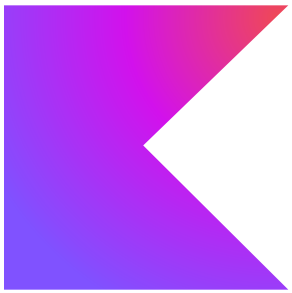


Kotlin

IUT Clermont Auvergne

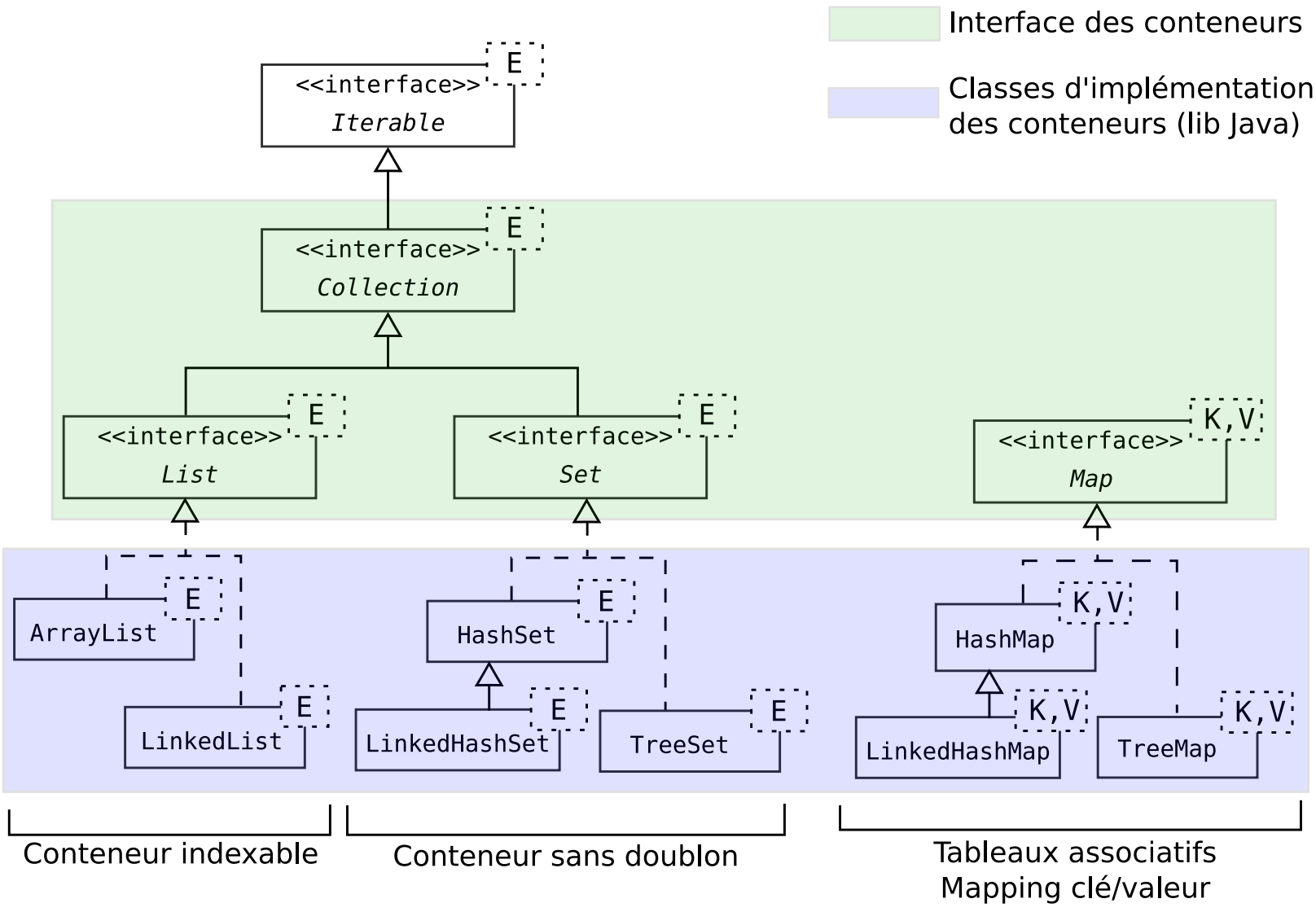
Janvier 2024

Kotlin Avancé

Collections

- Appartiennent à la bibliothèque standard de Kotlin
- package `kotlin.collections`
- Se repose sur les collections Java que vous connaissez
- Séparation en deux modes (interfaces) :
  1. Lecture seule (noms classiques)
  2. Modifiable (noms préfixés par `Mutable`)

Architecture des collections



Collections : création

- Instanciation de collection en lecture : `listOf()`, `setOf()`, `mapOf()`

```
val myList = listOf(1, 2, 3, 4, 5)
println(myList[2])    // Affiche 3
myList[2] = 42        // KO, lecture seule
```

- `emptyList()`, `emptySet()`, `emptyMap()` pour collection vide
- Le type exact est inféré en fonction du type du contenu
- On peut spécifier explicitement

```
val myStrings1 = listOf()  // KO, liste de quoi ???
val myStrings2: List<String> = listOf()
val myStrings3 = listOf<String>()
```

Collections : création

- Instanciation de collection en lecture/écriture : `mutableListOf()`, `mutableSetOf()`, `mutableMapOf()`

```
val myMap = mutableMapOf(1 to "one", 2 to "two", 3 to "three")
println(myMap[2])       // Affiche three
myMap[42] = "forty-two" // OK, modification possible
```

- Attention `val` ou `var` qualifie la variable, pas le contenu de la collection !

```
val myList = mutableListOf(1, 2, 3)
myList.add(4)                       // OK liste modifiable
myList = mutableListOf(5, 6, 7)  // KO, myList estampillée val
```

Collections

- Kotlin utilise les implémentations suivantes :
  - `listOf()` : `ArrayList`
  - `setOf()` : `LinkedHashSet`
  - `mapOf()` : `LinkedHashMap`
- `LinkedHashXXX` conserve l'ordre d'insertion (compromis entre `HashXXX` et `TreeXXX`)
- Toujours possibilité de choisir explicitement

```
val phoneNum = TreeMap<String, String>()
phoneNum["Bob"] = "0621724328"
phoneNum["Alice"] = "0686356472"
println(phoneNum) // Affiche {Alice=0686356472, Bob=0621724328}
```

Collections : utilisation

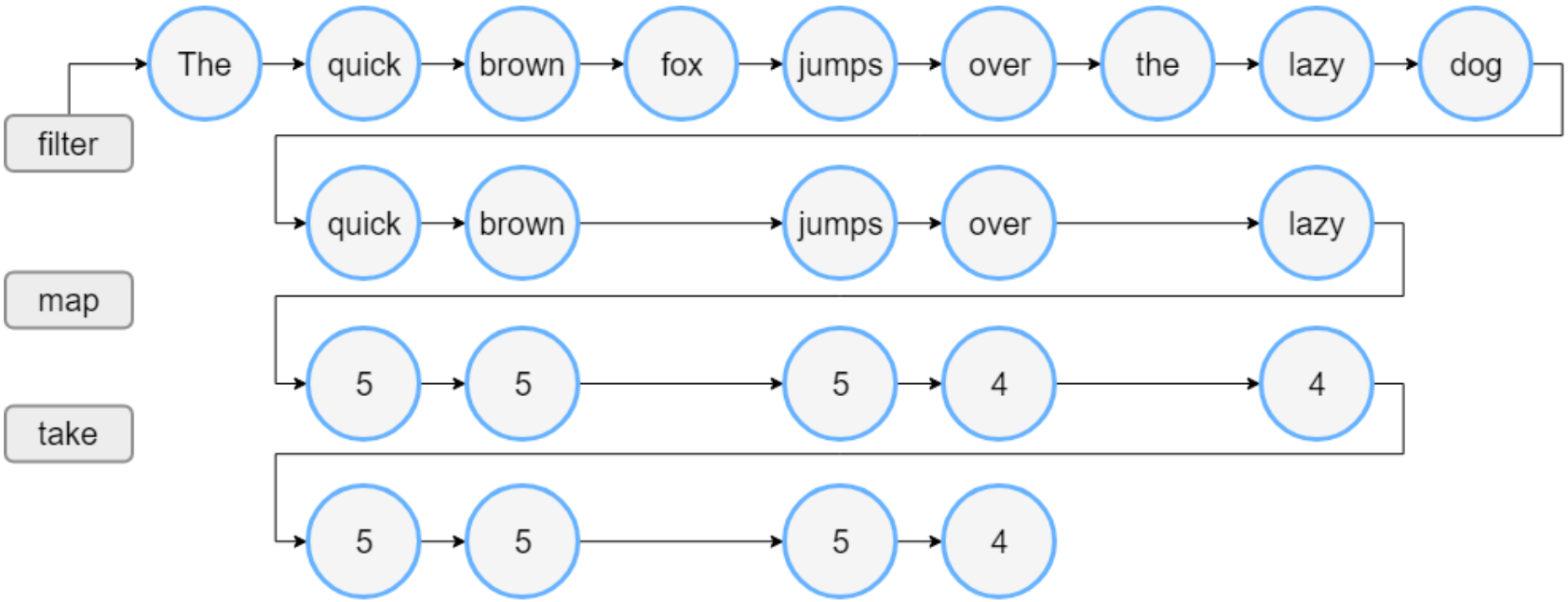
- Nombreuses opérations disponibles sur les collections
- Collections + fonctions d'ordre supérieur + lambda = expressivité
- Transformation (`map`, `associate`, ...)
- Filtrage (`filter`, `all`, ...)
- Regroupement (`groupeBy`, `partition`)
- Découpage (`slice`, `chunked`)
- Ordonnancement (`sorted`, `reversed`)
- Calcul (`sum`, `maxOrNull`, ...)

Collections : quelques exemples

```
// Transformation d'un range en liste
val digits = (0..9).toList()

// Application d'une lambda sur tous les éléments d'une liste
val squares = digits.map { it * it }

// Filtrage suivant un critère
val evenSquares = squares.filter { it % 2 == 0 }

// Décompte d'un nombre d'éléments
val evenSquareWithSix = evenSquares.count {
                            it.toString().contains('6') }

// Parcours parallèle de listes pour transformation en Map
val digitsNames = listOf("zero", "one", "two", "three", "four",
                         "five", "six", "seven", "eight", "nine")
val mapDigits = (digitsNames zip digits).toMap()
```

Fonctionnement (source kotlinlang.org)

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
val lengthsList = words.filter { it.length > 3 }
    .map { it.length }
    .take(4)
println("Lengths of first 4 words longer than 3 chars:")
println(lengthsList)
```



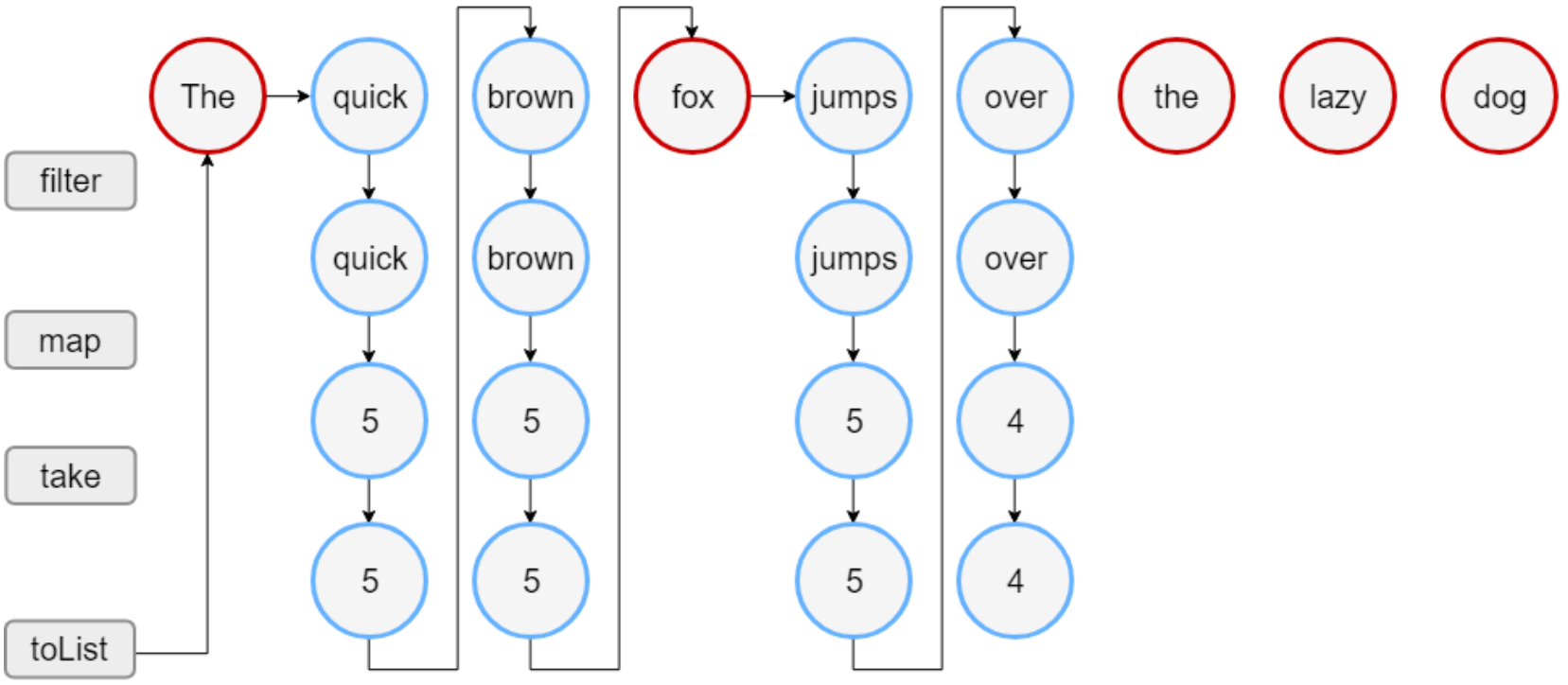
Séquences

- Création de collections temporaires pour les résultats intermédiaires
- Calculs exécutés au moment de l'appel des méthodes
- En anglais *eagerly executed*

- Il existe une autre possibilité : l'exécution paresseuse
- En anglais *lazily executed*
- C'est le domaine des `Sequences`
- Une séquence représente un pipeline de calculs
- Aucune opération n'est effectuée jusqu'à l'appel d'une opération terminale

Séquences : fonctionnement

```
val words = "The quick brown fox jumps over the lazy dog".split(" ")
val wordsSequence = words.asSequence() // Conversion en Sequence
val lengthsSequence = wordsSequence.filter { it.length > 3 }
    .map { it.length }
    .take(4)
println("Lengths of first 4 words longer than 3 chars")
println(lengthsSequence.toList())  // Opération terminale
```



Séquences

- Création de séquence : `sequenceOf()`

```
val numbersSequence = sequenceOf(1, 2, 3, 4, 5)
```

- Depuis un `Iterable` : `asSequence()`
- En utilisant une fonction génératrice : `generateSequence()`

```
val evenNumbers = generateSequence(0) { it + 2 }
println(evenNumbers.take(6).toList())
```

- S'arrête quand renvoie `null` (possibilité de séquences « infinie » sans soucis)

Séquences

- Possibilité de faire des opérations et renvoyer des éléments au fur et à mesure avec `yield()` ou `yieldAll()`

```
val fibonacci = sequence {
    var previous = 0
    var current = 1
    while (true) {
        yield(previous)
        val next = current + previous
        previous = current
        current = next
    }
}
println(fibonacci.take(10).toList())


// La même en plus concis avec la fonction de génération
val fibonacciSexy = generateSequence(Pair(0, 1)) {
    Pair(it.second, it.first + it.second)
}.map { it.first }
```

Généricité

- Actions sur les collections vérifiées à la compilation
- Possible grâce à la généricité
- Abstraction de type utile pour le compilateur
- Disparaît à l'exécution (*type erasure*, tout devient `Object` pour la JVM)

```
// Possible en Java… mais à éviter
ArrayList rawList = ArrayList();
rawList.add(42);
rawList.add("John Doe");

// Doit être contraint en Kotlin
val myList = ArrayList<Int>()
rawList.add(42)
rawList.add("John Doe")   // Pas possible, ne compile pas
```

- Évite au développeur les erreurs d'inattention

Généricité

- Vous pouvez vous-même créer vos types/fonctions génériques
- Syntaxe pour les classes

```
fun interface Observer<T> {
    fun update(data: T)
}

class Observable<T>(iValue: T) {
    var value = iValue
        set(value) {
            field = value
            notifyAll(value)
        }
    private val observers = mutableListOf<Observer<T>>()
    fun addObserver(obs: Observer<T>) = observers.add(obs)
    fun notifyAll(data: T) {
        for (obs in observers)
            obs.update(data)
    }
}
```

Généricité

- Syntaxe pour les fonctions

```
fun <S,T> makePair(first: S, second: T) = Pair(first, second)
```

- Possibilité de mettre des contraintes sur le type générique

```
fun <S : Number> squareDouble(v: S) =
    v.toDouble() * v.toDouble()
```

- Possibilité de faire des fonctions d'extension sur type générique

```
fun <T> T.printAndGet() : T {
    println("Getting value : $this")
    return this
}
val karembeu = 6 * 7.printAndGet()
```

Fonctions d'ordre supérieur

- Expressivité sur les collections obtenue grâce aux fonctions d'extension
- En plus celles-ci sont d'ordre supérieur : fonction qui prend une fonction en paramètre

Exemple `map`

- Prototype :

```
fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R>
```

- Implémentation possible

```
fun <T, R> Iterable<T>.map(transform: (T) -> R): List<R> {
    val result = mutableListOf()
    for (item in this)
        result.add(transform(item))
    return result
```

Fonctions d'ordre supérieur

- Écrivez les vôtres pour rendre plus *Kotlin spirit* votre code
- Coder les fonctions qui font :
  1. la somme de tous les éléments d'une collection
     [1, 2, 3, 4, 5] -> 15
  2. la création d'un message en fonction des éléments d'une collection et d'un préfixe
     "Mes amis sont" ["Alice", "Bob"] -> "Mes amis sont Alice Bob"
  3. le produit de tous les éléments d'une collection
     [1, 2, 3, 4, 5] -> 120

Fonctions d'ordre supérieur

- Le squelette est décrit dans une fonction d'ordre supérieur
- On paramètre en fonction des éléments modifiables

```
fun <T, R> Collection<T>.fold(initial: R,
  combine: (acc: R, nextElement: T) -> R) : R {
    var accumulator = initial
    for (element in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}

val numbers = listOf(1, 2, 3, 4, 5)
numbers.fold(0, {acc, i -> acc + i})    // 15
numbers.fold(1, Int::times)             // 120
val names = listOf("Alice", "Bob", "Carole", "David")
names.fold("Mes amis sont ") {res, i -> "$res $i"}
```

Scope functions

- Fonction (généralement d'extension) qui « ajoute » une portée ponctuelle pour faire des actions
- Permet d'appliquer des actions dans le contexte d'un objet
- Très pratique pour manipuler des nullables en conservant le *Kotlin spirit*
- Elles sont au nombre de 5 :

1. `let`
2. `run`
3. `also`
4. `apply`
5. `with`

let

- Prototype : `fun <T, R> T.let(block: (T) -> R): R`
- Permet d'exécuter le `block` de code où :
  - `this` reste inchangé
  - l'objet contextuel est manipulé avec `it`
  - la valeur retournée est la dernière instruction de `block`
- Usage : exécuter des actions sur un objet non null

```
val res: String? = nullableStringResult()
res?.let {
    println(it.uppercase())
}
// Remplace un bon vieux
if (res != null) {
    println(res.uppercase())
}
```

with

- Ça n'est pas une fonction d'extension
- Prototype :

```
fun <T, R> with(receiver: T, block: T.() -> R): R
```

- Permet d'exécuter le `block` de code où :
  - `this` est l'objet passé en paramètre
  - la valeur retournée est la dernière instruction de `block`
- Usage : appeler des méthodes sur un objet sans avoir à réécrire son nom

```
val window = Window()          // équivalent à
with(window) {                 //
    setTitle("Démo")           // window.setTitle("Démo")
    pack()                     // window.pack()
    setResizable(false)        // window.setResizable(false)
}
```

`run` (pas la plus utile)

1. Fonction d'extension

- Prototype : `fun <T, R> T.run(block: T.() -> R): R`
- Permet d'exécuter le `block` de code où :
  - ▷ l'objet contextuel est manipulé avec `this`
  - ▷ la valeur retournée est la dernière instruction de `block`
- Mix entre `with` et `let` : appelable comme `let`, actions comme `with`

2. Pas une fonction d'extension

- Prototype : `fun <R> run(block: () -> R): R`
- Permet d'exécuter le `block` de code où :
  - ▷ la valeur retournée est la dernière instruction de `block`
- Ensemble d'instructions utilisable comme une expression

apply

- Prototype :
  `fun <T> T.apply(block: T.() -> Unit): T`
- Permet d'exécuter le `block` de code où :
  - l'objet contextuel est manipulé avec `this`
  - la valeur retournée est l'objet contextuel
- Fournit une sorte de pattern builder low-cost
- Pratique pour opérer sur les propriétés/méthodes d'un objet

```
val person = Person().apply {
    name = "Mario"
    age  = 58
}
```

also

- Prototype : `fun <T> T.also(block: (T) -> Unit): T`
- Permet d'exécuter le `block` de code où :
  - ▷ l'objet contextuel est manipulé avec `it`
  - ▷ la valeur retournée est l'objet contextuel
- Fournit une sorte de pattern builder low-cost
- Pratique pour opérer sur un objet directement dans perdre le `this` original

```
val person = Person("John", 48).also {
    println("Log: création de $it")
}
```

Récapitulatif

- Les *scope functions* ont des cas d'utilisation qui se chevauchent
- Choisir en fonction de l'objet à manipuler, comment, et en fonction de la valeur de retour

Fonction	Référence de l'objet contextuel (ROC)	Valeur de retour
`let`	`it`	lambda
`with`	`this`	lambda
`run`	`it`	lambda
`apply`	`this`	ROC
`also`	`it`	ROC

takeIf et takeUnless

- Permettent de faire des vérifications sur un objet en fonction d'un prédicat
- `fun <T> T.takeIf(predicate:(T) -> Boolean):T?`
- `fun <T> T.takeUnless(predicate:(T) -> Boolean):T?`
- `takeIf` (resp. `takeUnless`) retourne `this` si l'objet satisfait (resp. ne satisfait pas) le prédicat, `null` sinon
- Utile en complément des *scope functions*

```
val name = "Mario"
name.first().takeIf { it.isUpperCase() }?.let {
    println("$name commence par la majuscule $it")
}
```

Gestion de ressource

- Bonne gestion des ressources en Java avec le *try-with-resource*
- Libère (`close()`) automatiquement la ressource en fin de portée

```
try (FileReader fr = new FileReader(path);
     BufferedReader br = new BufferedReader(fr)) {
    br.readLine();
    …
}
```

- Le pendant en Kotlin : `use`

```
fun <T : AutoCloseable?, R> T.use(block: (T) -> R): R
```

```
BufferedReader(FileReader(path)).use {
    it.readLine()
    …
}
```