catch(B b){
        // Exécuté (exception déjà traitée)
}
catch(A a){
        // Non exécuté (exception déjà traitée)
}
```

⚠ Écrire les catch du plus spécifique au plus général

Libération garantie des ressources

Libération des ressources

- Une fois utilisée, une ressource doit être libérée, en particulier un flux de données doit être fermé (fichier, connexion, etc.)
- En Java, ceci est matérialisé par l'appel d'une méthode `close()` (implémentation de l'interface `Closeable`
- Pour garantir que `close()` sera appelée en toutes circonstances, on le place dans un bloc spécial appelé `finally` , dans lequel on entre systématiquement (qu'il y ait eu levée d'exception ou non)

Exemple : Scanner

```
Scanner s = new Scanner(System.in);
try {
	int i = s.nextInt();
	System.out.println(i);
}
catch(InputMismatchException ime){
	System.out.println("Please type an integer");
}
finally{
	s.close();
}
```

Try-with-resources (Java 7+)

```
try(Scanner s = new Scanner(System.in)){
        int i = s.nextInt();
        System.out.println(i);
} // s.close () est automatiquement appelée
catch(InputMismatchException ime){
        System.out.println("Please type an integer");
}
```

- La méthode close() des sources initialisées dans les () du try est systématiquement appelée à la fin du bloc.
- Similaire au mot clé with en Python

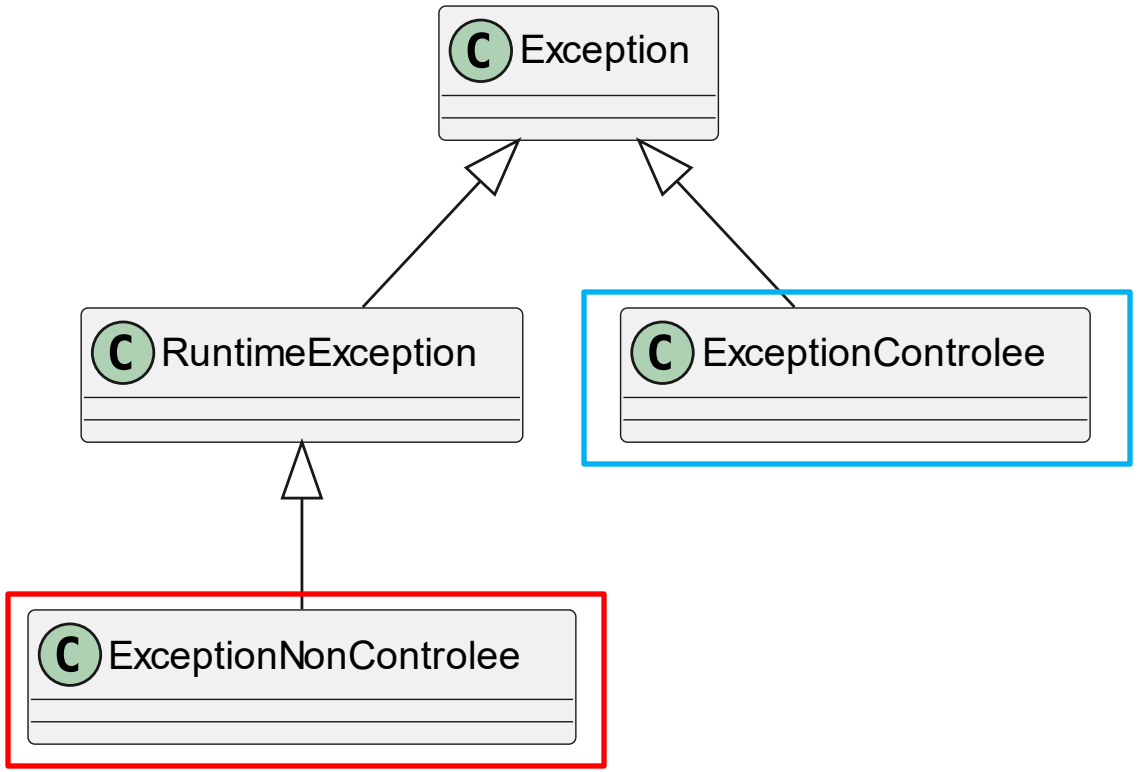
Les types d’exception

Les types d'exception

- Il existe 2 types d'exceptions en Java :
- Les exceptions contrôlées (***checked***)
  - Correspondent typiquement à des erreurs survenant lors d'interactions avec des entités externes
  - Exemples : saisie invalide, problème d'accès au disque, de connexion réseau…
- Les exceptions non contrôlées (***unchecked)***
  - Correspondent à des erreurs de programmation (de logique)
  - Exemples : division par zéro, pointeur nul…

Techniquement

- Les exceptions contrôlées héritent de Exception
- Les exceptions non contrôlées héritent de RuntimeException
- Les exceptions contrôlées font l'objet de vérifications à la compilation : elles doivent être ***soit traitées, soit déclarées***



Techniquement

- Quand une fonction f peut lancer une exception contrôlée A, cette fonction doit impérativement respecter les contraintes suivantes :
  - Elle doit explicitement le déclarer dans sa définition :

```
public void f() throws A
```

  - Toute fonction g() faisant appel à appel à f() doit le faire à l'intérieur d'un try/catch. Ou alors elle doit à son tour déclarer une potentielle levée de A
- On peut faire de même pour les exceptions non contrôlées, mais ceci est facultatif.

Gestion de l'exception lancée par `f` dans `g`

```
public static void f() throws Exception {
    throw new Exception("Error.");
}

public static void g(){
    try{
        f();
    } catch (Exception exception){
        // gestion de l'exception
    }
}
```

`g` ne gère pas l'exception mais la renvoie à son tour.

```
public static void f() throws Exception {
    throw new Exception("Error.");
}

public static void g() throws Exception {
        f();
        // faire d'autres choses
}
```

Créer ses propres exceptions

Créer ses propres exceptions

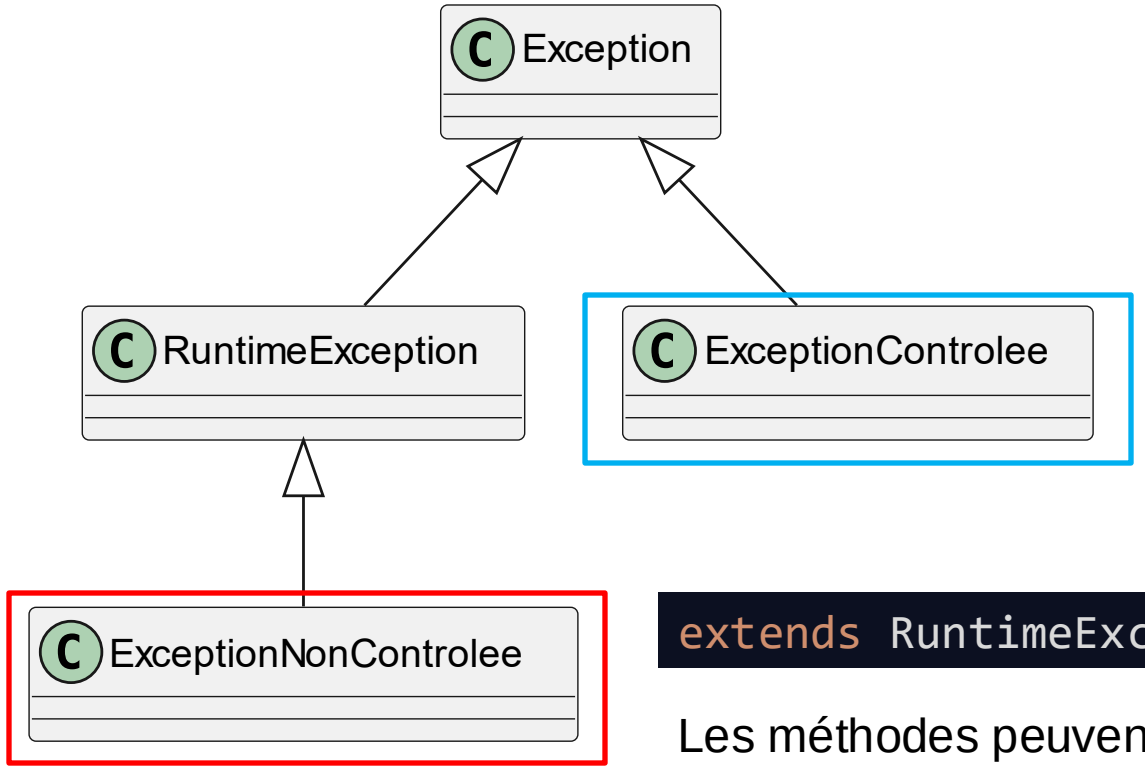
- Créer des exceptions personnalisées permet de modéliser des erreurs propres à une application.
  - Par exemple : mauvaise clé d'API, solde d'un compte insuffisant, matrice non diagonalisable, …
- Pour cela, il suffit d'hériter d'une exception existante et d'ajouter des attributs spécifiques. On peut construire le message d'erreur à partir de la valeur des attributs.

Exemple : solde insuffisant

```
public class AccountBalanceException extends RuntimeException{
    private float debit;
    private float balance;
    public AccountBalanceException(float debit, float balance) {
        super("Requested debit " + debit + " > account balance " + balance);
        this.balance = balance;
        this.debit = debit;
    }

    public float getDebit(){
        return debit;
    }

    public float getBalance(){
        return balance;
    }
}
```




```
extends Exception {
```

Les méthodes devront obligatoirement gérer ces exceptions.

```
extends RuntimeException {
```

Les méthodes peuvent ne pas gérer les exceptions.

Exception contrôlée ou non ?

- La recommandation d'Oracle est la suivante :

If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.

- Toutefois, le choix entre ces deux types d'exception fait débat : la notion de « récupérable » est assez subjective et d'autres éléments entrent en ligne de compte (comme la lisibilité/simplicité du code, l'évolution d'une API, la gestion effective des erreurs…)
  - Cela relève d'une préférence personnelle ou de pratiques de programmation dans une équipe.