



Formation Kotlin

ALI KOUDRI - AKFC

Agenda

- ▶ Introduction
- ▶ Bases
- ▶ Fonctions
- ▶ Classes
- ▶ Exceptions
- ▶ Structures de données
- ▶ API
- ▶ Concurrency

Introduction

Présentation générale

- ▶ Langage de programmation logiciel supportant différents paradigmes
 - ▶ **Programmation orientée objet** (POO) : Kotlin prend entièrement en charge les concepts de la POO, notamment les classes, les objets, l'héritage et l'encapsulation.
 - ▶ **Programmation fonctionnelle** : Kotlin prend fortement en charge la programmation fonctionnelle, notamment les fonctions d'ordre supérieur, les lambdas et l'immutabilité.
 - ▶ **Programmation impérative** : Kotlin permet la programmation impérative, où vous pouvez écrire du code qui décrit une séquence d'étapes à exécuter.

Présentation générale

- ▶ Langage de programmation logiciel supportant différents paradigmes
 - ▶ **Programmation réactive** : Kotlin prend en charge les paradigmes de programmation réactive, particulièrement utiles pour construire des applications réactives et évolutives.
 - ▶ **Programmation déclarative** : La syntaxe et les fonctionnalités de Kotlin permettent des styles de programmation déclaratifs, particulièrement utiles pour le développement d'interfaces utilisateur et la création de DSL.
 - ▶ **Programmation générique** : Kotlin supporte les génériques, permettant la création de code flexible et réutilisable.
 - ▶ **Programmation concurrente** : Grâce à ses coroutines, Kotlin offre un excellent support pour la programmation concurrente et parallèle.

Présentation générale

- ▶ **Processus de compilation** : Lorsque vous écrivez du code Kotlin ciblant la JVM, le compilateur Kotlin (kotlinc) compile vos fichiers sources Kotlin (.kt) en bytecode Java. Ce bytecode est stocké dans des fichiers .class, tout comme le code Java compilé. Le bytecode résultant est entièrement compatible avec la machine virtuelle Java.
- ▶ **Compatibilité du bytecode** : Le bytecode généré par le compilateur Kotlin est essentiellement le même que le bytecode généré par le compilateur Java. Cela signifie que, du point de vue de la JVM, il n'y a pas de différence entre le code Kotlin compilé et le code Java compilé. La JVM exécute ce bytecode sans savoir s'il provient du code source Kotlin ou Java.

Présentation générale

► Exécution par la JVM :

- Une fois compilé, le bytecode Kotlin est exécuté par la JVM de la même manière que le bytecode Java.
- La JVM interprète ou utilise la compilation Just-In-Time (JIT) pour convertir le bytecode en code machine au moment de l'exécution.

► Interopérabilité avec Java :

- L'une des principales caractéristiques de Kotlin est son interopérabilité à 100 % avec Java. Cela signifie que vous pouvez:
 - Appeler du code Java à partir de Kotlin
 - Appeler du code Kotlin à partir de Java
 - Mélanger du code Kotlin et du code Java dans le même projet.
- Cette interopérabilité est possible parce que les deux langages compilent dans le même format de bytecode.

Présentation générale

- ▶ Bien que Kotlin compile le même bytecode que Java, il offre plusieurs caractéristiques de langage modernes et des **améliorations syntaxiques** :
 - ▶ Sécurité des pointeurs nuls
 - ▶ Fonctions d'extension
 - ▶ Classes de données
 - ▶ Coroutines pour la programmation asynchrone
 - ▶ Syntaxe plus concise
- ▶ Ces fonctionnalités sont implémentées au niveau du compilateur, et traduites en bytecode JVM standard.

Présentation générale

- ▶ En termes de performances d'exécution, le code Kotlin se comporte généralement de la même manière que le code Java équivalent, puisqu'ils s'exécutent tous deux sur la JVM et utilisent les **mêmes optimisations du bytecode**.
- ▶ Au-delà de la JVM, Kotlin peut également être compilé en :
 - ▶ **JavaScript**, pour le développement web
 - ▶ **Code natif**, pour les plateformes comme iOS ou les systèmes embarqués.
 - ▶ Dans ces cas, les processus de compilation et d'exécution diffèrent de la cible JVM.

Historique

► **Origine et développement :**

- Kotlin a été développé par JetBrains à partir de 2010.
- Il a été dévoilé pour la première fois sous le nom de « Project Kotlin » en juillet 2011.
- Le projet a été mis en open-source sous la licence Apache 2 en février 2012.

► **Philosophie de conception :**

- Kotlin a été conçu pour être une alternative moderne, pragmatique et interopérable à Java.
- Ses principales caractéristiques sont la sécurité des pointeurs nuls, la concision et la facilité d'utilisation des outils.
- Il vise à résoudre les problèmes courants auxquels les développeurs sont confrontés et à intégrer les meilleures pratiques.

Historique

► Principales étapes :

- Kotlin 1.0, la première version stable, a été lancée le 15 février 2016.
- Lors de la conférence Google I/O 2017, Google a annoncé une prise en charge de premier ordre de Kotlin sur Android.
- En mai 2019, Google a déclaré que Kotlin était son langage préféré pour le développement d'applications Android.

► Adoption et croissance :

- De grandes entreprises comme Netflix, Uber, Twitter, Pinterest et Trello ont adopté Kotlin.
- En 2022, Meta (Facebook) a annoncé un effort pluriannuel pour faire passer ses applications Android de Java à Kotlin.

Historique

► Évolution du langage :

- Kotlin a fait l'objet de versions régulières, chaque version apportant de nouvelles fonctionnalités et des améliorations.
- Le langage suit les principes de l'évolution pragmatique, en se concentrant sur la modernité du langage, en maintenant une boucle de rétroaction avec les utilisateurs et en assurant des mises à jour régulières.

► Situation actuelle et avenir :

- En 2024, Kotlin 2.0 a été publié (le 21 mai 2024).
- Kotlin continue de gagner en popularité, en particulier dans le domaine du développement Android.
- Son avenir est prometteur, grâce au soutien de Google et à la croissance de la communauté des développeurs.

Avantages

- ▶ **Code partagé** : Kotlin Multiplatform (**KMP**) permet de partager une grande partie du code métier entre différentes plateformes (Android, iOS, Web, Desktop), réduisant ainsi la duplication de code et améliorant la maintenabilité.
- ▶ **Interopérabilité native** : Kotlin s'intègre parfaitement avec les langages natifs de chaque plateforme (Java pour Android, Swift pour iOS, JavaScript pour le Web), permettant d'utiliser les API et bibliothèques spécifiques à chaque plateforme.
- ▶ **Performances natives** : Le code Kotlin peut être compilé en code natif pour chaque plateforme, offrant des performances comparables aux langages natifs.

Avantages

- ▶ **Flexibilité** : Kotlin Multiplatform permet de choisir quelles parties du code partager et quelles parties implémenter spécifiquement pour chaque plateforme, offrant un bon équilibre entre réutilisation et optimisation.
- ▶ **Écosystème riche** : Kotlin bénéficie d'un écosystème en croissance avec de nombreuses bibliothèques multiplateforme, facilitant le développement d'applications complexes.
- ▶ **Réduction des coûts et du temps de développement** : En partageant le code entre les plateformes, on réduit le temps et les ressources nécessaires pour développer et maintenir des applications sur plusieurs plateformes.

Avantages

- ▶ **Cohérence** : Le partage de code assure une meilleure cohérence entre les versions des applications sur différentes plateformes, réduisant les divergences fonctionnelles.
- ▶ **Évolutivité** : Il est plus facile de faire évoluer une application multiplateforme, car les changements dans le code partagé se répercutent sur toutes les plateformes.
- ▶ **Tests simplifiés** : Les tests unitaires peuvent être écrits une seule fois pour le code partagé, réduisant l'effort de test tout en augmentant la couverture.
- ▶ **Support de JetBrains** : Kotlin étant développé par JetBrains, il bénéficie d'un excellent support IDE et d'outils de développement performants.

Outils

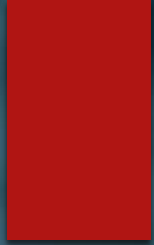
- ▶ **IntelliJ IDEA** : L'IDE phare pour le développement Kotlin, offrant un support natif et complet.
- ▶ **Android Studio** : Basé sur IntelliJ IDEA, spécialisé pour le développement Android avec Kotlin.
- ▶ **Eclipse** : Avec le plugin Kotlin, il offre un bon support pour le développement Kotlin.
- ▶ **Visual Studio Code** : Avec les extensions appropriées, il devient un excellent éditeur pour Kotlin.

Build

- ▶ **Gradle** : Système de build très populaire, particulièrement bien intégré avec Kotlin.
- ▶ **Gradle Kotlin** : Gradle basé sur la syntaxe Kotlin
- ▶ **Maven** : Outil de gestion de projet largement utilisé, compatible avec Kotlin.

Frameworks

- ▶ **Spring Boot** : Framework populaire pour le développement d'applications web en Kotlin.
- ▶ **Ktor** : Framework asynchrone pour créer des microservices et des applications web en Kotlin.
- ▶ **Android Jetpack** : Ensemble de bibliothèques pour le développement Android moderne.



Les bases

Introduction à Kotlin

- ▶ Emprunte beaucoup aux langages fonctionnels
 - ▶ Repose davantage sur l'inférence
 - ▶ Syntaxe simplifiée
 - ▶ Tout est objet
- ▶ Déclaration des données
 - ▶ Mutables: **var**
 - ▶ Non-mutables: **val**
 - ▶ Références nullables: **?**
 - ▶ Ex: `var a : Int?` // Evite les NPE
 - ▶ **Any** représente n'importe quel type
- ▶ Casts automatiques
- ▶ Point d'entrée des programmes exécutable
 - ▶ **fun** `main(args: Array<String>)`

Syntaxe Kotlin

- ▶ Conçue pour être concise et expressive.
- ▶ Déclaration de données mutables: Utilise **val** pour les variables immuables et **var** pour les variables mutables.
- ▶ Déclaration de constantes: **const val**
- ▶ Déclaration de fonction : Utilise le mot-clé **fun**.
- ▶ Pas de point-virgule : Les déclarations ne nécessitent pas de point-virgule à la fin.
- ▶ Templates : Permettent d'intégrer des expressions dans des chaînes de caractères à l'aide de **\$ {}**.
- ▶ Sécurité des pointeurs nuls: Le système de types fait la distinction entre les types "nullables" (?) et les types "non nullables".
- ▶ **Exercice:** étudier le fichier Main.kt et tester les différentes constructions syntaxiques.

Types primitifs

Type	Taille	Exemples de valeurs	Utilisation principale
Int	32 bits	42, -10, 1000	Entiers
Float	32 bits	3.14f, -0.01f	Réels, précision simple
Double	64 bits	2.71828, -123.456	Réels, haute précision
Boolean	1 bit	true, false	Logique, conditions
Char	16 bits	'A', 'Z'	Caractères
String	Variable	"Hello", "42"	Texte, messages

Tout est objet en Kotlin

Précisions sur les types primitifs

- ▶ **Sur la JVM** : Les types de base Kotlin sont automatiquement mappés sur les types Java, donc l'utilisation de types Java fonctionne.
- ▶ **Sur Kotlin/Native** : Les types Java ne sont pas disponibles. Utiliser explicitement des types Java dans du code multiplateforme ou natif empêchera la compilation ou l'exécution.
- ▶ **Bonne pratique** : Toujours utiliser les types de base Kotlin (Int, String, etc.) pour garantir la portabilité et la compatibilité multiplateforme

Conversion entre types primitifs

► Conversion explicite

- Kotlin ne réalise pas de conversions implicites entre types numériques qui pourraient entraîner une perte de précision ou des erreurs (contrairement à certains langages comme Java ou C).
- Exemple : il n'est pas possible d'assigner directement un Int à un Long ou un Double sans **conversion explicite**.

► Fonctions de conversion disponibles

- toByte()
- toShort()
- toInt()
- toLong()
- toFloat()
- toDouble()
- toChar()
- toString()

var, val et const val

► var

- Définition : Permet de déclarer une variable mutable (modifiable).
- Utilisation : On peut réassigner une nouvelle valeur **de même type** à la variable après son initialisation.

► val

- Définition : Permet de déclarer une variable en lecture seule (read-only). On ne peut pas la réassigner après son initialisation.
- Utilisation : La référence ne peut pas changer, mais l'objet pointé peut rester mutable (ex : une liste mutable).

var, val et const val

► **const val**

- Définition : Permet de déclarer une constante de compilation (connue à la compilation, invariable et inlinée dans le bytecode).
- Contraintes :
 - Doit être de type primitif ou String.
 - Doit être déclarée au niveau top-level, dans un objet ou un companion object, jamais dans une fonction ni une classe standard.
 - Sa valeur doit être connue à la compilation (pas d'appel de fonction ou d'instanciation dynamique).
- Utilisation : Pour les constantes globales ou les valeurs statiques.

Gestion des pointeurs nuls

- ▶ Kotlin fournit plusieurs mécanismes pour gérer les pointeurs nuls et travailler avec des types nullable. Ces fonctionnalités sont conçues pour éviter les exceptions de pointeur nul (NPE) et rendre le code plus sûr et plus robuste.
- ▶ En Kotlin, les types ne sont pas nullable par défaut. Pour rendre un type nullable, il faut ajouter un point d'interrogation (?) après le nom du type.

```
var hello: String = "Hello"  
var world: String? = "World"  
world = null
```

- ▶ L'opérateur d'appel sécurisé (?.) vous permet d'accéder en toute sécurité à des propriétés ou d'appeler des méthodes sur des objets potentiellement nuls.

```
var name: String? = null  
println(name?.length)
```

Gestion des pointeurs nuls

- L'opérateur Elvis (? :) fournit une valeur par défaut lorsqu'une expression est nulle.

```
var name: String? = null
val length: Int = name?.length ?: 0
```

- L'opérateur d'assertion non nulle (!!) convertit toute valeur en un type non nul et lance un NPE si la valeur est nulle. Utilisez cet opérateur avec parcimonie et uniquement lorsque vous êtes absolument certain que la valeur ne sera pas nulle.

```
val name: String? = "Kotlin"
val length: Int = name!!.length
```

Gestion des pointeurs nuls

- L'opérateur de conversion sûre (`as?`) tente de convertir une valeur en un type, en renvoyant `null` si la conversion n'est pas possible.

```
var obj: Any = "Hello"  
val str: String? = obj as? String  
val num: Int? = obj as? Int
```

- La fonction `let` peut être utilisée avec l'opérateur safe call `let` pour exécuter un bloc de code uniquement si la valeur n'est pas nulle.

```
val name: String? = "Kotlin"  
name?.let {  
    println("The name is $it")  
    println("It has ${it.length} characters")  
}
```

String : templates

- Les **String templates** en Kotlin sont une fonctionnalité qui permet d'insérer facilement des variables ou des expressions dans une chaîne de caractères, sans avoir à utiliser la concaténation classique (+) comme en Java

```
val a = 5  
val b = 3  
println("La somme de $a et $b est ${a + b}.")
```

Opérateurs

- ▶ Opérateur d'affectation: `= += -= *= /= %= &= ^= |= <<= >>=`
- ▶ Opérateurs arithmétiques: `+` `-` `*` `/` `%`
- ▶ Opérateurs unaires: `++var`, `var++`, `--var`, `var--`, `+var`, `-var`, `!var`
- ▶ Opérateurs de comparaison: `<` `>` `<=` `>=` `==` `!=`
- ▶ Opérateurs binaires: `and`, `or`, `xor`
- ▶ Opérateurs logiques: `&&` `||` `is`
- ▶ Opérateur ternaire: `val res = if (cond) 12 else 32`
- ▶ Autres opérateurs: intervalle `(..)`, index(`[]`), ensembliste (`in`)

Kotlin: Structures de contrôles

- ▶ **if** (bool exp) {...} **else if** (bool exp) {...} **else** {...}
- ▶ **for** (i **in** list) {...}
- ▶ **for** (i **in** 1..10 **step** 2) {...}
- ▶ **for** (i **in** 8 **downto** 1 **step** 2) print(i)
- ▶ **for** ((k, v) **in** map) { println("\$k -> \$v") }
- ▶ **while** (bool exp) {...}
- ▶ **when** (x) { 1 -> "one" **else** -> "unknown" }

Exercices: mettre en oeuvre ces structures sur des exemples simples

Fonctions

Fonctions

- En Kotlin, les fonctions sont déclarées à l'aide du mot-clé **fun**. Elles sont des citoyens de première ordre, ce qui signifie qu'elles peuvent être stockées dans des variables, transmises en tant qu'arguments et renvoyées par d'autres fonctions.

```
fun printInfo(name: String, age: Int) {  
    println("$name is $age years old")  
}  
  
fun greet(name: String): String {  
    return "Hello, $name!"  
}  
  
fun main() {  
    printInfo("Tom", 25)  
    println(greet("Alice"))  
}
```

Fonctions

- ▶ Si une fonction renvoie une seule expression, les accolades peuvent être omises et le corps est spécifié après le symbole =

```
fun double(x: Int): Int = x * 2
```

- ▶ Les fonctions peuvent avoir des valeurs par défaut pour les paramètres

```
fun greet(name: String, greeting: String = "Hello") {  
    println("$greeting, $name!")  
}
```

- ▶ Lors de l'appel d'une fonction, vous pouvez nommer les arguments pour une meilleure lisibilité

```
fun createUser(name: String, age: Int, isAdmin: Boolean) {  
    // function body  
}  
  
fun main() {  
    createUser(name = "Eve", age = 25, isAdmin = false)  
}
```

Fonctions

- Les fonctions peuvent accepter un nombre variable d'arguments (varargs)

```
fun sum(vararg numbers: Int): Int {  
    return numbers.sum()  
}  
  
fun main() {  
    println(sum(1, 2, 3, 4, 5))  
}
```

- Fonctions d'ordre supérieur: Fonctions prenant d'autres fonctions en paramètre ou peuvent même renvoyer une fonction

```
fun operation(x: Int, y: Int, op: (Int, Int) -> Int): Int {  
    return op(x, y)  
}  
  
fun main() {  
    val result = operation(10, 5) { a, b -> a + b }  
    println(result)  
}
```

Fonctions

- ▶ Vous pouvez ajouter des fonctions à des classes existantes sans modifier leur code source (extension)

```
fun String.addExclamation(): String {  
    return this + "!"  
}  
  
fun main() {  
    println("Hello".addExclamation())  
}
```

- ▶ Les fonctions marquées d'un infix peuvent être utilisées d'une manière plus naturelle, comme dans une langue (fonction infix)

```
infix fun Int.times(str: String) = str.repeat(this)  
  
fun main() {  
    println(3 times "Hello ")  
}
```

Fonctions en ligne et opérateurs

- ▶ Les fonctions en ligne permettent d'optimiser les performances. Lorsqu'une fonction est marquée comme étant **inline**, le compilateur remplace l'appel de fonction par le corps de la fonction à l'endroit de l'appel – ex: *inlining.kt*

```
inline fun foo(inlined: () -> Unit, noinline notInlined: () -> Unit) {...}
```

- ▶ Les **opérateurs** en Kotlin vous permettent de définir des comportements personnalisés pour les opérateurs standards (+, -, *, /, etc.) sur vos propres types.

```
data class Point(val x: Int, val y: Int) {  
    operator fun plus(other: Point) = Point(x + other.x, y + other.y)  
}
```

Opérateurs

Opérateur	Fonction à définir	Exemple d'appel
+	operator fun plus(...)	a + b
-	operator fun minus(...)	a - b
*	operator fun times(...)	a * b
/	operator fun div(...)	a / b
%	operator fun rem(...)	a % b
unary-	operator fun unaryMinus()	-a
unary+	operator fun unaryPlus()	+a

Exercices

- ▶ Écrire une fonction salutation qui prend un nom (String, valeur par défaut "Invité") et un prefixe (String, valeur par défaut "Cher"), et retourne "Bonjour [prefixe] [nom] !" en utilisant une définition en ligne avec =.
- ▶ Utiliser la fonction salutation de l'exercice précédent dans un main(), en spécifiant explicitement prefixe = "Dr." et nom = "Martin", dans un ordre inversé.
- ▶ Écrire une fonction moyenne qui prend un vararg de Double et retourne leur moyenne. Utilise une définition en ligne avec =.

Exercices

- ▶ Écrire une fonction *applyOperation* qui prend deux `Int` et une fonction `(Int, Int) -> Int`, et retourne le résultat de l'opération. Teste-la avec une addition et une multiplication.
- ▶ Ajouter une fonction d'extension `isPrime()` à `Int` qui retourne `true` si le nombre est premier.
- ▶ Écrire une fonction inline `mesurerTemps` qui exécute un bloc de code et retourne le temps d'exécution en millisecondes.
- ▶ Créer une classe `Vecteur` avec des propriétés `x` et `y`, puis surcharger l'opérateur `*` pour retourner le déterminant des deux vecteurs.

Fonctions récursives avec Tailrec

- Kotlin prend en charge la récursivité de queue, qui permet d'optimiser les fonctions récursives

```
tailrec fun factorial(n: Int, accumulator: Int = 1): Int {  
    if (n <= 1) return accumulator  
    return factorial(n - 1, n * accumulator)  
}  
  
fun main() {  
    println(factorial(5))  
}
```

- La récursivité de queue est un cas particulier de récursion dans lequel l'appel récursif est la dernière opération effectuée dans la fonction avant le retour.
- Le mot-clé tailrec en Kotlin est utilisé pour optimiser les fonctions récursives de queue. Lorsqu'une fonction est marquée par tailrec, le compilateur optimise la **récursivité** en une implémentation efficace **basée sur une boucle**.

Tailrec - Précisions

- ▶ Avantages de l'optimisation de la récursivité de queue :
 - Évite les erreurs de débordement de pile pour la récursion profonde
 - Améliore les performances en éliminant le besoin de suivre les cadres de pile précédents
 - Permet d'écrire un code récursif qui fonctionne aussi efficacement qu'un code itératif
- ▶ Conditions d'utilisation de **tailrec** :
 - La fonction doit s'appeler elle-même comme la dernière opération qu'elle effectue
 - Ne peut pas être utilisée lorsqu'il y a plus de code après l'appel récursif
 - Ne peut pas être utilisé dans les blocs **try/catch/finally**

Lambdas

- ▶ En Kotlin, les lambdas sont essentiellement des fonctions anonymes qui peuvent être traitées comme des valeurs.
- ▶ Elles permettent d'écrire un code concis et sont caractéristiques de la programmation fonctionnelle.

```
val sum: (Int, Int) -> Int = { a, b -> a + b }  
println(sum(3, 4))  
val multiply = { a: Int, b: Int -> a * b } //Using inference  
println(multiply(3, 4))  
val square: (Int) -> Int = { it * it } //Using implicit param  
println(square(5))
```

Lambdas

- Les lambdas sont souvent utilisés comme arguments de fonctions d'ordre supérieur.

```
val numbers = listOf(1, 2, 3, 4, 5)
val evenNumbers = numbers.filter { it % 2 == 0 }
println(evenNumbers)
```

- Si le dernier paramètre d'une fonction est un lambda, vous pouvez le placer en dehors des parenthèses

```
fun operation(x: Int, y: Int, op: (Int, Int) -> Int): Int {
    return op(x, y)
}

fun main() {
    val result = operation(10, 5) { a, b -> a + b }
    println(result)
}
```

Lambdas

- La dernière expression d'un lambda est automatiquement renvoyée

```
val getGreeting: (String) -> String = { name ->
    val currentHour = java.time.LocalTime.now().hour
    when {
        currentHour < 12 -> "Good morning, $name"
        currentHour < 18 -> "Good afternoon, $name"
        else -> "Good evening, $name"
    }
}
println(getGreeting("Alice"))
```


Lambdas

- Les lambdas peuvent capturer et utiliser des variables de la portée extérieure

```
fun counter(): () -> Int {  
    var count = 0  
    return { ++count }  
}  
  
val increment = counter()  
println(increment()) // Output: 1  
println(increment()) // Output: 2
```

Lambdas

- Les lambdas sont largement utilisés pour les opérations de collecte

```
val fruits = listOf("apple", "banana", "cherry", "date")  
val fruitLengths = fruits.map { it.length }  
println(fruitLengths) // Output: [5, 6, 6, 4]  
  
val totalLength = fruits.fold(0) { acc, fruit -> acc + fruit.length }  
println(totalLength) // Output: 21
```

Lambdas

- ▶ Kotlin permet de définir des lambdas avec des récepteurs, ce qui est utile pour créer des DSL.

```
val isEven: Int.() -> Boolean = { this % 2 == 0 }  
println(4.isEven()) // Output: true  
println(5.isEven()) // Output: false
```

Exercices récapitulatifs

- ▶ Écrivez une fonction qui prend deux nombres en paramètres et retourne leur produit. Appelez cette fonction avec différentes valeurs et affichez les résultats.
- ▶ Créez une fonction qui prend une chaîne de caractères en paramètre et retourne la longueur de cette chaîne. Testez votre fonction avec plusieurs exemples.
- ▶ Écrivez une fonction récursive pour calculer la somme des nombres de 1 à n , où n est passé en paramètre. Testez votre fonction avec différentes valeurs de n .
- ▶ Écrivez une fonction qui prend un tableau d'entiers et retourne un nouveau tableau contenant uniquement les nombres pairs du tableau d'origine. Utilisez une expression lambda pour filtrer les nombres pairs.

Exercices récapitulatifs

- ▶ Implémentez une fonction d'ordre supérieur qui prend une liste d'entiers et une fonction de prédicat en paramètres, et retourne une nouvelle liste contenant uniquement les éléments qui satisfont le prédicat. Testez votre fonction avec différents prédicats.
- ▶ Créez une fonction qui prend une liste de chaînes de caractères et retourne une nouvelle liste où chaque chaîne est convertie en majuscules. Utilisez une référence de fonction pour la conversion en majuscules.
- ▶ Implémentez une fonction qui prend une liste de nombres et retourne leur moyenne, mais utilisez la gestion des exceptions pour gérer le cas où la liste est vide. Utilisez une expression lambda pour le calcul de la moyenne.

Exercices récapitulatifs

- ▶ Créez une fonction d'ordre supérieur appelée "compose" qui prend deux fonctions f et g en paramètres et retourne une nouvelle fonction qui applique $f(g(x))$.
 - ▶ La fonction doit être générique et fonctionner avec n'importe quels types compatibles pour f et g. Testez votre fonction "compose" avec différentes combinaisons de fonctions.
- ▶ Ecrire une fonction $f: x \rightarrow x^2$
- ▶ Ecrire une fonction $g: x \rightarrow x + 5$
- ▶ Composer les fonctions f et g pour calculer:
 - ▶ $(2 + 5)^2$
 - ▶ $2^2 + 5$

Exercices récapitulatifs

- ▶ Ecrire une méthode récursive qui calcul le terme de rang n de la suite de Fibonacci
 - ▶ Exécuter et mesurer le temps d'exécution pour fibo(50)
 - ▶ Qu'observez-vous ?
- ▶ Améliorer la méthode en utilisant un accumulateur
 - ▶ Exécuter et mesurer le temps d'exécution pour fibo(50)
 - ▶ Qu'observez-vous ?
 - ▶ Comment peut-on améliorer l'algorithme ?
- ▶ Utilisez enfin le mot clef tailrec afin de transformer la récursion en itération
- ▶ Quelle autre méthode pour améliorer le calcul ?
- ▶ Ecrire une méthode qui affiche tous les nombres pairs de la suite de Fibonacci inférieurs ou égaux à un nombre donné en paramètre

Séquences

Séquences

- ▶ Les séquences en Kotlin sont une structure de données qui permet de représenter une suite paresseuse d'éléments.
- ▶ Elles offrent une alternative efficace aux collections standard lorsqu'il s'agit de traiter de grands ensembles de données de manière séquentielle.

```
val sequence = generateSequence(1) { it + 1 }  
val result = sequence.take(10).toList()
```

Séquences : Caractéristiques

- ▶ **Évaluation paresseuse** : Contrairement aux collections standard qui évaluent les opérations immédiatement, les séquences évaluent les éléments de manière paresseuse. Cela signifie que les éléments ne sont évalués que lorsque cela est nécessaire, ce qui peut améliorer les performances et réduire l'utilisation de la mémoire.
- ▶ **Chaînage d'opérations** : Les séquences permettent de chaîner plusieurs opérations ensemble, telles que map, filter, sort, etc. Chaque opération renvoie une nouvelle séquence, permettant ainsi de construire des pipelines de traitement de données efficaces.
- ▶ **Génération à la demande** : Les éléments d'une séquence sont générés à la demande, ce qui signifie qu'ils ne sont calculés que lorsqu'ils sont réellement nécessaires. Cela permet de travailler avec des séquences infinies ou de grandes quantités de données sans consommer trop de mémoire.

Séquences : Caractéristiques

- ▶ **Opérations terminales** : Les séquences ont des opérations terminales qui déclenchent l'évaluation réelle de la séquence et renvoient un résultat. Des exemples d'opérations terminales sont `toList()`, `toSet()`, `first()`, `last()`, `sum()`, etc.
- ▶ **Création de séquences** : Il existe plusieurs façons de créer des séquences en Kotlin, telles que `asSequence()` pour convertir une collection en séquence, `generateSequence()` pour générer une séquence basée sur une fonction, et d'autres fonctions comme `sequenceOf()`, `emptySequence()`, etc.
- ▶ **Efficacité** : Les séquences sont particulièrement efficaces lorsque vous avez de grandes quantités de données et que vous devez effectuer plusieurs opérations sur ces données. Elles minimisent le nombre d'itérations sur les données et évitent la création de collections intermédiaires inutiles.

Séquences : Caractéristiques

- **Interopérabilité avec les collections** : Kotlin fournit des fonctions d'extension pour convertir facilement entre les séquences et les collections standard. Vous pouvez utiliser `toList()` ou `toSet()` pour convertir une séquence en collection, et `asSequence()` pour convertir une collection en séquence.

```
val result = listOf(1, 2, 3, 4, 5)
    .asSequence()
    .filter { it % 2 != 0 }
    .map { it * it }
    .toList()
```

```
fun main() { new *
    val sequence = generateSequence(1) { it + 1 }
    val result = sequence.take(10).toList()
    println(result)
}
```



Séquences vs Streams

- ▶ **Natif à Kotlin** : Les séquences font partie intégrante du langage Kotlin, tandis que les flux proviennent de Java et peuvent être utilisés en Kotlin grâce à l'interopérabilité.
- ▶ **Évaluation paresseuse** : Les séquences et les flux utilisent tous deux l'évaluation paresseuse.
- ▶ **Performance** :
 - ▶ Les séquences utilisent souvent moins de mémoire lors du traitement des données.
 - ▶ Les séquences peuvent être plus efficaces grâce aux fonctions en ligne.

Séquences vs Streams

- ▶ **Parallélisme** : Les flux Java offrent un parallélisme facile grâce aux flux parallèles, que les séquences ne prennent pas en charge de manière native.
- ▶ **Sécurité des nuls** : Les séquences fonctionnent mieux avec les fonctionnalités de sécurité des nullités de Kotlin
- ▶ **Opérations terminales** : Les séquences ont généralement des opérations terminales plus simples que les collecteurs de flux.

Exercices

- ▶ Écrivez une fonction qui prend une séquence d'entiers en entrée et renvoie la somme des carrés de ces entiers en utilisant les opérations de séquence.
- ▶ Écrivez une fonction qui prend une séquence de chaînes de caractères et renvoie une nouvelle séquence contenant uniquement les chaînes de plus de 5 caractères, converties en majuscules.
- ▶ Écrivez une fonction qui génère une séquence des nombres de Fibonacci jusqu'à une valeur maximale donnée. La séquence de Fibonacci commence par 0 et 1, et chaque nombre suivant est la somme des deux précédents.
 - Pour cela, utilisez les fonctions `sequence` et `yield`
- ▶ Cf. <https://kotlinlang.org/api/core/kotlin-stdlib/kotlin.sequences/>

Classes

Présentation

- ▶ Les classes permettent de regrouper les caractéristiques communes à un groupe d'objets
- ▶ Les classes encapsulent des données et des opérations
- ▶ Les classes sont organisées au sein de paquetages hiérarchisés

```
class Person(val name: String, val age: Int) {  
    val fullName: String  
        get() = "$name $age"  
  
    init {  
        require(name.isNotBlank()) { "Name cannot be blank" }  
        require(value: age >= 0) { "Age cannot be negative" }  
    }  
  
    fun isAdult() = age >= 18  
}
```

Déclaration

- ▶ Une classe se déclare simplement par le mot clef `class`
 - ▶ `class` Person
- ▶ Par défaut une classe en Kotlin est marquée "final"
 - ▶ Pour permettre l'héritage: `open class` Shape
- ▶ Pour les POJOs (Beans): `data class` Customer(`val` name: String)
 - ▶ Génération automatique de: accesseurs, equals, copy, ...
- ▶ Pour les types énumérés: `enum class` DayOfWeek{...}
- ▶ Pour les classes scellées: `sealed class` Color
- ▶ Singleton: `object` Foo { `val` name = "XXX" }
- ▶ Possibilité de surcharger les opérateurs

Déclaration - Exemples

```
open class Shape
class Circle(val radius: Double) : Shape()
class Rectangle(val width: Double, val height: Double) : Shape()
```

```
enum class DayOfWeek {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
}
```

```
sealed class Shape {
    class Circle(val radius: Double) : Shape()
    class Rectangle(val width: Double, val height: Double) : Shape()
    class Triangle(val base: Double, val height: Double) : Shape()
}
```

Héritage

- ▶ Toutes les classes en Kotlin héritent implicitement de **Any** (similaire à l'**Object** de Java).
- ▶ Pour rendre une classe héritable, utilisez le mot-clé **open**
- ▶ Utilisez **:** pour spécifier la superclasse
- ▶ Utilisez le mot-clé **override** pour surcharger une méthode
- ▶ On peut remplacer une propriété **val** par une propriété **var**, mais pas l'inverse.
- ▶ Utilisez le mot-clé **super** pour appeler les méthodes ou les accesseurs de propriété de la superclasse
- ▶ Kotlin supporte la délégation comme alternative à l'héritage (mot-clef **by**)

```
interface Shape {  
    fun area(): Double  
}  
  
class Rectangle(val width: Double, val height: Double) : Shape {  
    override fun area() = width * height  
}  
  
class Square(val rect: Rectangle) : Shape by rect {  
    constructor(side: Double) : this(Rectangle(side, side))  
  
    override fun area() = rect.area()  
}  
  
class Circle(val radius: Double) : Shape {  
    override fun area() = Math.PI * radius * radius  
}
```

Classes abstraites

- ▶ Certaines classes ne servent qu'à regrouper des attributs / opérations communs et ne sont pas destinées à être instanciées
 - ▶ On les marque alors comme **abstract**
 - ▶ Ex: **abstract class** Shape { ... }
- ▶ Les classes abstraites sont généralement partiellement définies
 - ▶ Certaines opérations n'ont pas de méthodes et sont également marquées **abstract**

```
abstract class Shape {  
    abstract fun area(): Double  
}  
  
class Circle(val radius: Double) : Shape() {  
    override fun area() = Math.PI * radius * radius  
}  
  
class Rectangle(val width: Double, val height: Double) : Shape() {  
    override fun area() = width * height  
}
```


Classes abstraites - Exercices

- ▶ Modéliser un système pour un refuge animalier.
- ▶ Représenter différents types d'animaux, mais tous partagent certaines caractéristiques : un nom, un âge, et la capacité d'émettre un son (par exemple, aboyer ou miauler).
- ▶ Certains animaux ont des comportements spécifiques.
- ▶ Tester vos classes dans une fonction main

Énumérations

- ▶ Les énumérations en Kotlin, appelées classes d'énumérations, sont utilisées pour représenter un ensemble fixe de constantes.
- ▶ Elles sont plus puissantes que de simples collections de constantes, car ce sont des classes à part entière
- ▶ Les classes d'énumérations peuvent avoir des propriétés, des méthodes et même leur propre corps de classe

```
enum class Color(val rgb: Int) {  
    RED( rgb: 0xFF0000) {  
        override fun print() { println("This is red") }  
    },  
    GREEN( rgb: 0x00FF00) {  
        override fun print() { println("This is green") }  
    };  
  
    abstract fun print()  
}
```

Énumérations

- ▶ Les classes Enum peuvent implémenter des interfaces, ce qui les rend très flexibles
- ▶ Propriétés et méthodes intégrées :
 - **name**: Renvoie le nom de la constante de l'énumération
 - **ordinal**: renvoie la position zéro dans la déclaration de l'énumération
 - **values()**: renvoie un tableau de toutes les constantes de l'énumération
 - **valueOf(value : String)**: renvoie la constante de l'énumération avec le nom spécifié
- ▶ Les énumérations sont particulièrement utiles avec l'expression **when** de Kotlin, car elles garantissent une vérification exhaustive.
- ▶ Les énumérations renforcent la sécurité des types en limitant une variable à un ensemble prédéfini de valeurs.
- ▶ **Exercice**: Développer une application de gestion de tâches. Chaque tâche possède une priorité, qui peut être : BASSE, MOYENNE, ou HAUTE.

Classes de données

- ▶ Les classes de données sont déclarées à l'aide du mot-clé **data**.
- ▶ Elles sont conçues pour contenir des données et fournissent automatiquement plusieurs fonctions utilitaires.
 - **equals()**: Vérification de l'égalité structurelle
 - **hashCode()**: Cohérent avec equals()
 - **toString()**: renvoie une représentation sous forme de chaîne de caractères de la classe
 - **componentN()**: Fonctions de déstructuration des déclarations
 - **copy()**: Crée une copie d'une instance, permettant de modifier les propriétés.

Classes de données

- ▶ Tous les paramètres du constructeur primaire doivent être marqués comme **val** ou **var**.
- ▶ Seules les propriétés définies dans le constructeur primaire sont utilisées dans les fonctions générées automatiquement.
- ▶ Les classes de données prennent en charge les déclarations de déstructuration, ce qui permet d'extraire facilement des propriétés.
- ▶ Les classes de données peuvent hériter d'autres classes et implémenter des interfaces. Cependant, elles sont implicitement finales et ne peuvent pas être abstraites, ouvertes, scellées ou internes.
- ▶ Les classes de données ne peuvent pas hériter d'autres classes de données

```
data class Config(val host: String = "localhost", val port: Int = 8080)

fun main() {
    val config = Config()
    val (host, port) = config
    println("Host: $host, Port: $port")
}
```

Classes de données - Exercice

- ▶ Développer une application pour gérer une bibliothèque.
- ▶ Chaque livre possède un titre, un auteur et une année de publication.
- ▶ Permettre la manipulation efficace de ces objets : comparaison, création de copies, etc.

Classes scellées

- ▶ Les classes scellées sont utilisées pour représenter des hiérarchies de classes restreintes, où une valeur peut avoir l'un des types d'un ensemble limité, mais ne peut avoir aucun autre type.
- ▶ Elles sont particulièrement utiles pour représenter les machines à états ou les types de données algébriques.
- ▶ Caractéristiques principales :
 - ▶ Une classe scellée est abstraite par elle-même, elle ne peut pas être instanciée directement.
 - ▶ Les sous-classes directes d'une classe scellée doivent être déclarées dans le même fichier que la classe scellée.
 - ▶ Les sous-classes peuvent être déclarées en dehors du corps de la classe scellée.
- ▶ Exhaustivité des expressions « when » :
- ▶ L'un des principaux avantages des classes scellées est qu'elles permettent d'utiliser des expressions **when** exhaustives sans avoir besoin d'une clause **else**

```
sealed class Result {  
    class Success(val data: String) : Result()  
    class Error(val message: String) : Result()  
    object Loading : Result()  
}  
  
fun handleResult(result: Result) = when (result) {  
    is Result.Success -> println("Success: ${result.data}")  
    is Result.Error -> println("Error: ${result.message}")  
    is Result.Loading -> println("Loading...")  
}
```


Classes scellées

► Propriétés :

- Les classes scellées sont très flexibles, car chaque sous-classe peut avoir ses propres propriétés et méthodes.
- Les instances des classes scellées peuvent contenir un état.
- Les classes scellées peuvent avoir plusieurs instances du même type.
- Les classes scellées sont implicitement **abstract** et ne peuvent pas avoir le modificateur **open**.
- Vous pouvez utiliser la réflexion pour obtenir toutes les sous-classes d'une classe scellée.
- Les sous-classes d'une classe scellée peuvent faire l'objet d'un héritage supplémentaire, à moins qu'elles ne soient marquées comme **final**

► Cas d'utilisation :

- Représentation de différents états dans les machines à états
- Modélisation des types de données algébriques
- Gestion de différents types de réponses d'API
- Mise en œuvre du modèle Result pour la gestion des erreurs

Classes scellées - Exercice

- ▶ Représenter le résultat d'une opération réseau qui peut aboutir à trois états :
 - Succès, avec des données reçues (une chaîne de caractères)
 - Erreur, avec un message d'erreur
 - En cours de chargement (aucune information supplémentaire)
- ▶ Garantir que seuls ces trois états sont possibles, et exploiter la sécurité offerte par les classes scellées pour forcer le traitement de chaque cas dans un when sans avoir besoin d'un else.

Classes imbriquées

- ▶ Les classes imbriquées en Kotlin sont des classes définies à l'intérieur d'une autre classe.
- ▶ Par défaut, elles sont statiques, ce qui signifie qu'elles n'ont pas accès à l'instance de la classe extérieure.
- ▶ Points clés :
 - Déclarées à l'intérieur d'une autre classe sans aucun modificateur
 - Ne peuvent pas accéder aux membres de la classe extérieure
 - Similaires aux classes imbriquées statiques en Java

Classes internes

- ▶ Les classes internes sont des classes imbriquées non statiques. Elles ont accès aux membres de la classe externe :
 - Déclarées à l'aide du mot-clé **inner**
 - Peuvent accéder aux membres de la classe externe
 - Détiennent une référence à l'instance de la classe externe
- ▶ Les classes internes peuvent accéder aux membres des classes externes en utilisant **this@OuterClass**

Classes imbriquées et internes

Exercice

- ▶ Modéliser un ordinateur qui contient une carte graphique et un disque dur.
- ▶ La carte graphique est une classe imbriquée (elle n'a pas besoin d'accéder aux attributs de l'ordinateur).
- ▶ Le disque dur est une classe interne (il doit pouvoir accéder à la capacité totale de l'ordinateur).

Companion Objects

- Les objets compagnons en Kotlin ont une fonction similaire à celle des propriétés statiques en Java.
- Utile par exemple pour implanter le pattern singleton

```
class Configuration(val host: String = "localhost", val port: Int = 8080) {  
    companion object {  
        fun getInstance() = Configuration()  
    }  
}
```

Singletons

- Pour les singletons, utilisez le mot-clé "object"

```
object DatabaseConnection {  
    fun connect() = println("Connected to database")  
}  
  
fun main() {  
    DatabaseConnection.connect()  
}
```


Companion object et Singleton

Exercice

- ▶ Développer une application qui gère des utilisateurs.
- ▶ Chaque utilisateur a un identifiant unique, attribué automatiquement à la création.
- ▶ L'application dispose d'un journal (logger) centralisé, accessible partout dans le code.

Constructeur

- ▶ Le constructeur sert à initialiser l'état des instance
- ▶ Les classes Kotlin peuvent avoir un constructeur primaire, qui fait partie de l'en-tête de la classe
- ▶ Les classes peuvent avoir des constructeurs secondaires en utilisant le mot-clé **constructor**
- ▶ En Kotlin, vous créez une instance d'une classe en utilisant le nom de la classe suivi de parenthèses, sans le mot-clé "new".
 - ▶ Ex: `val v = Voiture(age=20, kilometers=20_000);`

```
class Person(val name: String, val age: Int) {  
  
    var weight = 0.0  
  
    val fullName: String  
        get() = "$name $age"  
  
    init {  
        require(name.isNotBlank()) { "Name cannot be blank" }  
        require(value: age >= 0) { "Age cannot be negative" }  
    }  
  
    constructor(name: String, age: Int, weight: Double) : this(name, age) {  
        this.weight = weight  
    }  
  
    fun isAdult() = age >= 18  
}
```

Constructeur - Exercices

- ▶ Développer une application pour gérer différents véhicules.
- ▶ Chaque véhicule a un modèle, une année et éventuellement une couleur.
- ▶ Vous souhaitez permettre la création de véhicules :
 - soit en spécifiant toutes les informations,
 - soit en ne donnant que le modèle et l'année (la couleur sera alors "blanc" par défaut),
 - soit en ne donnant que le modèle (l'année sera 2024 et la couleur "blanc").

Modifieurs - Visibilité

- ▶ **public** (par défaut) :
 - ▶ Visible partout
 - ▶ Visibilité par défaut si aucun modificateur n'est spécifié
- ▶ **private** :
 - ▶ Visible dans le fichier contenant la déclaration (pour les déclarations de premier niveau)
 - ▶ Visible uniquement à l'intérieur de la classe (pour les membres de la classe)
- ▶ **protected** :
 - ▶ Non disponible pour les déclarations de premier niveau
 - ▶ Visible dans la classe et ses sous-classes
- ▶ **internal** :
 - ▶ Visible à l'intérieur du même module
 - ▶ Un module est un ensemble de fichiers Kotlin compilés ensemble (par exemple, un module IntelliJ IDEA, un projet Maven/Gradle).

Modifieurs - Héritage

- ▶ **open** :
 - ▶ Permet à une classe d'être héritée de
 - ▶ Permet à un membre d'être surchargé
- ▶ **final** (par défaut) :
 - ▶ Empêche une classe d'être héritée
 - ▶ Empêche la surcharge d'un membre
- ▶ **abstract** :
 - ▶ Marque une classe comme abstraite (ne peut être instanciée)
 - ▶ Marque un membre comme n'ayant pas d'implémentation dans sa classe

Modifieurs - Propriétés

► **lateinit :**

- Permet d'initialiser des propriétés (var) non nulles en dehors d'un constructeur
- Ne peut pas être utilisé avec des types primitifs ou des propriétés nullable

► **lazy:**

- Permet d'effectuer une initialisation paresseuse (lazy initialization) d'une propriété : la valeur n'est calculée et mémorisée que lors du premier accès à la propriété, puis réutilisée à chaque accès suivant.
- C'est une forme de délégation de propriété, très utile pour optimiser les performances ou différer des calculs coûteux jusqu'à ce qu'ils soient réellement nécessaires.

```
val maPropriete by lazy {  
    // Bloc d'initialisation exécuté une seule fois  
    calculLourd()  
}
```

Modifieurs - Propriétés

► **lateinit :**

- Permet d'initialiser des propriétés (var) non nulles en dehors d'un constructeur
- Ne peut pas être utilisé avec des types primitifs ou des propriétés nullable

► **lazy:**

- Permet d'effectuer une initialisation paresseuse (lazy initialization) d'une propriété : la valeur n'est calculée et mémorisée que lors du premier accès à la propriété, puis réutilisée à chaque accès suivant.
- C'est une forme de délégation de propriété, très utile pour optimiser les performances ou différer des calculs coûteux jusqu'à ce qu'ils soient réellement nécessaires.

► Les modifieurs transient et volatil du java sont propres au fonctionnement de la JVM, ils sont remplacés dans kotlin par des annotations de même nom:

- **@Transient**: Empêche l'attribut d'être sérialisé
- **@Volatile**: Garantit qu'un thread n'accède pas à une valeur périmée

► De la même manière, il est possible d'utiliser l'annotation de même nom pour spécifier des sections critiques : **@Synchronized**

Modifieurs - Exercices

- ▶ Déclarer une classe abstraite Oeuvre avec :
 - ▶ Une propriété titre (String), visible uniquement dans la classe et ses sous-classes.
 - ▶ Une propriété abstraite auteur (String).
 - ▶ Une fonction protégée descriptionCourte() qui retourne une courte description de l'œuvre (titre et auteur).
 - ▶ Une fonction abstraite afficherInfos().
- ▶ Déclarer une classe Livre qui hérite de Oeuvre :
 - ▶ Le constructeur prend titre et auteur en paramètres.
 - ▶ Implemente la fonction afficherInfos() pour afficher la description courte.
 - ▶ Ajoute une propriété privée isbn (String) accessible uniquement dans la classe.
 - ▶ Ajoute une fonction interne isbnMasque() qui retourne les 3 premiers caractères de l'isbn suivis de "***".

Modifieurs - Exercices

- ▶ Déclare une classe ouverte Bibliotheque :
 - ▶ Possède une propriété interne catalogue (liste d'œuvres).
 - ▶ Possède une fonction ouverte ajouterOeuvre(oeuvre: Oeuvre) qui ajoute une œuvre au catalogue.
 - ▶ Possède une fonction publique afficherCatalogue() qui affiche les infos de chaque œuvre.
- ▶ Dans la fonction main() :
 - ▶ Crée une bibliothèque, ajoute-y deux livres, affiche le catalogue.
 - ▶ Tente d'accéder à la propriété isbn et à la fonction isbnMasque() depuis l'extérieur de la classe Livre (explique pourquoi cela fonctionne ou non).

Polymorphisme

- ▶ Consiste à fournir une interface commune à des objets ayant différents types
- ▶ Plusieurs types de polymorphisme
 - ▶ Ad-hoc => Surcharge ou redéfinition
 - ▶ Paramétrée => Utilisation de génériques
 - ▶ Par héritage => Une hiérarchie de classe implantant une même interface

Interfaces

- ▶ Une interface définit un contrat que des classes implanteront
- ▶ Une interface représente un comportement visible de l'extérieur
- ▶ Une interface spécifie un ensemble d'opérations implicitement **abstract**
- ▶ Les opérations sont implicitement **public**
- ▶ Les interfaces peuvent contenir des propriété / fonctions statiques

```
interface Vehicle {  
    companion object {  
        const val WHEELS = 4  
        @JvmStatic fun factory(): Vehicle = Car()  
    }  
  
    fun drive()  
}  
  
interface Medium {  
    fun transport()  
}
```

Interfaces

- ▶ Une classe peut réaliser plusieurs interfaces
- ▶ Une classe qui réalise une interface doit implanter toutes les opérations définies par cette dernière
- ▶ Quand une interface évolue, par l'ajout de fonctions, les classes qui l'implante doivent toutes être resynchronisées:
 - ▶ Implanter les méthodes ajoutées
 - ▶ Être déclarées abstract
- ▶ On peut éviter la situation où les classes qui implantent l'interface se retrouve dans un état incohérent
 - ▶ On donne une implantation par défaut à la fonction

```
interface Medium {  
    fun transport() {  
        println("Transporting people")  
    }  
}  
  
class Car : Vehicle, Medium {  
    override fun drive() {  
        println("Driving a car")  
    }  
}
```

Exercice: Développer une application pour gérer des objets connectés (smart devices). Certains objets peuvent être allumés et éteints, d'autres peuvent être réglés (par exemple, une ampoule peut changer d'intensité, un thermostat peut changer de température).

Annotations

- ▶ Métadonnées sur le code source
 - ▶ Introduites en alternative aux fichiers XML
 - ▶ Accessibles grâce à la réflexion
- ▶ Les annotations peuvent être utilisées lors de la compilation ou à l'exécution
- ▶ Cas d'utilisation :
 - ▶ Génération de code
 - ▶ Contrôles au moment de la compilation
 - ▶ Traitement au moment de l'exécution (par exemple, injection de dépendances, sérialisation)

```
@Target(AnnotationTarget.CLASS, AnnotationTarget.FUNCTION)
@Retention(AnnotationRetention.RUNTIME)
@MustBeDocumented
@Repeatable
annotation class TestCase(
    val testName: String,
    val priority: Int = 0,
    val tags: Array<String> = [],
    val expectedExceptions: Array<KClass<out Throwable>> = []
)

@TestCase(testName = "Test 1", priority = 1, tags = ["unit", "fast"])
fun test1() {
    println("Test 1")
}
```

Méta-annotations

- ▶ Il s'agit d'annotations appliquées à d'autres annotations:
 - ▶ **@Target** : Spécifie où l'annotation peut être utilisée (par exemple, classes, fonctions, propriétés).
 - ▶ **@Retention** : Définit la manière dont l'annotation est stockée (SOURCE, BINARY, RUNTIME)
 - ▶ **@Repeatable** : Permet à une annotation d'être utilisée plusieurs fois sur le même élément.
 - ▶ **@MustBeDocumented** : Indique que l'annotation doit faire partie de l'API publique.
- ▶ Cibles des annotations :
 - ▶ Les annotations peuvent cibler des éléments spécifiques tels que **CLASS**, **FUNCTION**, **PROPERTY**, etc.
- ▶ Rétention : Définit la durée de conservation des annotations :
 - ▶ **SOURCE** : Uniquement dans le code source
 - ▶ **BINAIRE** : dans le fichier .class compilé
 - ▶ **RUNTIME** : Disponible à l'exécution par réflexion

```
fun runTests(testClass: Any) {  
    val kClass = testClass::class  
    val testMethods = kClass.members  
        .filter { it.annotations.any { ann -> ann is TestCase } }  
  
    testMethods.sortedBy {  
        it.annotations.filterIsInstance<TestCase>().first().priority  
    }  
        .forEach { method ->  
            val testCase = method.annotations.filterIsInstance<TestCase>().first()  
            println("Running test: ${testCase.testName}")  
            try {  
                method.call(testClass)  
                println("Test passed")  
            } catch (e: AssertionError) {  
                println("Test failed: ${e.message}")  
            }  
            println("---")  
        }  
}  
  
fun main() {  
    runTests(test1())  
}
```


Annotations standards

- ▶ **@OptIn** : Permet l'utilisation d'API expérimentales.
- ▶ **@Deprecated** : marque le code comme étant obsolète.
- ▶ **@Suppress** : Utilisé pour supprimer les avertissements du compilateur ou d'autres caractéristiques du langage.
- ▶ **@PublishedApi** : Permet d'utiliser des membres internes à partir de fonctions publiques en ligne.
- ▶ **@JvmStatic** : Utilisé dans les objets compagnons pour l'interopérabilité Java.
- ▶ **@JvmOverloads** : Génère des méthodes surchargées pour les valeurs de paramètres par défaut en Java.

Annotations standards

- ▶ **@Throws** : Spécifie les exceptions qu'une méthode peut lancer pour l'interopérabilité Java.
- ▶ **@BuilderInference** : Aide à l'inférence de type dans les fonctions de type constructeur.
- ▶ **@ExperimentalContracts** : Marque l'utilisation des déclarations de contrats expérimentaux.
- ▶ **@RequiresOptIn** : Utilisé pour marquer les API qui requièrent un consentement explicite.
- ▶ **@DslMarker** : Utilisé pour créer des langages spécifiques à un domaine (DSL).
- ▶ **@SinceKotlin** : Spécifie la version de Kotlin dans laquelle une API a été introduite.

Génériques

- ▶ Amélioration du système de types
 - ▶ Permettant d'opérer sur des objets de différents types de manière sûre et reproductible
 - ▶ Préoccupations de capitalisation et de réutilisation
 - ▶ Vérification de la sûreté du code à la compilation
 - ▶ Très utilisé dans les collections par exemple
 - ▶ Evite des opérations de cast ou de vérification
 - ▶ S'appliquent sur les classes ou les opérations
- ▶ La généricité s'applique aux classes comme aux fonctions

```
val persons = List<Person>(10) { Person("Person $it", it) }
```

```
fun <T> printItem(item: T) {  
    println(item)  
}  
  
class Box<T>(var content: T) {  
    fun get(): T = content  
}  
  
fun main() {  
    val stringBox = Box(content: "Hello")  
    val intBox = Box(content: 42)  
}
```

Génériques

- ▶ Vous pouvez limiter les types qui peuvent être utilisés avec un générique en utilisant la clause **where** ou en spécifiant un supertype
- ▶ Lorsque vous ne connaissez pas ou ne vous souciez pas des arguments de type spécifiques, vous pouvez utiliser la projection en étoile (*).
- ▶ Kotlin vous permet d'accéder au type réel passé en tant que paramètre de type au moment de l'exécution dans les fonctions en ligne avec le mot-clef **reified**
- ▶ Les alias de type peuvent être utilisés avec les génériques pour créer des noms plus courts pour des types complexes.

```
fun <T> copyWhenGreater(list: List<T>, threshold: T): List<T> {
    where T : CharSequence,
        T : Comparable<T> {
        return list.filter { it > threshold }
    }
}
```

```
fun acceptStrings(strings: List<out CharSequence>) {
    strings.forEach { println(it) }
}

fun addComparables(dest: MutableList<in Int>) {
    dest.add(42)
}
```

Génériques - Réification

- ▶ Le mot-clef **reified** permet d'accéder au type générique réel à l'exécution dans une fonction inline.
- ▶ Cela permet d'utiliser `T::class`, `is T`, `as T`, ou de passer le type à une API de réflexion ou de sérialisation sans avoir à fournir explicitement la classe.
- ▶ C'est indispensable pour certains usages avancés de la généricité en Kotlin et n'existe pas en Java

```
inline fun <reified T> List<Any>.filterType(): List<T> =  
    this.filter { it is T }.map { it as T }
```

Génériques : Covariance

- ▶ **Usage** : Le mot-clé **out** s'utilise lorsqu'une classe générique ne fait que produire des valeurs du type générique (par exemple, une liste en lecture seule).
- ▶ **Effet** : Si une classe est déclarée avec **out T**, alors `MyClass<Child>` est considéré comme un sous-type de `MyClass<Parent>` si `Child` est un sous-type de `Parent`.
- ▶ **Illustration** :
`List<out T>` est immuable : on ne peut que lire des éléments de type `T` ou ses sous-types, jamais en ajouter

Générique: Contravariance

- ▶ **Usage** : Le mot-clé **in** s'utilise lorsqu'une classe générique ne fait que consommer des valeurs du type générique (par exemple, un comparateur).
- ▶ **Effet** : Si une classe est déclarée avec **in T**, alors `MyClass<Parent>` est considéré comme un sous-type de `MyClass<Child>`.
- ▶ **Illustration:**
`Comparator<in T>` ne reçoit que des objets de type `T` ou de ses sous-types en argument

Génériques - Exercice

- ▶ Compléter l'exercice sur la hiérarchie des animaux avec les classes et fonctions suivantes:
 - Des cages **covariantes** (peuvent contenir un type et ses sous-types)
 - Des nourrisseurs **contravariants** (peuvent nourrir un type et ses super-types)
 - Une fonction **réifiée** pour filtrer les animaux par type

Réflexion et Introspection

- ▶ Définition et objectif :
 - ▶ La réflexion est un ensemble de caractéristiques du langage et de la bibliothèque qui permet l'introspection de la structure d'un programme au moment de l'exécution.
 - ▶ Elle permet d'examiner les classes, les fonctions et les propriétés de manière dynamique.
- ▶ Composants clés :
 - ▶ KClass : Représente une classe Kotlin.
 - ▶ KFunction : Représente une fonction Kotlin
 - ▶ KProperty : Représente une propriété Kotlin
- ▶ Références de classe :
 - ▶ Obtenues à l'aide de la syntaxe `::class`
 - ▶ Exemple : `val stringClass : KClass<String> = String::class`

Réflexion et Introspection

- ▶ Références de fonctions :
 - ▶ Créées à l'aide de l'opérateur ::
 - ▶ Peuvent être utilisées comme objets de type fonction
 - ▶ Exemple : `val functionReference = ::sampleFunction`
- ▶ Références de propriétés :
 - ▶ Permettent d'accéder aux propriétés de manière dynamique
 - ▶ Permet d'obtenir ou de définir les valeurs des propriétés au moment de l'exécution
- ▶ Références de constructeurs :
 - ▶ Référencées comme les méthodes utilisant `::ClassName`

Réflexion et Introspection

```
fun displayClassInfo(clazz: Class<*>) {  
    println("Class: ${clazz.simpleName}")  
    // Display properties  
    println("Properties:")  
    clazz.kotlin.declaredMemberProperties.forEach { property ->  
        println("- ${property.name}: ${property.returnType}")  
    }  
}  
  
fun main() {  
    // Get the class object for Rectangle  
    val rectangleClass = Rectangle::class.java  
    // Display properties and methods of Rectangle  
    displayClassInfo(rectangleClass)  
}
```

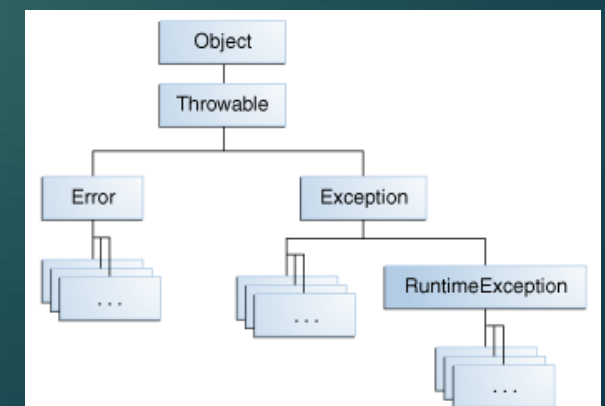
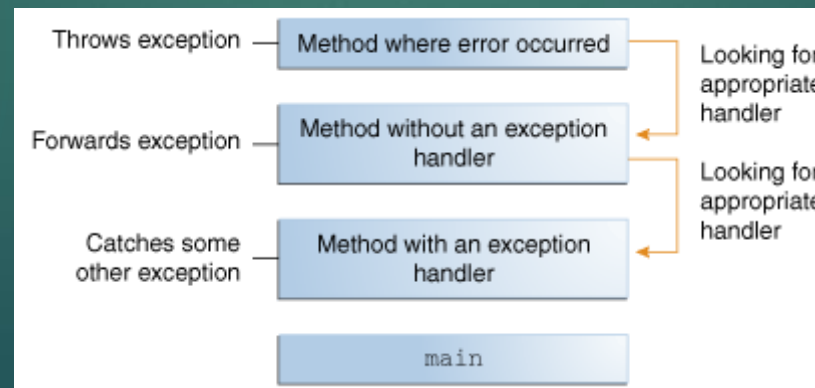
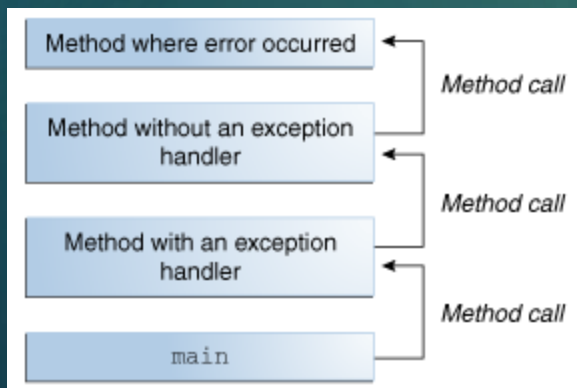
Exercices

- ▶ Créez une classe générique "Stack<T>" qui représente une pile (LIFO). Implémentez des fonctions pour push (empiler), pop (dépiler) et peek (regarder le sommet sans dépiler). Assurez-vous que les exceptions appropriées sont levées lorsque l'on essaie de dépiler une pile vide.
- ▶ Définissez une classe "Graph" qui représente un graphe dirigé à l'aide d'une liste d'adjacence. Implémentez des fonctions pour ajouter des nœuds et des arêtes, ainsi que pour effectuer des parcours en profondeur (DFS) et en largeur (BFS) sur le graphe.

Exceptions

Exceptions

- ▶ Les exceptions sont un mécanisme permettant de gérer les erreurs et autres événements exceptionnels survenant au cours de l'exécution d'un programme.
- ▶ 3 types d'exception:
 - ▶ Les exceptions anticipées
 - ▶ Les exceptions non-anticipées (erreurs)
 - ▶ Les exceptions d'exécution (runtime exception)



Exceptions anticipées

- ▶ **Définition** : Ce sont des situations exceptionnelles que le programmeur doit anticiper et gérer explicitement (par un bloc try/catch ou une clause throws).
- ▶ **Hiérarchie** : Hérite de java.lang.Exception (hors RuntimeException).
- ▶ **Obligation** : Le compilateur force le développeur à gérer ou déclarer ces exceptions.
- ▶ **Exemples courants** :
 - ▶ IOException (erreur d'entrée/sortie, fichier introuvable)
 - ▶ SQLException (erreur lors d'un accès à une base de données)
 - ▶ ClassNotFoundException (classe non trouvée lors du chargement dynamique)
- ▶ **But** : Permettre à l'appelant de réagir ou de récupérer la situation

Exceptions non vérifiées (Runtime)

- ▶ **Définition** : Ce sont des erreurs de programmation ou des situations inattendues à l'exécution, souvent dues à des bugs (mauvaise utilisation d'une API, logique erronée...).
- ▶ **Hierarchie** : Hérite de `java.lang.RuntimeException`, elle-même sous-classe de `Exception`.
- ▶ **Obligation** : Le compilateur n'oblige pas à gérer ou déclarer ces exceptions ; elles peuvent survenir à tout moment et ne sont généralement pas prévues.
- ▶ **Exemples courants** :
 - ▶ `NullPointerException` (accès à une référence nulle)
 - ▶ `ArrayIndexOutOfBoundsException` (dépassement d'indice de tableau)
 - ▶ `IllegalArgumentException` (argument de méthode invalide)
 - ▶ `ArithmeticException` (division par zéro)
 - ▶ `ClassCastException` (mauvais cast)
- ▶ **But** : Signaler des erreurs de programmation que le développeur doit corriger, pas l'utilisateur final

Erreurs

- ▶ **Définition** : Ce sont des problèmes graves liés à l'environnement d'exécution ou à la JVM elle-même, sur lesquels le programme n'a généralement aucun contrôle.
- ▶ **Hiérarchie** : Hérite de `java.lang.Error`, elle-même sous-classe de `Throwable`.
- ▶ **Obligation** : Le compilateur n'oblige pas à les gérer, et il est déconseillé de les intercepter (`catch`).
- ▶ **Exemples courants** :
 - ▶ `OutOfMemoryError` (plus de mémoire disponible)
 - ▶ `StackOverflowError` (dépassement de pile, récursion infinie)
 - ▶ `NoClassDefFoundError` (classe manquante à l'exécution)
- ▶ **But** : Indiquer un état critique du système ou de la JVM, souvent irrécupérable

Exceptions personnalisées - Règles

- ▶ **Nom explicite:** Donner un nom précis à ton exception, décrivant clairement l'erreur (ex : `InvalidAgeException`, `InsufficientFundsException`).
- ▶ **Hériter de `RuntimeException` sauf besoin particulier:** Kotlin ne distingue pas les exceptions vérifiées/non vérifiées : il est généralement recommandé d'hériter de `RuntimeException` pour les erreurs de logique ou de validation, sauf à forcer la gestion explicite de l'exception.
- ▶ **Inclure des informations utiles:** Ajouter des propriétés ou des messages détaillés pour faciliter le débogage (par exemple, la valeur fautive, le contexte, etc.).
- ▶ **Créer une hiérarchie si besoin:** Si le domaine applicatif comporte plusieurs types d'erreurs, crée une hiérarchie d'exceptions (classe de base abstraite ou sealed class, puis sous-classes spécifiques).
- ▶ **N'utiliser les exceptions personnalisées que si nécessaire:** Privilégier les exceptions standards si elles sont adaptées. Créer une exception personnalisée uniquement si cela apporte de la clarté ou une gestion plus fine des erreurs.
- ▶ **Documenter les exceptions :** dans quel cas sont-elles lancées, que signifient-elles, comment les gérer.

Erreurs – Précisions

- ▶ La classe `Error` représente des problèmes graves liés à l'environnement d'exécution ou à la JVM elle-même (par exemple, `OutOfMemoryError`, `StackOverflowError`). Ces erreurs signalent des situations anormales dont un programme ne peut généralement pas se remettre et qu'il ne devrait pas essayer d'attraper ou de gérer.
- ▶ Les exceptions personnalisées doivent hériter de `Exception` ou de ses sous-classes (`RuntimeException`), car elles sont faites pour signaler des erreurs applicatives ou métier, que le code peut raisonnablement gérer ou anticiper.
- ▶ Créer une classe héritant de `Error` reviendrait à signaler un problème fondamental du système, ce qui n'est pas le rôle d'une exception métier ou applicative. Par convention et par sécurité, il ne faut pas créer de nouvelles sous-classes de `Error` dans ses propres programmes.

Gestion des exceptions

- ▶ Vous pouvez lancer des exceptions à l'aide du mot-clé **throw**. Cela indique qu'une erreur d'exécution inattendue s'est produite.
- ▶ Vous pouvez définir des exceptions personnalisées en créant des classes qui étendent la classe **Exception** intégrée.
- ▶ Kotlin fournit des fonctions de précondition comme **require**, **check**, **assert** et **error** pour lancer automatiquement des exceptions lorsque les conditions ne sont pas remplies.
- ▶ Vous pouvez avoir plusieurs blocs **catch** pour gérer différents types d'exceptions.
- ▶ Un bloc **final** peut être utilisé pour exécuter du code, qu'une exception soit levée ou non.
- ▶ Les exceptions personnalisées vous permettent de créer des types d'erreurs spécifiques adaptés aux besoins de votre application.

```
class InvalidUserInputException(message: String) : Exception(message)

fun getInput(): String {
    val userInput = " "
    if (userInput.isBlank()) {
        throw InvalidUserInputException("User input cannot be blank")
    }
    return userInput
}

fun main() {
    try {
        getInput()
    } catch (e: InvalidUserInputException) {
        println(e.message)
    } finally {
        println("Program finished")
    }
}
```


Exercices

- ▶ Créez une classe personnalisée "InvalidAgeException" qui hérite de la classe Exception. Écrivez une fonction qui prend un âge en paramètre et lève cette exception si l'âge est inférieur à 0 ou supérieur à 120.
- ▶ Écrivez une fonction qui prend une liste d'entiers et un index en paramètres. La fonction doit renvoyer l'élément à l'index spécifié. Utilisez un bloc try-catch pour gérer l'exception IndexOutOfBoundsException qui peut être levée si l'index est hors limites.
- ▶ Créez une classe "BankAccount" avec des méthodes pour déposer et retirer de l'argent. Levez une exception personnalisée "InsufficientFundsException" si une tentative de retrait est effectuée avec un montant supérieur au solde du compte.



Fonctions standards

Présentation

- ▶ Ces fonctions sont souvent appelées fonctions de portée (scope functions) et fonctions utilitaires ; elles facilitent l'écriture d'un code plus lisible, concis et sûr, en manipulant des objets dans un contexte temporaire ou conditionnel.
- ▶ Motivations
 - **Lisibilité** : Regrouper des opérations liées à un même objet sans répéter son nom.
 - **Sécurité** : Gérer facilement les valeurs nulles et les vérifications conditionnelles.
 - **Configuration** : Initialiser ou modifier un objet de façon fluide.
 - **Chaining** : Enchaîner des opérations de manière élégante.

Fonctions

► **let**

- Usage : Transformation, exécution conditionnelle (null safety).
- Retourne : Le résultat du bloc.

► **also**

- Usage : Exécuter des effets de bord sans modifier l'objet (log, validation...).
- Retourne : L'objet d'origine.

► **apply**

- Usage : Configurer ou initialiser un objet.
- Retourne : L'objet d'origine.

```
var name: String? = "Alice"
val longueur = name?.let { it.length }
println("longueur = $longueur")
```

```
val liste = mutableListOf(1, 2, 3)
.also { println("Avant ajout : $it") }
.apply { add(4) }
.also { println("Après ajout : $it") }
```

```
val personne = Personne().apply {
    nom = "Bob"
    age = 30
}
```

Fonctions

► run

- Usage : Calcul ou transformation sur un objet, initialisation complexe.
- Retourne : Le résultat du bloc.

► with

- Usage : Grouper des opérations sur un même objet, surtout pour des objets déjà existants.
- Retourne : Le résultat du bloc.

```
val longueur = "Kotlin".run { length * 2 }
```

```
val sb = StringBuilder()  
val result = with(sb) {  
    append("Hello, ")  
    append("world!")  
    toString()  
}
```

Fonctions

► **takeIf**

- Usage : Retourne l'objet si la condition est vraie, sinon null (filtrage sur un seul objet).
- Retourne : L'objet ou null.

► **takeUnless**

- Usage : Retourne l'objet si la condition est fausse, sinon null (filtrage inverse).
- Retourne : L'objet ou null.

```
val input1 = "Kotlin"  
val nonVide = input1.takeIf { it.isNotEmpty() } // "Kotlin" ou null
```

```
val input2 = ""  
val vide = input2.takeUnless { it.isNotEmpty() } // "" ou null
```

Résumé

- ▶ **let** : Utilisé pour la null safety ou la transformation temporaire.
- ▶ **also** : Pour les effets de bord sans modifier l'objet (logging, validation).
- ▶ **apply** : Pour initialiser/configurer un objet lors de sa création.
- ▶ **run** et **with** : Pour grouper des opérations et retourner un résultat calculé.
- ▶ **takelf/takeUnless** : Pour filtrer ou valider un objet dans une chaîne d'opérations, souvent combiné avec **let** pour exécuter un bloc si la condition est satisfaite.

Exercices

- ▶ Gérer un objet Utilisateur qui possède les propriétés suivantes : nom (String), email (String), age (Int), et actif (Boolean).
- ▶ Les objectifs sont les suivants:
 - Initialiser un utilisateur,
 - Modifier ses propriétés,
 - Appliquer des validations,
 - Afficher des informations selon certains critères,
 - Logger certaines opérations.

Exercices

- ▶ Initialisation fluide: Créer une instance mutable de Utilisateur en utilisant **apply** pour initialiser toutes ses propriétés.
- ▶ Effet de bord et logging: Utiliser **also** pour afficher dans la console un message après la création de l'utilisateur.
- ▶ Modification conditionnelle: Utiliser **let** pour augmenter l'âge de l'utilisateur de 1 si l'utilisateur est actif (null safety ou filtrage conditionnel).
- ▶ Validation avec takelf/takeUnless:
 - ▶ Utiliser **takelf** pour ne garder l'utilisateur que si son email contient un "@".
 - ▶ Utiliser **takeUnless** pour afficher un avertissement si l'âge est inférieur à 18 ans.
- ▶ Transformation et affichage
 - ▶ Utiliser **run** pour créer une chaîne de présentation de l'utilisateur (ex : "Alice (alice@mail.com), 30 ans").
 - ▶ Utiliser **with** pour grouper plusieurs affichages ou opérations sur l'utilisateur.

Exercices

- ▶ Que se passe-t-il si l'email ne contient pas de "@" ? Et si l'âge est inférieur à 18 ?
- ▶ Modifier le code pour utiliser un utilisateur inactif ou avec un email invalide, et observe le comportement des différentes fonctions de portée.
- ▶ Ajouter un effet de bord supplémentaire (log, modification, etc.) avec `also` ou `let`.



Bonnes pratiques

Bonnes pratiques

- ▶ Utilisez **val** plutôt que **var** autant que possible pour favoriser l'immuabilité et réduire les effets secondaires.
- ▶ Tirez parti des fonctions d'extension pour étendre les fonctionnalités des classes existantes sans les modifier.
- ▶ Utilisez les classes de données pour les objets qui servent principalement à contenir des données.
- ▶ Profitez de la sécurité des pointeurs nuls de Kotlin en évitant l'utilisation excessive de l'opérateur `!!`.
- ▶ Utilisez les expressions `when` pour remplacer les longues chaînes `if-else`, en particulier pour le filtrage de types.

Bonnes pratiques

- ▶ Adoptez une approche fonctionnelle en utilisant les fonctions d'ordre supérieur et les lambdas pour un code plus concis et expressif.
- ▶ Utilisez les coroutines pour gérer les opérations asynchrones et la concurrence de manière plus simple et plus efficace.
- ▶ Tirez parti des fonctions inline pour améliorer les performances, en particulier pour les fonctions d'ordre supérieur courtes.
- ▶ Utilisez les propriétés de délégation pour réutiliser la logique commune des propriétés.
- ▶ Adoptez le principe de "fail-fast" en utilisant **require()**, **check()** et **assert()** pour valider les entrées et les états.

Bonnes pratiques

- ▶ Utilisez les classes scellées pour représenter des hiérarchies fermées de types.
- ▶ Préférez les objets compagnons aux méthodes et propriétés statiques de Java.
- ▶ Utilisez les fonctions à portée pour créer des DSL (Domain Specific Languages) lisibles et expressifs.
- ▶ Tirez parti de la déstructuration pour travailler avec des objets complexes de manière plus lisible.
- ▶ Utilisez les fonctions de covariance (**out**) et de contravariance (**in**) pour créer des API plus flexibles avec des génériques.

Fail-Fast

► **require()**

- Utilisation : Vérifier les arguments d'entrée d'une fonction.
- Comportement : Lance une `IllegalArgumentException` si la condition est fausse.
- Quand l'utiliser : Au début d'une fonction pour valider les paramètres d'entrée.
- `require(age >= 0) { "L'âge ne peut pas être négatif" }`

► **check()**

- Utilisation : Vérifier l'état interne d'un objet ou les conditions d'exécution.
- Comportement : Lance une `IllegalStateException` si la condition est fausse.
- Quand l'utiliser : Pour vérifier que l'état interne d'un objet est valide avant d'exécuter une opération.
- `check(balance >= amount) { "Solde insuffisant" }`

Fail-Fast

► **assert()**

- Utilisation : Vérifier des conditions qui devraient toujours être vraies.
- Comportement : Lance une `AssertionError` si la condition est fausse, mais uniquement si les assertions sont activées (elles sont désactivées par défaut en production).
- Quand l'utiliser : Pour le débogage et les tests, pas pour le contrôle du flux de programme en production.
- `assert(area >= 0.0) { "L'aire ne peut pas être négative" }`

► **error()**

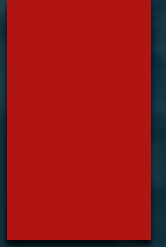
- Utilisation : Lever une exception pour signaler une erreur irrécupérable ou une condition qui ne devrait jamais se produire.
- Comportement : Lance toujours une `IllegalStateException` avec le message spécifié, indépendamment du mode d'exécution (debug ou production).
- Quand l'utiliser : Pour signaler des états invalides ou impossibles dans le programme, arrêtant immédiatement l'exécution. À utiliser lorsqu'une condition est si grave qu'elle ne permet pas la poursuite normale du programme.
- `error("Cette condition ne devrait jamais se produire")`

TODO

- ▶ La fonction **TODO()** est une fonction intégrée à Kotlin qui est utilisée pour marquer du code incomplet ou qui doit encore être implémenté. Elle est généralement utilisée comme un rappel pour les développeurs.
- ▶ Lorsque **TODO()** est appelée, elle lève toujours une exception `NotImplementedError`. Cela signifie que si du code contenant un appel à **TODO()** est exécuté, il se terminera immédiatement avec cette exception.
- ▶ **TODO()** prend un argument de chaîne optionnel qui permet de fournir une description de ce qui doit être fait. Cette description apparaîtra dans le message d'exception.

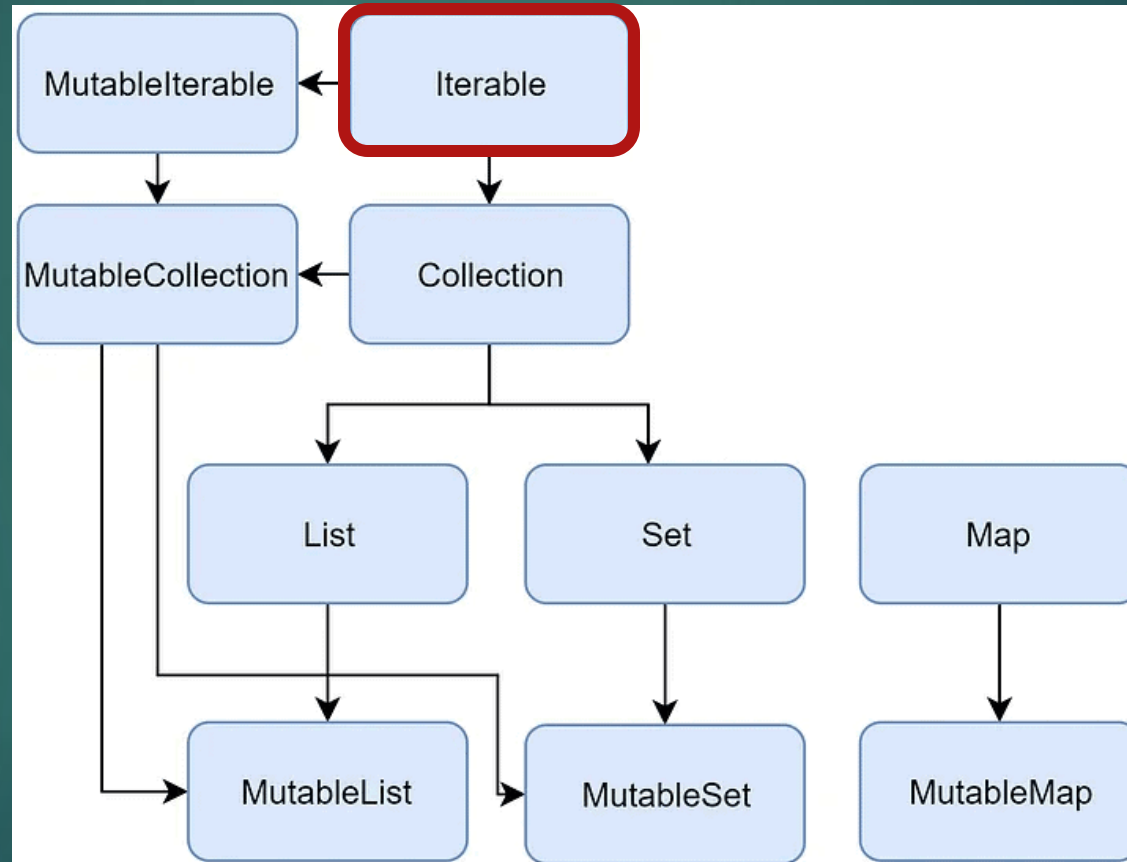
Bonnes pratiques - Exercices

- ▶ Améliorer le code précédent avec les fonctions fail-fast et la fonction `TODO()`



Structures de données

Vue d'ensemble



Collections

- ▶ Ensemble de classes et d'interfaces implantant les structures de données les plus communes
- ▶ 2 grandes catégories de collection
 - ▶ Liste – collection ordonnée
 - ▶ Ensemble – collection non ordonnée, à éléments uniques
- ▶ Définis dans le paquetage `kotlin.collections`

Collections

- ▶ Kotlin fournit des implantations pour les types de collection basiques
 - ▶ Mutables ou non-mutables
 - ▶ Exple: `val nums = mutableListOf("One", "Two", "Three")`
 - ▶ Exple: `val numsMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3)`
 - ▶ Cf. <https://kotlinlang.org/docs/collections-overview.html>
- ▶ Les collections itérables ont un itérateur accessible pour le mot-clef `it`
- ▶ Opérations: Filtres, Transformations, Groupage, Agrégation
 - ▶ Cf. <https://kotlinlang.org/docs/collection-operations.html>

```
val numbers = listOf("one", "two", "three", "four")
numbers.forEach {
    println(it)
}
```


Listes

- ▶ En Kotlin, les listes constituent un élément essentiel du cadre de collection du langage. Elles permettent de stocker des collections ordonnées d'éléments.
- ▶ **Listes immuables :**
 - Créées à l'aide de la fonction **listOf()**.
 - Elles ne peuvent pas être modifiées après leur création (c'est-à-dire qu'aucun élément ne peut être ajouté ou supprimé).
- ▶ **Listes mutables :**
 - Créées à l'aide de la fonction **mutableListOf()**.
 - Elles peuvent être modifiées après leur création (c'est-à-dire que des éléments peuvent être ajoutés, supprimés ou mis à jour).

Listes

► Opérations usuelles

- Accès aux éléments : `myList[index]`
- Itération: `for (e in myList) { println(e) }`
- Ajout: `myList.add(e)`, `myList.addAll(otherList)`
- Suppression: `myList.remove(e)`, `myList.removeAt(index)`
- Mise à jour; `myList[index] = e`
- Vérifications: `myList.isEmpty()`, `myList.contains(e)`

► Fonctions d'ordre supérieur

- Filtre: `myList.filter { it.startsWith("A") }`
- Transformation: `myList.map { it.length }`
- Tri: `myList.sorted()`, `myList.sortedBy { it.length }`
- Réduction: `myList.reduce { acc, s -> acc + s }`

Listes - Implantations

Implémentation	Points forts	Points faibles
ArrayList	Accès rapide par index, ajout fin rapide	Insertion/suppression début/milieu lent
LinkedList	Insertion/suppression début/milieu rapide	Accès par index lent
ArrayDeque	Ajout/retrait rapide aux deux extrémités	Pas d'accès direct par index
CopyOnWriteArrayList	Thread-safe pour lecture	Écritures coûteuses

Listes - Exercice

- ▶ Création d'une liste immutable
 - ▶ Créer une liste immutable appelée `courses` contenant au moins trois éléments (par exemple : "Pommes", "Pâtes", "Lait").
 - ▶ Tenter d'ajouter un nouvel élément à cette liste et observe le résultat (explique pourquoi).
- ▶ Création d'une liste mutable
 - ▶ Créer une liste mutable appelée `coursesMutable` contenant les mêmes éléments que la liste immutable.
 - ▶ Ajouter "Œufs" à la liste mutable.
 - ▶ Supprimer "Pâtes" de la liste mutable.
 - ▶ Modifier le premier élément de la liste mutable en "Bananes".
- ▶ Conversion entre listes
 - ▶ Créer une nouvelle liste immutable à partir de la liste mutable modifiée.
 - ▶ Tenter de modifier cette nouvelle liste immutable (explique pourquoi c'est possible ou non).
- ▶ Afficher le contenu de chaque liste après chaque opération.

Ensembles (Set)

- ▶ Un ensemble est une collection qui contient des éléments uniques, ce qui signifie qu'il n'autorise pas les valeurs dupliquées.
- ▶ Ensemble immuable
 - **Définition:** Un ensemble immuable est une collection en lecture seule qui ne peut être modifiée après sa création.
 - **Création:** La fonction `setOf()` permet de créer un ensemble immuable.
 - **Caractéristiques:** Une fois créé, il n'est pas possible d'ajouter, de supprimer ou de mettre à jour les éléments d'un ensemble immuable.
- ▶ Ensemble mutable
 - **Définition:** Un ensemble mutable peut être modifié après sa création, c'est-à-dire qu'il est possible d'ajouter ou de supprimer des éléments.
 - **Création:** La fonction `mutableSetOf()` permet de créer un ensemble mutable.
 - **Caractéristiques:** Il est possible d'effectuer des opérations telles que l'ajout ou la suppression d'éléments.

Ensembles (Set)

► Opérations usuelles:

- Les opérations CRUD sont les mêmes que les listes
- Union: `val u = set1 + set2`
- Intersection: `val u = set1.intersect(set2)`

Sets - Implantations

Implémentation	Ordre conservé	Tri automatique	Performance	Utilisation typique
LinkedHashSet	Oui	Non	Bonne, stable	Ordre d'insertion important
HashSet	Non	Non	Très rapide	Performance, pas d'ordre requis
TreeSet (Java)	Oui (trié)	Oui	Moins rapide	Besoin d'un ensemble trié

Sets - Exercices

- ▶ Création d'un ensemble immuable
 - ▶ Créer un ensemble immuable contacts contenant trois noms différents (par exemple : "Alice", "Bob", "Charlie").
 - ▶ Tenter d'ajouter un nouveau nom à cet ensemble et observe le résultat (explique pourquoi).
- ▶ Création d'un ensemble mutable
 - ▶ Créer un ensemble mutable contactsMutable contenant les mêmes éléments que l'ensemble immuable.
 - ▶ Ajouter "Diane" à l'ensemble mutable.
 - ▶ Supprimer "Bob" de l'ensemble mutable.
 - ▶ Tenter d'ajouter "Alice" à nouveau et observe le résultat.
- ▶ Conversion entre ensembles
 - ▶ Convertir l'ensemble mutable modifié en ensemble immuable.
 - ▶ Tenter de modifier ce nouvel ensemble immuable (explique pourquoi c'est possible ou non).
- ▶ Afficher le contenu de chaque ensemble après chaque opération.

Sets: Exercice

- ▶ Compléter le code ci-dessous (collections.kt) pour:
 - ▶ Trier par âge et par nom en utilisant la fonction **sortedWith**
 - ▶ Lister tous les hommes dont l'âge est supérieur à 41 ans
 - ▶ Lister tous les hommes dont l'âge est compris entre 28 et 41 ans
 - ▶ Implanter un arbre binaire de recherche afin d'accélérer la recherche

```
data class Man(val name: String, val age: Int)

fun main() {
    val mens = mutableListOf<Man>()
    mens.add(Man("Michel", 32))
    mens.add(Man("Paul", 28))
    mens.add(Man("Joseph", 41))
    mens.add(Man("Michel", 45))
    mens.add(Man("Paul", 53))
    mens.add(Man("Albert", 41))
}
```

Parcours d'objets itérables

- ▶ **Boucle for-in:** La manière la plus idiomatique et concise en Kotlin pour parcourir un objet itérable
- ▶ **Boucle while avec itérateur:** Pour un contrôle plus bas niveau (par exemple, suppression d'éléments pendant l'itération)
- ▶ **forEach et autres fonctions d'ordre supérieur:** Pour un style fonctionnel et concis
- ▶ **ListIterator pour les listes:** Permet d'itérer dans les deux sens, de modifier ou d'ajouter des éléments pendant l'itération
- ▶ Cf. Exemples dans le fichiers iterations.kt

Fonctions standards pour itérables

► Parcours

- **forEach** : Exécute une action pour chaque élément.
- **forEachIndexed** : Idem, mais avec l'index.

► Recherche et test

- **contains, containsAll** : Vérifie la présence d'un ou plusieurs éléments.
- **any, all, none** : Teste si au moins un, tous, ou aucun élément ne satisfait un prédicat.
- **find, firstOrNull, lastOrNull** : Recherche le premier/dernier élément correspondant à un prédicat.

► Filtrage

- **filter, filterNot, filterIndexed** : Garde (ou exclut) les éléments selon un prédicat.
- **filterNotNull** : Élimine les éléments nuls.

Fonctions standards pour itérables

► Transformation

- **map**, **mapIndexed**, **mapNotNull** : Transforme chaque élément de la collection.
- **flatMap** : Aplati les résultats de la transformation (liste de listes → liste simple).
- **groupBy** : Regroupe les éléments selon une clé.
- **associateBy**, **associateWith** : Transforme la collection en map.

► Tri et ordonnancement

- **sorted**, **sortedBy**, **sortedDescending** : Trie la collection selon l'ordre naturel ou un critère.

► Agrégation et réduction

- **sum**, **average**, **count**, **minOrNull**, **maxOrNull** : Calculs sur la collection.
- **reduce**, **fold** : Agrège les éléments en une seule valeur

Fonctions standards pour itérables

► Sous-collections

- **take, takeWhile, drop, dropWhile** : Prend ou ignore un certain nombre d'éléments.
- **chunked, windowed** : Découpe la collection en morceaux ou en fenêtres glissantes.

► Modifications

- **add, remove, clear, retainAll, removeAll** : Ajout, suppression, nettoyage, filtrage sur place

► Opérations avec destination

- **filterTo, mapTo, groupByTo** : Permettent de spécifier une collection de destination pour les résultats

Exercices

- Soit la liste nombres: `val nombres = listOf(2, 7, 4, 9, 6, 1, 8)`
 - Afficher tous les nombres pairs de la liste, un par ligne.
 - Afficher `true` si tous les nombres sont supérieurs à 0, sinon `false`.
 - Créer une nouvelle liste contenant le carré de chaque nombre.
 - Calculer la somme de tous les nombres impairs.
 - Trouver le premier nombre supérieur à 5, ou affiche «Aucun» si aucun élément ne correspond.
 - Trier la liste dans l'ordre décroissant et affiche le résultat.
 - Découper la liste nombres en sous-listes de 3 éléments (chunks).
 - Créer une liste contenant la somme de chaque fenêtre de 2 éléments consécutifs.
 - Créer une `MutableList` contenant les éléments de nombres.
 - Supprimer tous les nombres inférieurs à 5.
 - Ajouter 10 et 12 à la liste.
 - Trier la liste dans l'ordre croissant.
 - Calculer le produit de tous les éléments de nombres.

Dictionnaires (Map)

- ▶ Un dictionnaire est une collection de paires clé-valeur où chaque clé est unique et associée à une valeur spécifique. Les dictionnaires sont utiles pour stocker des données qui peuvent être rapidement récupérées à l'aide d'une clé
- ▶ Dictionnaire immuable
 - **Définition:** Un dictionnaire immuable est une structure de données en lecture seule qui ne peut être modifiée après sa création.
 - **Création:** La fonction `mapOf()` permet de créer un dictionnaire immuable.
 - **Caractéristiques:** Une fois créée, il n'est pas possible d'ajouter, de supprimer ou de mettre à jour des éléments dans un dictionnaire immuable.
- ▶ Dictionnaire mutable
 - **Définition:** Un dictionnaire mutable peut être modifiée après sa création, c'est-à-dire qu'il est possible d'ajouter, de supprimer et de mettre à jour des éléments.
 - **Création:** La fonction `mutableMapOf()` permet de créer un dictionnaire mutable.
 - **Caractéristiques:** Vous pouvez effectuer des opérations telles que l'ajout ou la suppression de paires clé-valeur.

Dictionnaires (Map)

► Opérations usuelles:

- Lecture: `myMap.get(key)`
- Ajout: `myMap[key] = value`, `myMap.put(key, value)`
- Suppression: `myMap.remove(key)`
- Itération: `for ((k, v) in myMap) { println("${k} : ${v}") }`
- Vérification: `myMap.containsKey(key)`, `myMap.containsValue(value)`
- Filtre: `myMap.filterKeys { it.startsWith("A") }`
- Transformation: `myMap.mapValues { it.value * 2 }`

Maps - Implantations

Implémentation	Ordre conservé	Tri automatique	Mutabilité	Usage principal
LinkedHashMap	Oui	Non	Oui	Ordre d'insertion, usage courant
HashMap	Non	Non	Oui	Performance, pas d'ordre requis
TreeMap	Oui (trié)	Oui	Oui	Map triée par clé
MapBuilder	N/A	N/A	Oui (transitoire)	Construction efficace avant immutable
Map (immutable)	Oui	Non	Non	Lecture seule, sécurité

Fonctions standards pour maps

► Accès et récupération

- **get**(key) ou [key] : Récupère la valeur associée à une clé, retourne null si la clé n'existe pas.
- **getValue**(key) : Récupère la valeur, lève une exception si la clé n'existe pas.
- **getOrElse**(key) { ... } : Retourne la valeur ou le résultat d'un lambda si la clé est absente.
- **getOrElseDefault**(key, defaultValue) : Retourne la valeur ou une valeur par défaut si la clé n'existe pas.

► Parcours et itération

- **for** ((key, value) in map) : Parcours toutes les paires clé/valeur.
- **forEach** { (key, value) -> ... } : Applique une action à chaque entrée.
- **map.keys** et **map.values** : Accès direct aux ensembles des clés ou des valeurs

Fonctions standards pour maps

► Transformation

- **mapKeys** { ... } : Transforme les clés (en gardant les valeurs).
- **mapValues** { ... } : Transforme les valeurs (en gardant les clés).
- **map** { ... } : Transforme chaque entrée en un nouvel objet, retourne une liste.

► Filtrage

- **filter** { (key, value) -> ... } : Garde les entrées qui satisfont une condition.
- **filterKeys** { ... } : Filtre selon la clé uniquement.
- **filterValues** { ... } : Filtre selon la valeur uniquement.

Fonctions standards pour maps

► Recherche et test

- **containsKey**(key), **containsValue**(value) : Vérifie la présence d'une clé ou d'une valeur.
- **isEmpty**(), **isNotEmpty**() : Teste si la map est vide ou non

► Modification (pour MutableMap)

- **put**(key, value) : Ajoute ou remplace une entrée.
- **remove**(key) : Supprime une entrée.
- **putAll**(map) : Ajoute toutes les entrées d'une autre map.
- **clear**() : Vide la map

Fonctions standards pour maps

► Groupement et comptage

- **groupBy** : Permet de regrouper des éléments d'une collection en une map selon un critère.
- **groupingBy().eachCount()** : Compte les occurrences de chaque élément et retourne une map

► Autres fonctions utiles

- **entries** : Retourne un ensemble de paires clé/valeur.
- **toList()**, **toMap()** : Conversion entre map et liste de paires.
- **plus/minus** : Ajoute ou retire des entrées (retourne une nouvelle map, non destructive).

Maps - Exercices

- Soit la map: `val notes = mapOf("Alice" to 15, "Bob" to 12, "Charlie" to 18)`
 - Afficher la note de "Alice" et de "Eve" (en utilisant `getOrElse` pour Eve, valeur par défaut 0).
 - Vérifier si "Bob" est dans la map (`containsKey`).
 - Afficher toutes les clés et toutes les valeurs séparément.
 - Créer une nouvelle map où chaque note est doublée (`mapValues`).
 - Créer une nouvelle map où chaque clé est en majuscules (`mapKeys`).
 - Créer une nouvelle map où chaque clé est en majuscules et chaque note est multipliée par 10 (`map + toMap`).
 - Garder uniquement les élèves ayant une note supérieure ou égale à 15 (`filterValues`).
 - Garder uniquement les entrées dont la clé commence par "A" ou "B" (`filterKeys`).
 - Calculer la moyenne des notes.
 - Trouver l'élève ayant la meilleure note (utilise `maxByOrNull` sur les entrées).
 - Afficher le nom de tous les élèves ayant la note maximale.
 - Créer une `MutableMap` à partir de la map `notes`.
 - Ajouter un nouvel élève "Diane" avec la note 17.
 - Modifier la note de "Alice" à 16.
 - Supprimer "Bob" de la map.
 - Afficher la map après chaque opération.

API Standard

I/O

- ▶ 2 types de Flux:
 - ▶ Entrants : InputStream (byte), Reader (char): Fichier, audio, Objet, Pipe, Buffer, etc
 - ▶ Sortants: OutputStream (byte), Writer (char): Fichier, Filtre, Objet, Pipe, etc.
- ▶ Les versions bufferisées de ces flux permettent d'éviter les appels directs à l'OS
 - ▶ Améliore les performances (temps d'accès, accès concurrents, etc.)
- ▶ Pour faciliter la lecture des fichiers, la version 8 de java a introduit l'objet Scanner
 - ▶ Se charge de convertir le contenu d'un fichier en tokens (séparés par des espaces par défaut).

I/O

- ▶ Écriture sur la sortie standard: `print()`, `println()`
- ▶ Lecture sur l'entrée standard: `readLine()`
- ▶ L'API `java.io.File` offre des facilités pour manipuler des fichiers
- ▶ L'API `java.nio.file.Files` offre des facilités pour manipuler des ensembles de fichiers ou de dossier
- ▶ Il est important de noter que **Kotlin fournit également des extensions et des fonctions utilitaires pour travailler avec les fichiers**, ce qui rend l'utilisation de `java.io.File` plus pratique et idiomatique en Kotlin.
- ▶ Ces extensions et fonctions utilitaires sont définies dans le package `kotlin.io`, et elles opèrent sur les objets `java.io.File`.

<https://kotlinlang.org/api/core/kotlin-stdlib/>

Exercices

- ▶ Ecrire une classe Titanic avec les méthodes suivantes
 - ▶ loadData(): charge les données dans un tableau
 - ▶ Utiliser la classe Scanner avec les contraintes suivantes:
 - ▶ ";" utilisé comme séparateur de champs
 - ▶ Le deuxième champs doit être interprété comme un booléen
 - ▶ Le quatrième champs doit être interprété comme une énumération (homme | femme)
 - ▶ Seul les 5 premiers champs sont nécessaires
 - ▶ head(n: int) et tail(n: int) qui renvoient respectivement les n premières / dernières lignes du tableau dans un format compréhensible
 - ▶ getPassengers(): renvoie la liste de l'intégralité des entrées de la table

Exercices

- ▶ Dans la méthode main, utiliser les séquences pour faire les requêtes suivantes:
 - ▶ Âge moyen des hommes
 - ▶ Civilité de chaque personne dans le format suivant: titre | prénom | nom
 - ▶ Moyenne des âges par sexe
 - ▶ Nombre de personnes par catégories d'âge
 - ▶ Moins de 20 ans, 20 /40 ans, 40 / 60 ans, plus de 60 ans
 - ▶ Afficher les entrées correspondantes aux femmes célibataires voyageant en première classe trié par âge
- ▶ Pour cet exercice, vous devrez écrire une énumération Sex et une classe de donnée Passenger

JDBC

- ▶ Interface permettant d'accéder aux bases de données SQL
 - ▶ Implantations fournies par les fournisseurs de SGBD
 - ▶ Définie dans les paquetages java.sql et javax.sql
 - ▶ Le gestionnaire de pilotes a la responsabilité de chargé les bonnes implantations pour requêter les bases de données

```
try {
    // Establish a connection
    connection = DriverManager.getConnection(url)

    // Create the Man table if it doesn't exist
    val createTableQuery = """
        CREATE TABLE IF NOT EXISTS Man (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT NOT NULL,
            age INTEGER NOT NULL
        )
    """.trimIndent()
    connection.createStatement().execute(createTableQuery)
} catch (e: Exception) {
    e.printStackTrace()
} finally {
    // Ensure the connection is closed even if an error occurs
    connection?.close()
}
```


Internationalisation

- ▶ Adaptation aux caractéristiques locales des environnements d'exécution
 - ▶ Langue
 - ▶ Monnaie
 - ▶ OS
 - ▶ ...
- ▶ Utilisation des classes "Locale" et "ResourceBundle" pour internationaliser ses applications
 - ▶ La JVM est instanciée avec un objet Local par défaut, que l'on peut accéder par la méthode `Local.getDefault()`
 - ▶ La classe `ResourceBundle` s'appuie sur des fichiers dictionnaires postfixé par la langue et le pays
- ▶ Cf. Fichier `i18n.kt` pour l'exemple

Exercices : Client Http

- ▶ Il vous est demandé pour cet exercice de requêter le web sémantique afin de:
 - ▶ Récupérer le titre et l'année de parution de tous les albums des Beatles
 - ▶ Vous utiliserez pour cela le point d'accès SPARQL suivant:
 - ▶ <https://query.wikidata.org/sparql>
 - ▶ Générez une page HTML présentant les résultats sous la forme d'un tableau
 - ▶ Vous devrez pour cela utiliser l'API java.xml.transform

```
val client = HttpClient.newBuilder().followRedirects(HttpClient.Redirect.NORMAL).build()
val url = URI("https://query.wikidata.org/sparql?query=${URLEncoder.encode(query, "UTF-8")}")
val request = HttpRequest.newBuilder().GET().uri(url).header("Accept", "application/sparql-results+xml").build()
val response = client.send(request, HttpResponse.BodyHandlers.ofString())
println(response.body())
```

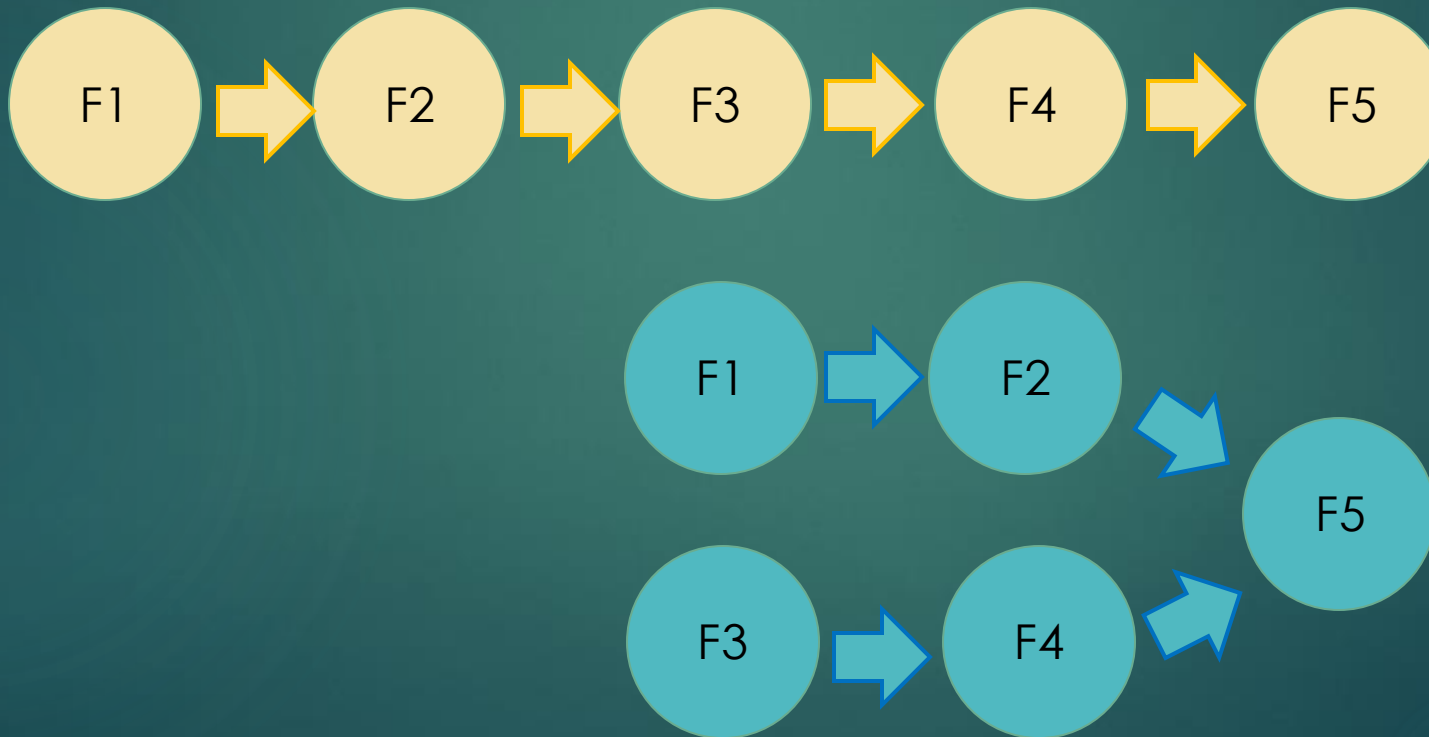
Concurrence

Concurrence

- ▶ Concepts fondamentaux: Process (processus lourd) et Thread (processus léger)
- ▶ En java, l'exécution de la JVM est représenté par un processus
 - ▶ De fait, on ne manipule que des threads en java, qui nécessitent des mécanismes de communication / synchronisation plus simples
 - ▶ Les threads partagent le même espace mémoire, celui de leur processus
 - ▶ Depuis la version 9, il est possible d'avoir plus d'information sur le processus qui exécute la JVM (pid, ppid, état, etc.)
- ▶ Parallèle ou concurrent ?
 - ▶ Concurrence: parallélisme potentielle, i.e. possibilité d'exécuter du code en parallèle
 - ▶ Parallèle: Relatif à l'architecture de la machine (SISD, SIMD, MISD, MIMD)
- ▶ Pourquoi paralléliser ?
 - ▶ Essentiellement pour des raisons de performances
 - ▶ Pour tirer pleinement des avantages offerts par la machine

Présentation

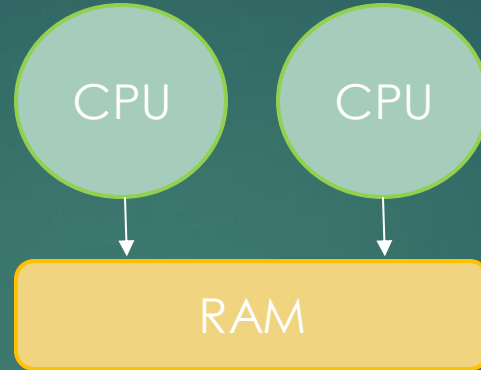
- ▶ Basé sur l'analyse du séquençement et des dépendances
- ▶ La difficulté vient de l'accès exclusif aux ressources partagées
 - ▶ Surtout quand ils sont exécutés sur des architectures parallèles



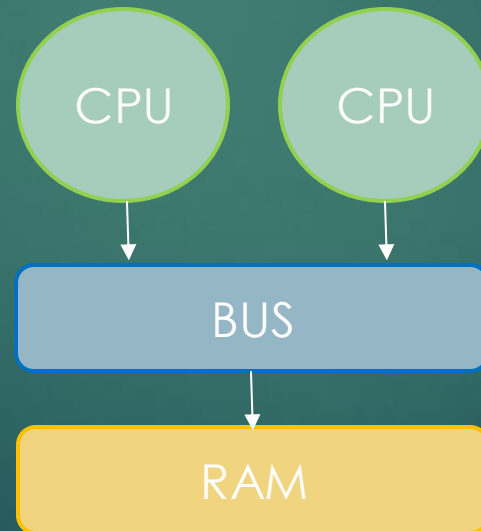
Présentation

► Taxinomie de Flynn

		Instruction	
		Simple	Multiple
Data	Simple	SISD	MISD
	Multiple	SIMD	MIMD



UMA: Uniform Memory Access



SMP: Symmetric Multiprocessing

Problème de la mémoire partagée: **cohérence des caches**

Dans le cas d'architectures distribuées (NUMA, Distributed Memory Access), la cohérence ne peut être assurée qu'au prix de gros efforts.

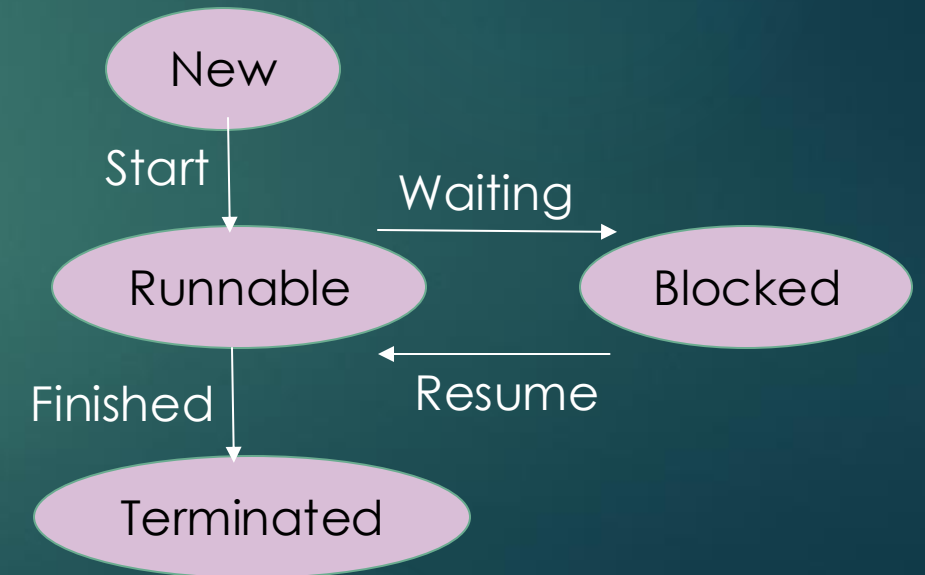
Présentation

- ▶ Ordonnancement assuré par l'OS



- ▶ Différentes stratégies d'ordonnancement

- ▶ Pour maximiser les débits
- ▶ Pour maximiser l'égalité d'accès aux ressources
- ▶ Pour minimiser les temps d'attente
- ▶ Pour minimiser la latence



Etats d'un thread

Présentation

- ▶ Lors de l'exécution d'un programme
 - ▶ La JVM exécute un thread principal (main)
 - ▶ Le thread principal peut exécuter d'autres threads (enfants)
 - ▶ Un thread peut exécuter d'autres threads (arborescence)
 - ▶ Un thread doit attendre que ses enfants soient terminés avant de se terminer (join)
- ▶ En plus des 4 états vus dans la diapositive précédente, java ajoute deux états qui sont des variants de l'état Blocked
 - ▶ Waiting – un thread attend qu'un autre thread ait fini une action
 - ▶ Timed_Waiting - Même chose que waiting mais dans un temps borné
 - ▶ La méthode statique `Thread.currentThread()` renvoie une référence vers le thread en cours d'exécution

Présentation

- ▶ Un thread est caractérisé par
 - ▶ Un identifiant unique (long)
 - ▶ Un nom (généré si non spécifié - Thread-N)
 - ▶ Une priorité, de 1 à 10, de valeur 5 par défaut
- ▶ 2 manières de spécifier un thread
 - ▶ Implanter l'interface Runnable
 - ▶ Pratique si la classe doit étendre une autre classe
 - ▶ Pratique si plusieurs threads partagent le même code
 - ▶ Étendre la classe Thread
- ▶ Un thread peut être détaché (daemon)
 - ▶ Le parent qui l'a créé n'a pas attendre qu'il soit terminé avant de se terminer
 - ▶ `setDaemon(true)`
 - ▶ À utiliser avec attention car cela peut laisser des ressources dans un état incohérent

Coroutines

- ▶ Les coroutines sont une fonctionnalité puissante de Kotlin qui vous permet d'écrire du code asynchrone et concurrent de manière séquentielle.
- ▶ Elles permettent d'écrire du code non bloquant qui semble être synchrone, ce qui facilite le raisonnement et la maintenance.

Coroutines - Caractéristiques

- ▶ **Exécution asynchrone** : Les coroutines vous permettent d'exécuter du code de manière asynchrone, ce qui signifie que l'exécution peut être suspendue et reprise plus tard sans bloquer le fil d'exécution principal. Cela est particulièrement utile pour exécuter des tâches de longue durée, telles que des requêtes réseau ou des opérations de base de données, sans geler l'interface utilisateur.
- ▶ **Légèreté** : Les coroutines sont des threads légers gérés par le moteur d'exécution Kotlin. Elles ont une empreinte mémoire plus faible que les threads traditionnels et peuvent être créées et gérées plus efficacement.
- ▶ **Suspension des fonctions** : Les coroutines introduisent le concept de fonctions suspendues. Il s'agit de fonctions qui peuvent être interrompues et reprises ultérieurement sans bloquer le fil d'exécution. Les fonctions suspendues sont marquées par le mot-clé **suspend** et ne peuvent être appelées qu'à l'intérieur d'une coroutine ou d'une autre fonction suspendue.

Coroutines - Caractéristiques

- ▶ **Constructeurs de coroutines** : Kotlin fournit des constructeurs de coroutines, tels que **launch** et **async**, pour créer et démarrer des coroutines. Le constructeur **launch** est utilisé pour lancer et oublier une coroutine, tandis que le constructeur **async** est utilisé lorsque vous devez renvoyer un résultat de la coroutine.
- ▶ **Portée des coroutines** : Les coroutines sont exécutées dans une portée spécifique, qui définit leur cycle de vie et leur comportement d'annulation. Les portées les plus courantes sont **GlobalScope** (pour les coroutines de niveau supérieur) et **CoroutineScope** (pour les coroutines liées à un cycle de vie spécifique, tel qu'un composant d'interface utilisateur).
- ▶ **Concurrence structurée** : Kotlin favorise une concurrence structurée, ce qui signifie que les coroutines sont généralement lancées dans un périmètre spécifique et sont automatiquement annulées lorsque ce périmètre est annulé. Cela permet de gérer le cycle de vie des coroutines et d'éviter les fuites.

Coroutines vs Threads

► **Efficacité des ressources :**

- Coroutines : Elles sont extrêmement légères et permettent de créer des milliers, voire des millions de coroutines sans impact significatif sur les performances. Elles consomment très peu de mémoire, généralement quelques dizaines d'octets chacune.
- Les threads : Ils sont plus gourmands en ressources. Chaque thread nécessite son propre espace de pile (souvent plusieurs mégaoctets) et implique davantage de frais généraux lors de la création et de la gestion.

► **Évolutivité :**

- Coroutines : Elles sont beaucoup plus évolutives en raison de leur légèreté. Vous pouvez facilement exécuter de nombreuses coroutines simultanément sans surcharger les ressources du système.
- Threads : Le nombre de threads pouvant être créés simultanément est limité en raison de leurs exigences plus élevées en matière de ressources.

► **Changement de contexte :**

- Coroutines : Utilisent le multitâche coopératif, elles cèdent volontairement le contrôle. Le changement de contexte est ainsi beaucoup plus rapide.
- Threads : S'appuient sur le multitâche préemptif géré par le système d'exploitation, ce qui est plus coûteux en termes de performances.

Coroutines vs Threads

► **Contrôle précis :**

- Coroutines : Elles offrent un contrôle plus fin de l'exécution, ce qui facilite la mise en œuvre de mesures telles que l'annulation coopérative et les délais d'attente.
- Threads : Ils offrent un contrôle moins fin et nécessitent souvent des mécanismes de synchronisation supplémentaires.

► **Intégration avec les API asynchrones :**

- Coroutines : Elles s'intègrent facilement aux API asynchrones existantes en suspendant les fonctions et les continuations.
- Threads : Nécessitent souvent des wrappers ou des adaptateurs supplémentaires pour fonctionner avec les API asynchrones.

► **Modèle de mémoire :**

- Coroutines : Simplifient le modèle de mémoire pour la programmation concurrente. L'état mutable partagé reste un problème, mais les coroutines permettent de le gérer plus facilement grâce à une concurrence structurée.
- Threads : Exigent une prise en compte attentive du modèle de mémoire et nécessitent souvent des mécanismes de synchronisation explicites pour éviter les conditions de concurrence.

Coroutines vs Threads

► **Simplicité et lisibilité :**

- Coroutines : Elles permettent d'écrire du code asynchrone de manière séquentielle, ce qui le rend plus facile à lire et à comprendre. Les opérations asynchrones complexes peuvent être écrites comme si elles étaient synchrones.
- Threads : Nécessitent souvent des structures de code plus complexes, comme des callbacks ou des machines d'état, pour gérer les opérations asynchrones, ce qui peut conduire à un code moins lisible.

► **Annulation et gestion des exceptions :**

- Coroutines : Elles offrent une prise en charge intégrée de la concurrence structurée, ce qui facilite la gestion des annulations et des exceptions dans le cadre de plusieurs opérations simultanées.
- Threads : La gestion des annulations et des exceptions entre les threads est plus complexe et plus sujette aux erreurs.

► **Indépendance de la plate-forme :**

- Coroutines : Les coroutines sont une fonctionnalité du langage Kotlin et fonctionnent de manière cohérente sur différentes plateformes (JVM, Android, JavaScript, Native).
- Threads : La mise en œuvre peut varier d'une plateforme à l'autre, ce qui peut nécessiter un code spécifique à la plateforme.

Job

- ▶ Dans les coroutines Kotlin, un Job représente une unité de travail qui peut être annulée.
 - Il s'agit d'un concept fondamental dans les coroutines et joue un rôle crucial dans la gestion du cycle de vie et de l'annulation des coroutines.
- ▶ Un Job est une interface qui représente une tâche annulable ou une coroutine.
 - Il est responsable de la gestion du cycle de vie d'une coroutine et permet l'annulation et les relations parent-enfant.

Job – États

- ▶ **New**: Le job a été créé mais n'a pas encore commencé.
- ▶ **Active** : Le job est en cours d'exécution ou suspendu.
- ▶ **Completing** : Le job est en cours d'achèvement.
- ▶ **Completed**: Le job est terminé, soit avec succès, soit avec une exception.
- ▶ **Cancelling**: Le job est en cours d'annulation.
- ▶ **Cancelled**: Le job a été annulé.

Jobs - Création

- ▶ Un Job peut être créé explicitement à l'aide du constructeur `Job()` ou implicitement lors du lancement d'une coroutine à l'aide de **launch** ou **async**
- ▶ Les jobs peuvent former des relations parent-enfant. Lorsqu'une coroutine est lancée dans le contexte d'une autre coroutine, le Job de la coroutine enfant devient un enfant du Job de la coroutine parent.
 - Cela permet l'annulation hiérarchique et la propagation des erreurs.

Jobs - Annulation

- ▶ Un Job peut être annulé à l'aide de la fonction `cancel()`. Lorsqu'un Job est annulé, il passe à l'état "Cancelling" et lance le processus d'annulation pour tous ses travaux enfants.
- ▶ Lorsqu'un Job parent est annulé, tous ses Jobs enfants sont également annulés de manière récursive.
 - Cela permet de s'assurer que l'ensemble de la hiérarchie des coroutines est correctement nettoyée et qu'aucune coroutine orpheline n'est laissée en cours d'exécution.
- ▶ Si une exception non capturée se produit dans une coroutine, elle est propagée à son Job parent.
 - Le Job parent s'annule alors lui-même et propage l'exception plus haut dans la hiérarchie jusqu'à ce qu'elle soit traitée ou qu'elle atteigne la coroutine racine.

Jobs - Synchronisation

- ▶ La fonction `join()` peut être utilisée pour attendre la fin d'un job. Elle suspend la coroutine appelante jusqu'à ce que le job et tous ses jobs enfants soient terminés.

```
fun main() = runBlocking {  
    val result = async {  
        delay(1000) // Simulate a long-running task  
        "Hello, Coroutines!"  
    }  
    println("Doing some other work...")  
    println(result.await())  
}
```

```
fun main() {  
    val job = Job()  
    val scope = CoroutineScope(job)  
    scope.launch {  
        println("Hello")  
    }  
    job.cancel()  
}
```

Scope

- ▶ Les scopes permettent une concurrence structurée, un paradigme de programmation qui garantit que :
 - ▶ Les coroutines enfant sont annulées lorsque la portée de leur parent est annulée
 - ▶ Les exceptions dans les coroutines enfant se propagent à leur parent
 - ▶ Les ressources sont correctement nettoyées
- ▶ Il existe 2 scopes par défaut (GlobalScope et CoroutineScope)
 - ▶ Mais vous pouvez créer un scope d'application personnalisé à l'aide du constructeur CoroutineScope
 - ▶ `val scope = CoroutineScope(Dispatchers.Default + Job())`

Scope

► **GlobalScope :**

- C'est un scope de haut niveau qui n'est lié à aucun Job spécifique.
- Les coroutines lancées dans le GlobalScope peuvent s'exécuter tant que l'application est en cours d'exécution.
- Il faut l'utiliser avec précaution car il peut entraîner des fuites de mémoire s'il n'est pas géré correctement, puisque les coroutines dans ce scope ne sont pas liées au cycle de vie d'un composant spécifique.

► **CoroutineScope :**

- C'est une interface qui définit une portée pour les nouvelles coroutines.
- Son principal objectif est de garder une trace de toutes les coroutines créées en son sein et de les annuler lorsque le scope lui-même est annulé.
- On peut créer un CoroutineScope personnalisé à l'aide du constructeur `CoroutineScope()`. C'est utile pour définir une portée pour un composant ou une fonctionnalité spécifique de l'application.
- Les coroutines lancées dans un CoroutineScope sont liées au Job de ce scope. Quand le Job du scope est annulé, toutes ses coroutines sont aussi annulées.

Scope

- ▶ Les champs d'application fournissent un contexte pour les coroutines, qui peut inclure des éléments tels que :
 - ▶ **Job** : Contrôle le cycle de vie de la coroutine
 - ▶ **Dispatcher** (répartiteur) : Détermine le thread sur lequel la coroutine s'exécute
 - ▶ **CoroutineName** : Attribue un nom à la coroutine pour le débogage.
 - ▶ **ExceptionHandler** : Gère les exceptions non capturées
- ▶ **Bonnes pratiques** :
 - ▶ Utiliser des scopes pour gérer le cycle de vie des coroutines
 - ▶ Éviter d'utiliser GlobalScope dans le code de production
 - ▶ Annuler les scopes lorsqu'ils ne sont plus nécessaires afin d'éviter les fuites.
 - ▶ Utiliser la concurrence structurée pour gérer les relations parent-enfant entre les coroutines.

Dispatchers

- ▶ Dans les coroutines Kotlin, les dispatchers déterminent le thread ou le pool de threads sur lequel une coroutine doit être exécutée.
 - Ils définissent le contexte d'exécution des coroutines et jouent un rôle crucial dans la gestion de la concurrence et du parallélisme du code basé sur les coroutines.
- ▶ Un dispatcher est chargé de distribuer l'exécution de la coroutine à un thread spécifique ou à un pool de threads. Il détermine l'endroit où la coroutine s'exécutera et la manière dont elle sera programmée.

Dispatchers - Types

▶ **Dispatchers.Main :**

- ▶ Exécute les coroutines sur le thread principal d'une application, généralement utilisé pour les tâches liées à l'interface utilisateur.
- ▶ Dans Android, il correspond au thread principal de l'application.
- ▶ Dans les applications JVM, il utilise une boucle d'événements à un seul thread.

▶ **Dispatchers.IO :**

- ▶ Conçu pour les tâches à forte intensité d'E/S, telles que les opérations réseau, les E/S de fichiers ou les interactions avec les bases de données.
- ▶ Il utilise un pool de threads partagé avec un grand nombre de threads pour gérer efficacement les tâches d'E/S bloquantes.

Dispatchers - Types

▶ **Dispatchers.Default :**

- ▶ Convient aux tâches intensives de l'unité centrale ou aux opérations coûteuses en termes de calcul.
- ▶ Utilise un pool de threads partagé avec un nombre de threads égal au nombre de cœurs de l'unité centrale.

▶ **Dispatchers.Unconfined :**

- ▶ Exécute les coroutines dans le thread de l'appelant jusqu'au premier point de suspension, puis reprend dans le thread de la fonction suspendue.
- ▶ Doit être utilisé avec prudence car il peut entraîner un comportement inattendu s'il n'est pas utilisé correctement.

Dispatchers - Création / modification

- ▶ Vous pouvez spécifier le dispatcher lorsque vous lancez une coroutine à l'aide des fonctions **launch** ou **async**
 - `launch(Dispatchers.Main) { /* ... */ }`
 - `async(Dispatchers.IO) { /* ... */ }`
- ▶ Vous pouvez changer le contexte d'exécution d'une coroutine à l'aide de la fonction **withContext**


```
fun main() {  
    runBlocking {  
        launch(Dispatchers.Main) {  
            val result = withContext(Dispatchers.IO) {  
                // Perform I/O operation  
            }  
            // Update UI with the result  
        }  
    }  
}
```


Dispatchers - Customisation

- ▶ Par défaut, les coroutines héritent du dispatcher de leur coroutine mère. Si aucun dispatcher n'est explicitement spécifié, la coroutine utilisera le même dispatcher que son parent.
- ▶ Outre les répartiteurs intégrés, vous pouvez créer des dispatchers personnalisés à l'aide des classes **ExecutorCoroutineDispatcher** ou **ThreadPoolDispatcher**. Cela vous permet de définir vos propres pools de threads ou d'utiliser des instances d'**Executor** existantes.

Contexte

- ▶ Dans les coroutines Kotlin, un `CoroutineContext` représente le contexte dans lequel une coroutine est exécutée.
- ▶ Il s'agit d'un ensemble d'éléments divers qui définissent le comportement et l'environnement d'une coroutine.

```
fun main() = runBlocking {  *  
    // Création d'un contexte personnalisé  
    val customContext = Dispatchers.Default + CoroutineName("MonContextePersonnalisé")  
  
    // Lancement d'une coroutine avec le contexte personnalisé  
    val job = launch(customContext) {  
        // Accès aux éléments du contexte  
        val dispatcher = coroutineContext[CoroutineDispatcher]  
        val name = coroutineContext[CoroutineName]?.name  
  
        println("Coroutine en cours d'exécution sur ${Thread.currentThread().name}")  
        println("Nom de la coroutine : $name")  
        println("Dispatcher utilisé : $dispatcher")  
  
        delay(1000) // Simulation d'un travail long  
        println("Travail terminé")  
    }  
  
    job.join() // Attente de la fin de la coroutine  
}
```

Contexte - Caractéristiques

- ▶ Job + Dispatcher
- ▶ CoroutineName :
 - Attribue un nom à une coroutine à des fins de débogage et d'identification.
 - Aide à comprendre l'objectif ou l'origine d'une coroutine
 - Exemple : `CoroutineName(« MaCoroutine »)`
- ▶ CoroutineExceptionHandler :
 - Définit la manière dont les exceptions non capturées au sein d'une coroutine doivent être gérées.
 - Permet de personnaliser le comportement de gestion des exceptions
 - Exemple : `CoroutineExceptionHandler { _, exception -> println(« Caught $exception ») }`

Contexte - Caractéristiques

- ▶ Les éléments d'un CoroutineContext peuvent être combinés à l'aide de l'opérateur +.
 - `val context = Dispatchers.Default + CoroutineName("MyCoroutine") + Job()`
- ▶ Les coroutines héritent par défaut du contexte de leur parent.
 - Cela signifie qu'une coroutine enfant aura les mêmes Job, Dispatcher et autres éléments que son parent, à moins qu'ils ne soient explicitement surchargés.
- ▶ Les éléments d'un CoroutineContext peuvent être remplacés lors du lancement d'une nouvelle coroutine.
- ▶ Les éléments d'un CoroutineContext sont accessibles à l'aide de l'opérateur []
- ▶ Les opérateurs + et - peuvent être utilisés pour ajouter ou supprimer des éléments d'un CoroutineContext

Gestion des exceptions

► Propagation des exceptions :

- Lorsqu'une exception est levée dans une coroutine, elle est propagée jusqu'au premier gestionnaire d'exceptions rencontré dans la hiérarchie des coroutines.
- Si aucun gestionnaire d'exceptions n'est trouvé, l'exception est traitée par le `CoroutineExceptionHandler` par défaut, qui généralement annule la coroutine parente et propage l'exception.

► `CoroutineExceptionHandler` :

- C'est un élément clé de la gestion des exceptions dans les coroutines.
- Il définit une politique de gestion des exceptions non gérées.
- Vous pouvez fournir votre propre `CoroutineExceptionHandler` pour personnaliser le traitement des exceptions non gérées.

```
val handler = CoroutineExceptionHandler { _, exception ->
    println("Caught $exception")
}

val job1 = launch(Dispatchers.Default + handler) {
    val userData = fetchUserData(userId)
    delay(1500)
    throw Exception("Oops!")
    println("User data: $userData")
}
```

Exercices

- ▶ Écrivez une fonction suspendue qui simule un délai de 1 seconde, puis imprime un message. Appelez cette fonction depuis une coroutine lancée dans le scope global.
- ▶ Créez deux fonctions suspendues qui simulent des opérations asynchrones (par exemple, `fetchUserData()` et `fetchUserPosts()`). Utilisez `launch` pour exécuter ces fonctions en parallèle et affichez les résultats.
- ▶ Implémentez une fonction qui lance 10 coroutines, chacune attendant un délai aléatoire entre 1 et 5 secondes avant d'imprimer son numéro. Utilisez un `Job` pour attendre que toutes les coroutines soient terminées avant d'afficher un message de fin.
 - Utilisez pour cela la fonction **`repeat(n) {...}`**

Channel

- ▶ Un channel est une structure de données qui permet la communication entre différentes coroutines.
- ▶ Il fournit un moyen pour les coroutines d'envoyer et de recevoir des valeurs de manière asynchrone, sans avoir à se synchroniser explicitement ou à partager un état mutable.

Channel : Caractéristiques

- ▶ **Communication asynchrone** : Les channels permettent aux coroutines d'envoyer et de recevoir des valeurs de manière asynchrone. Une coroutine peut envoyer une valeur dans le channel sans attendre qu'une autre coroutine la reçoive immédiatement.
- ▶ **Producteur-Consommateur** : Les channels sont souvent utilisés dans un modèle producteur-consommateur, où une ou plusieurs coroutines produisent des valeurs et les envoient dans le channel, tandis qu'une ou plusieurs autres coroutines consomment ces valeurs à partir du channel.
- ▶ **Tampons (Buffers)** : Les channels peuvent être créés avec une capacité tampon spécifiée. Cela permet aux producteurs d'envoyer des valeurs dans le channel sans être bloqués, tant que le tampon n'est pas plein. Les consommateurs peuvent recevoir des valeurs du channel sans être bloqués, tant que le tampon n'est pas vide.

Channels : Caractéristiques

- **Suspension** : Les opérations d'envoi et de réception sur un channel sont suspensives. Si un producteur essaie d'envoyer une valeur dans un channel plein (sans tampon), il sera suspendu jusqu'à ce qu'un consommateur reçoive une valeur. De même, si un consommateur essaie de recevoir une valeur d'un channel vide, il sera suspendu jusqu'à ce qu'un producteur envoie une valeur.
- **Fermeture** : Un channel peut être fermé pour indiquer qu'aucune autre valeur ne sera envoyée. Les consommateurs peuvent détecter la fermeture du channel et terminer leur traitement en conséquence.

```
launch {  
    for (i in 1 ≤ .. ≤ 5) {  
        channel.send(i)  
    }  
    channel.close()  
}  
  
launch {  
    for (value in channel) {  
        println("Received: $value")  
    }  
}
```



Exercices

- ▶ Créez une fonction qui simule le téléchargement de fichiers. Utilisez `async` pour télécharger 5 fichiers en parallèle, puis affichez le temps total de téléchargement.
 - Utilisez les fonctions `awaitAll` et `measureTimeMillis`
- ▶ Créez une fonction qui simule une requête API avec un délai aléatoire et une possibilité d'échec. Utilisez `withTimeout` pour annuler la requête si elle prend plus de 2 secondes. Gérez les exceptions appropriées.
- ▶ Implémentez un pool de workers en utilisant un channel pour distribuer des tâches à un nombre fixe de coroutines. Chaque tâche doit simuler un traitement avec un délai aléatoire.

Exercice: Data race

- ▶ Exécutez le programme `race.kt`
 - ▶ Que constatez-vous ?
- ▶ Modifier le nombre de boucle à 1.000.000
 - ▶ Que constatez-vous ?
 - ▶ Quel est le problème ?
 - ▶ Comment y remédier ?

Mutex

- ▶ Dans les cas simples, on peut faire usage des types de données atomiques
 - ▶ Définis dans le paquetage `kotlin.concurrent`
 - ▶ Accès automatiquement protégé par des verrous
- ▶ On peut utiliser le mot clef **synchronized** sur un bloc pour déclarer une section critique
 - ▶ Il faut préciser sur quel objet poser le verrou
- ▶ Lock ou synchronized ?
 - ▶ Synchronized est sûr et facile à mettre en œuvre
 - ▶ Lock apporte beaucoup plus de flexibilité et améliore les performances

Exercice – Data race

- ▶ Apportez une première modification en utilisant un type de donnée atomique
- ▶ Apportez une autre modification en utilisant le mot-clef **synchronized**
 - ▶ En se synchronisant sur un objet quelconque
 - ▶ En se synchronisant sur le compteur lui-même
 - ▶ Dans les deux cas, que constatez-vous ?

Mutex

▶ Reentrant lock

- ▶ Peut être verrouillé plusieurs fois
 - ▶ Doit être alors déverrouillé autant de fois qu'il a été verrouillé
 - ▶ Utile notamment dans le cas des fonctions récursives
- ▶ Fournit une méthode non-bloquante du lock
 - ▶ Permet à un Job d'éviter une attente active avant l'obtention du verrou
 - ▶ L'opération tryLock() retourne vrai si le thread a pu prendre le verrou
 - ▶ Elle renvoie faux sinon et le thread continue son exécution, rendant le programme plus performant

```
f1() {  
    lock();  
    ...  
    unlock();  
}  
  
f2() {  
    lock();  
    f1();  
    unlock();  
}
```

Exercice – Data Race

- ▶ Apportez une nouvelle modification utilisant un lock réentrant
 - ▶ Qu'observez-vous ?
 - ▶ Quel est le problème ?
 - ▶ Comment améliorer la situation ?
 - ▶ Comment évitez une attente active ?

Mutex

- ▶ Reader-Writer Lock permet de poser des verrous:
 - ▶ En lecture partagée - `readLock()`
 - ▶ En écriture exclusive – `writeLock()`
 - ▶ Intéressant quand le nombre de lecteurs est largement supérieur au nombre d'écrivains
 - ▶ Ex: accès à une base de données
 - ▶ Mécanismes de synchronisation plus complexes, à utiliser avec parcimonie

Exercice – Data Race

- ▶ Modifiez le code de manière à avoir:
 - ▶ 8 threads ayant accès en lecture au compteur
 - ▶ 2 threads ayant accès en écriture au compteur

Mutex: Problèmes

► Deadlock

- Interblocage mutuel
 - Ex: Le diner des philosophes
- Un thread ayant acquis des verrous se termine avant d'avoir relâcher ces verrous

► Starvation (famine)

- Un thread pouvant avoir une priorité moindre n'arrive jamais à acquérir les verrous
- Un grand nombre de threads accède à des ressources limitées

► Livelock

- Chaque thread, en contribuant à résoudre des problèmes de deadlock finit par bloquer les autres threads
- Difficile à détecter et à déboguer

► Solutions possibles:

- Veiller à bien ordonnancer l'acquisition des verrous
- Mettre un timeout sur l'acquisition des verrous et réessayer après un temps d'attente aléatoire



Mutex

- ▶ Problème: Accès aux données partagées
 - ▶ Notamment dans le cas de la cohérence des caches
 - ▶ Très difficile à détecter et à debugger
 - ▶ Il faut s'en prémunir de manière active
 - ▶ Par un verrou (mutex) - Son acquisition est atomique
 - ▶ Attention toutefois à relâcher le verrou rapidement pour ne pas bloquer les autres threads
 - ▶ On utilise un objet de type Lock, défini dans le paquetage `java.util.concurrent.locks`
 - ▶ Reentrant Lock
 - ▶ Read/Write Lock
- ▶ L'utilisation des verrous peut générer différents problèmes
 - ▶ Deadlocks
 - ▶ Livelocks
 - ▶ Starvation

Exercice: Diner des philosophes

- ▶ Implanter le diner des philosophes
 - ▶ Chaque philosophe est représenté par un thread
 - ▶ La quantité de nourriture restante est représentée par un entier
 - ▶ Les fourchettes sont représentées par des locks
 - ▶ Pour pouvoir manger, un philosophe doit avoir 2 fourchettes
 - ▶ A chaque fois qu'il mange, il enlève une certaine quantité à la quantité restante
 - ▶ Quand il s'est servi, il repose ses fourchettes
 - ▶ Il recommence tant qu'il reste de la nourriture sur la table
- ▶ Illustrer les différents cas de deadlocks
 - ▶ Comment les éviter ?

Sémaphore

- ▶ Contrairement au mutex, il peut être utilisé par plusieurs threads en même temps
- ▶ Davantage utilisé comme un mécanisme de synchronisation
- ▶ Un compteur associé au sémaphore permet de limiter l'accès simultané à un certain nombre de threads
 - ▶ Les Job n'ayant pas pu accéder au sémaphore sont notifiés dès que le compteur est incrémenté
 - ▶ Un sémaphore binaire (compteur limité à 1) ressemble à un mutex
 - ▶ Mais le compteur peut être modifié par n'importe quel thread
 - ▶ L'acquisition d'un sémaphore par le thread est bloquante
 - ▶ Cela endort le thread jusqu'à ce que le sémaphore soit disponible

Sémaphore: Exemple

```
fun main() = runBlocking {  
    val semaphore = Semaphore(3) // Allow 3 concurrent operations  
    val jobs = List(10) { jobId ->  
        launch {  
            semaphore.acquire() // Acquire a permit  
            try {  
                performOperation(jobId)  
            } finally {  
                semaphore.release() // Release the permit  
            }  
        }  
    }  
    jobs.joinAll()  
}  
  
suspend fun performOperation(id: Int) {  
    println("Starting operation $id")  
    delay(1000) // Simulate work  
    println("Finished operation $id")  
}
```


Barrières

- ▶ Les verrous protègent l'accès concurrent aux données
 - ▶ Mais ils ne garantissent pas les séquencements critiques (race conditions)
 - ▶ L'ordre d'exécution des threads peut avoir un impact sur l'exécution globale
 - ▶ Et provoquer des bugs très difficiles à détecter et corriger
- ▶ Pour protéger les séquencements critiques, on utilise des barrières

- Il existe plusieurs 2 types de barrière:
 - Les barrières réutilisables (CyclicBarrier)
 - Les barrières à usage unique (CountDownLatch)
 - Mécanisme différent (compteur)



Barrières: Exemple

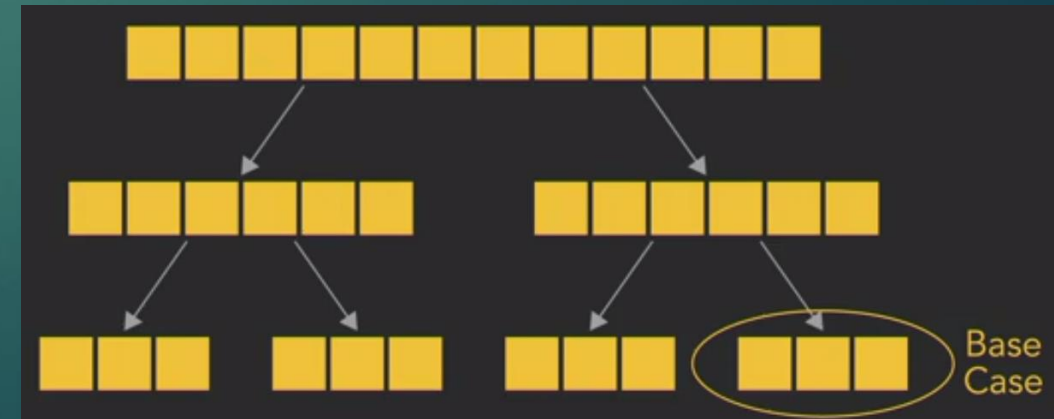
```
fun main() = runBlocking {  
    val participantCount = 5  
    val barrier = CyclicBarrier(participantCount) {  
        println("Barrier is opening!")  
    }  
  
    val jobs = List(participantCount) { id ->  
        launch(Dispatchers.Default) {  
            println("Participant $id is preparing")  
            delay((500 ≤ .. ≤ 1500).random().toLong()) // Random preparation time  
            println("Participant $id is ready and waiting at the barrier")  
            barrier.await() // Wait for all participants  
            println("Participant $id has crossed the barrier")  
        }  
    }  
  
    jobs.joinAll()  
}
```

Exercice: Barrières

- ▶ Planter deux threads
 - ▶ Un premier qui plante la fonction $f: x \rightarrow x^2$
 - ▶ Un second qui plante la fonction $g: x \rightarrow x+5$
- ▶ Instancier 5 threads de chaque
 - ▶ S'assurer qu'ils se coordonnent afin qu'ils se combinent et produisent toujours le même résultat: $x^2 + 5$.
 - ▶ Utiliser pour cela une barrière cyclique

ThreadPool: Divide & Conquer

- ▶ Java fournit des API pour exécuter des algorithmes récursifs de manière optimisée
 1. On divise un problème en sous-problèmes
 2. On résout les sous-problèmes de manière récursive
 3. On combine les différentes solutions
- ▶ Le problème est représenté sous la forme d'un arbre binaire
 - ▶ Une branche gauche et une branche droite
- ▶ Il faut instancier la classe ForkJoinPool dans ce cas
 - ▶ La méthode `fork()` pour diviser le problème
 - ▶ La méthode `join()` pour combiner les solutions
- ▶ 2 types de tâches
 - ▶ Avec retour: `RecursiveTask<V>`
 - ▶ Sans retour: `RecursiveAction`

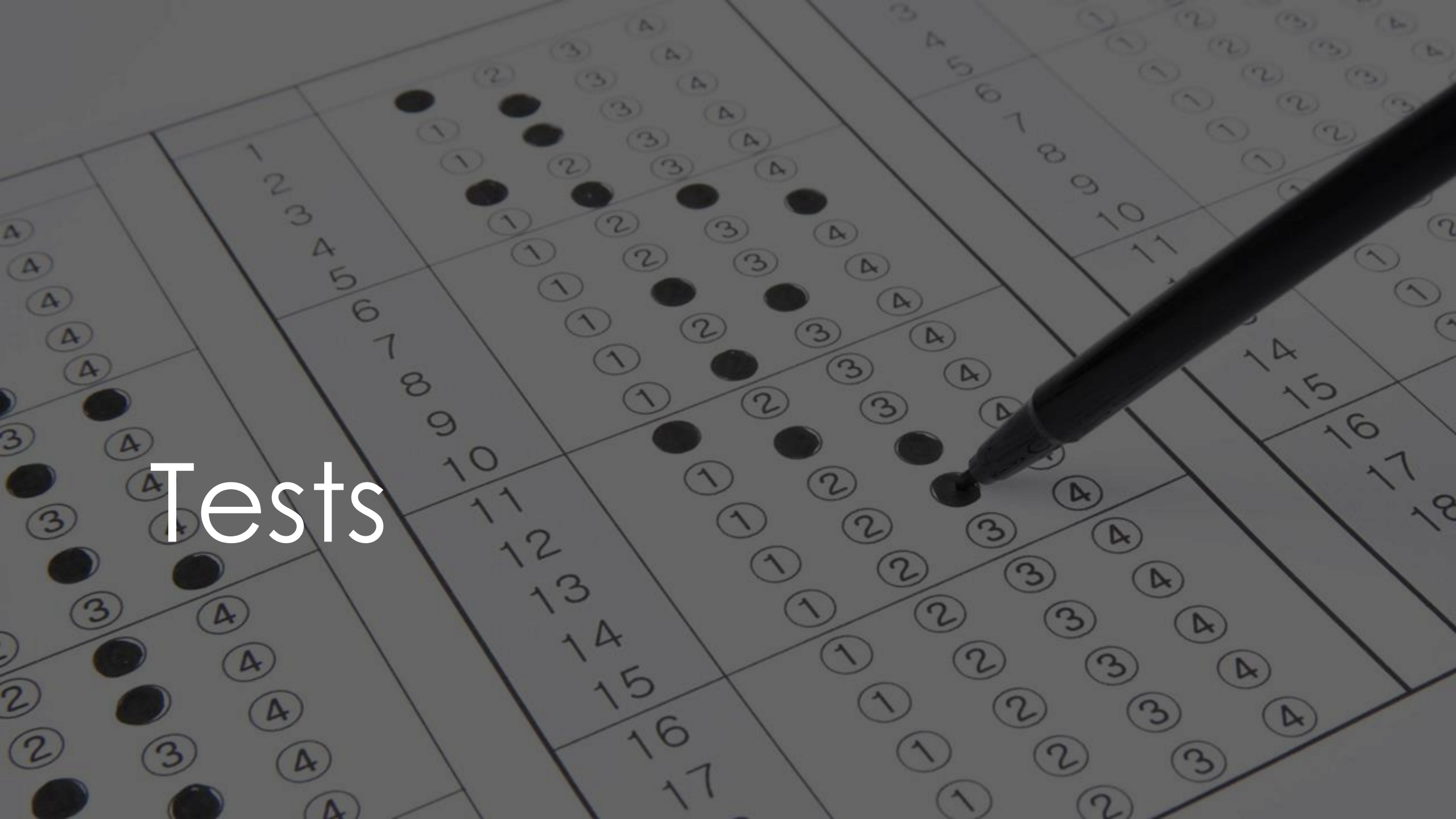


Recursive Task: Example

```
fun main() {  
    var total = 0L  
    for (i in 0..1_000_000_000) {  
        total += i  
    }  
  
    val pool = ForkJoinPool.commonPool()  
    total = pool.invoke(RecursiveSum(0, 1_000_000_000))  
    pool.shutdown()  
}
```

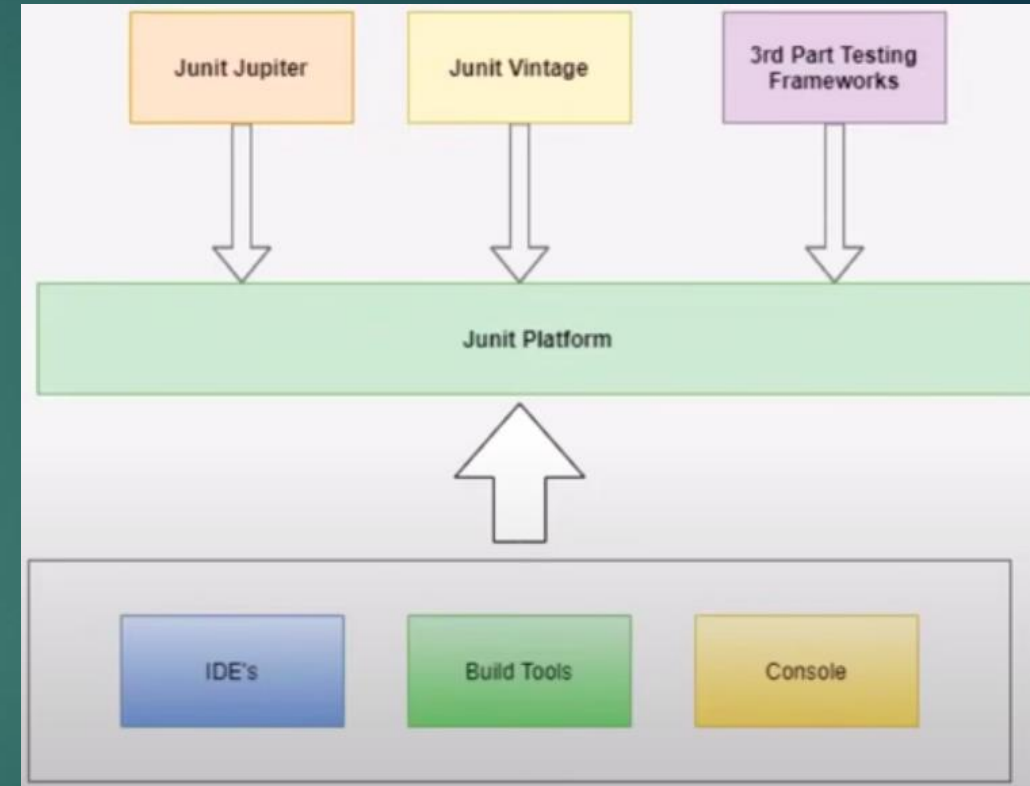
```
class RecursiveSum(private val lo: Long, private val hi: Long) : RecursiveTask<Long>()  
  
    override fun compute(): Long {  
        return if (hi - lo <= 1000) {  
            var total = 0L  
            for (i in lo..hi) {  
                total += i  
            }  
            total  
        } else {  
            val mid = (hi + lo) / 2  
            val left = RecursiveSum(lo, mid)  
            val right = RecursiveSum(mid + 1, hi)  
            left.fork()  
            right.compute() + left.join()  
        }  
    }  
}
```

Tests



TDD: JUnit

- ▶ Cadre pour spécifier et exécuter des tests reproductibles
- ▶ Permet de tester des classes ou des méthodes (SUT: Subject Under Test)
- ▶ Constitué de 4 composants:
 - ▶ Platform: Interface pour lancer des tests depuis un client
 - ▶ Jupiter: Fournit les constructions pour spécifier des tests et des extensions
 - ▶ Vintage: Moteur de test supportant une compatibilité ascendante
 - ▶ 3rd Party: Permet de créer son propre cadre de tests en réutilisant les briques de JUnit



TDD: JUnit

- ▶ Nouveauté de la version 5:
 - ▶ les tests imbriqués
 - ▶ les tests dynamiques
 - ▶ les tests paramétrés qui offrent différentes sources de données
 - ▶ un nouveau modèle d'extension
 - ▶ l'injection d'instances en paramètres des méthodes de tests
- ▶ Cycle de vie
 - ▶ **BeforeAll**: appelée à l'instanciation de chaque classe de test
 - ▶ **BeforeEach**: appelée avant l'exécution de chaque test
 - ▶ **Test**: exécution du test
 - ▶ **AfterEach**: appelée après l'exécution de chaque test
 - ▶ **AfterAll**: appelée quand la classe est déchargée

@BeforeAll

@BeforeEach

@Test

@AfterEach

@AfterAll

TDD: JUnit

[Source: <https://www.jmdoudoux.fr/java/dej/chap-junit5.htm>]

Annotation	Rôle
@Test	La méthode annotée est un cas de test. Contrairement à l'annotation @Test de JUnit, celle-ci ne possède aucun attribut
@ParameterizedTest	La méthode annotée est un cas de test paramétré
@RepeatedTest	La méthode annotée est un cas de test répété
@TestFactory	La méthode annotée est une fabrique pour des tests dynamiques
@TestInstance	Configurer le cycle de vie des instances de tests
@TestTemplate	La méthode est un modèle pour des cas de tests à exécution multiple
@DisplayName	Définir un libellé pour la classe ou la méthode de test annotée
@BeforeEach	La méthode annotée sera invoquée avant l'exécution de chaque méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @Before de JUnit 4
@AfterEach	La méthode annotée sera invoquée après l'exécution de chaque méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @After de JUnit 4
@BeforeAll	La méthode annotée sera invoquée avant l'exécution de la première méthode de la classe annotée avec @Test, @RepeatedTest, @ParameterizedTest ou @Testfactory. Cette annotation est équivalente à @BeforeClass de JUnit 4. La méthode annotée doit être static sauf si le cycle de vie de l'instance est per-class
@AfterAll	La méthode annotée sera invoquée après l'exécution de toutes les méthodes de la classe annotées avec @Test, @RepeatedTest, @ParameterizedTest et @Testfactory. Cette annotation est équivalente à @AfterClass de JUnit 4. La méthode annotée doit être static sauf si le cycle de vie de l'instance est per-class.
@Nested	Indiquer que la classe annotée correspond à un test imbriqué
@Tag	Définir une balise sur une classe ou une méthode qui permettra de filtrer les tests exécutés. Cette annotation est équivalente aux Categories de JUnit 4 ou aux groups de TestNG
@Disabled	Désactiver les tests de la classe ou la méthode annotée. Cette annotation est similaire à @Ignore de JUnit 4
@ExtendWith	Enregistrer une extension

TDD – JUnit Assertions

Egalité	Nullité	Exceptions
assertEquals()	assertNull()	assertThrows()
assertNotEquals()	assertNotNull()	
assertTrue()		
assertFalse()		
assertSame()		
assertNotSame()		

Et bien d'autres méthodes encore...

TDD – JUnit Example

```
class MyTests {  
  
    @Test  
    fun testMultiplication() {  
        val result = 2 * 3  
        assertEquals(6, result)  
    }  
  
    @Test  
    fun testDivision() {  
        val result = 6 / 2  
        assertEquals(3, result)  
    }  
}
```

TDD – JUnit Test de performance

```
@Test
fun checkTimeout() {
    Assertions.assertTimeout(
        Duration.ofMillis(200), () -> { return ""; }
    );
}
```

Exercice: Implanter des tests pour chacun des exercices réalisés au cours de cette formation



Conclusion

Conclusion

- ▶ Kotlin réduit considérablement le code « boilerplate » par rapport à Java, ce qui se traduit par un code plus concis et plus lisible. Des caractéristiques telles que les classes de données, les fonctions d'extension et les casts intelligents y contribuent.
- ▶ Kotlin est entièrement interopérable avec Java, ce qui permet aux développeurs d'utiliser les bibliothèques Java existantes et de migrer progressivement les projets Java existants vers Kotlin.
- ▶ Le système de types de Kotlin fait la distinction entre les types nullable et non nullable, ce qui permet d'éviter les exceptions liées aux pointeurs nuls à la compilation plutôt qu'à l'exécution.
- ▶ Kotlin offre un excellent support pour les paradigmes de programmation fonctionnelle avec des fonctionnalités telles que les expressions lambda, les fonctions d'ordre supérieur et les structures de données immuables.
- ▶ Les coroutines de Kotlin offrent un moyen puissant et efficace de gérer les opérations asynchrones et la concurrence, simplifiant ainsi le code asynchrone complexe.

Conclusion

- ▶ Kotlin prend en charge le développement multiplateforme, permettant le partage de code entre les plateformes JVM, JavaScript et Native.
- ▶ Kotlin est le langage préféré pour le développement Android, car il offre une meilleure productivité et des fonctions de sécurité spécifiquement utiles pour les applications Android.
- ▶ Kotlin dispose d'un écosystème en pleine expansion, avec un soutien accru des bibliothèques, des outils et des ressources communautaires.
- ▶ Kotlin combine la sécurité d'un langage à typage statique avec l'expressivité souvent associée aux langages dynamiques.
- ▶ Kotlin intègre de nombreuses caractéristiques des langages de programmation modernes, telles que le filtrage, les classes scellées et la délégation, ce qui améliore l'organisation et l'expressivité du code.

Pour aller plus loin

- ▶ Kotlin Multi Platform – KMP
- ▶ Kotlin pour Android
- ▶ Micro-services avec KTOR
- ▶ Applications réactives avec Reactor
- ▶ Spring Boot
- ▶ Micronaut