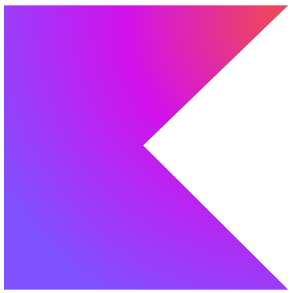


Kotlin

IUT Clermont Auvergne

Février 2024

Les coroutines en Kotlin

Petit point sur différentes notions

- Traitement séquentiel ?... concurrent ?
- Parallélisme ?... multi-threading ?
- Appel bloquant / non-bloquant ?
- Par rapport à quoi ? interruptions ? threads ? calculs ?

C'est quoi tout ce bordel ???

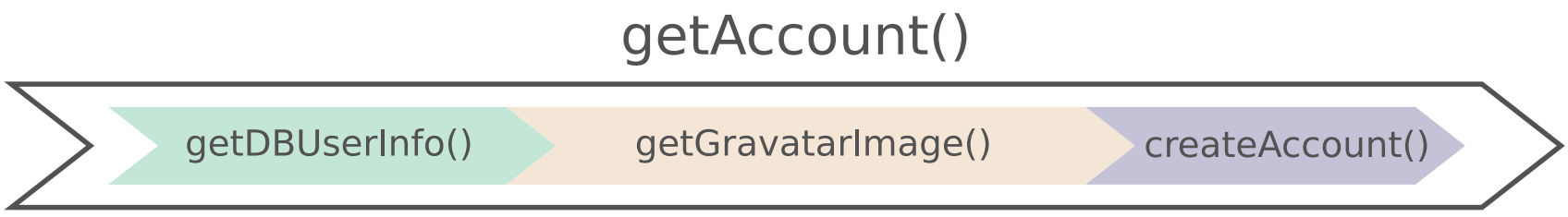
Et pourquoi ça nous intéresse ?

Séquentiel vs concurrent

- Séquentiel : on fait les opérations les unes après les autres

```
fun getAccount(id: Int) : Account {
    val userInfo = getDBUserInfo(id)
    val avatar = getGravatarImage(id)

    return createAccount(userInfo, avatar)
}
```



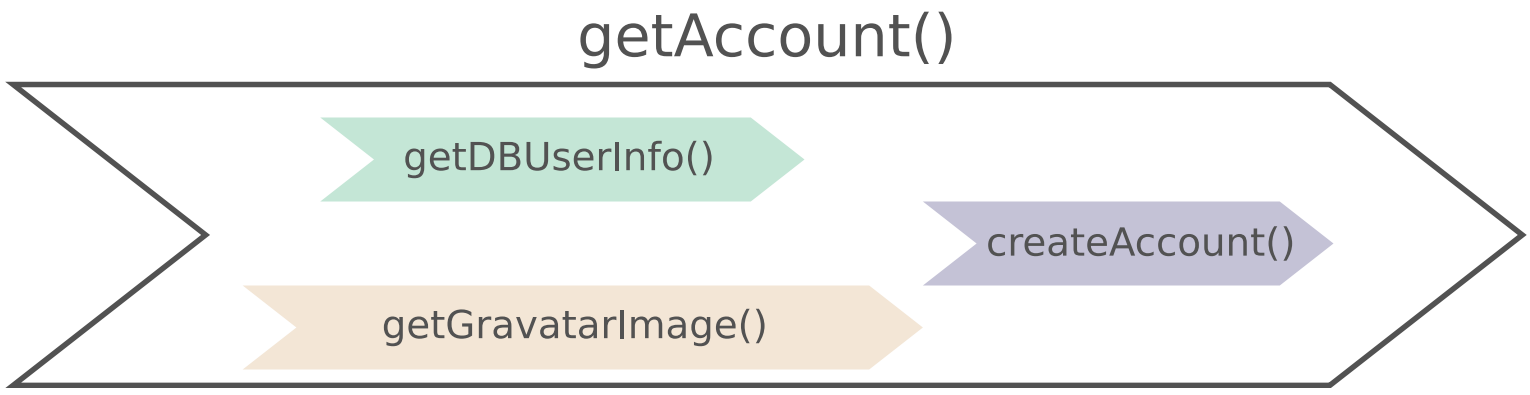
- Chaque opération doit attendre que la précédente soit terminée

Séquentiel vs concurrent

- Concurrent : variabilité dans l'ordre de certaines opérations, mais résultat déterministe

```
suspend fun getAccount(id: Int) : Account = coroutineScope {
    val userInfo = async { getDBUserInfo(id) }
    val avatar = async { getGravatarImage(id) }

    return createAccount(userInfo.await(), avatar.await())
}
```

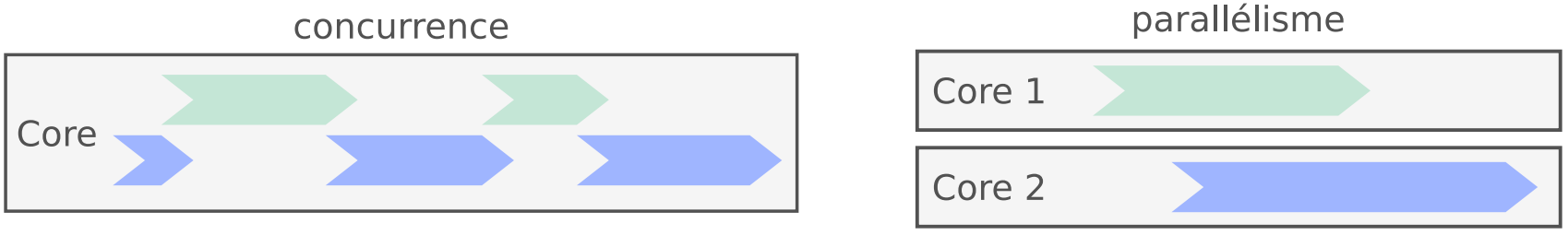


- Nécessite de l'indépendance entre certaines fonctions

Concurrence vs parallélisme

- Parallélisme : exécution en même temps (possiblités : plusieurs CPU, plusieurs cœurs, threads)
- Coroutines souvent comparées aux threads (maladroit)
- Thread : multi-tâche préemptif
- Coroutines : multi-tâche coopératif

- Coroutines utilisées pour mettre en place de la concurrence…
- … mais profite du parallélisme



Coroutines : définition

- Concept ancien, début des années 60 (Melvin Conway)
- Remis au goût du jour pour faire de l'asynchrone à « faible coût »

Coroutine

Exécution (calcul) qui peut être suspendue puis reprise sans bloquer les threads sur lesquels elle s'exécute.

Coroutines en Kotlin

- De base dans le langage Kotlin et sa lib standard :
  - ▷ mot-clé `suspend`
  - ▷ quelques types de base, i.e. `CoroutineContext`, `Continuation`
  - ▷ quelques fonctions, i.e. `createCoroutine()`, `startCoroutine()`, `suspendCoroutine()`, `resume()`

- Pour tout le reste : intégration à travers une bibliothèque (`kotlinx.coroutines`)
  https://github.com/Kotlin/kotlinx.coroutines

Problèmes classiques quand on fait de la concurrence

- Structuration du code difficile
- Fameux *callback hell*

```
const verifyUser = function(username, password, callback){
   dataBase.verifyUser(username, password, (error, userInfo) => {
       if (error) {
           callback(error)
       }else{
           dataBase.getRoles(username, (error, roles) => {
               if (error){
                   callback(error)
               }else {
                   dataBase.logAccess(username, (error) => {
                       if (error){
                           callback(error);
                       }else{
                           callback(null, userInfo, roles);
                       }
                   })
               }
           })
       }
   })
};
```

- Nouvelle API à appréhender (Fluent API des Promise/Future)
- Que avec threads : couteux

Démo : threads vs coroutines

```kotlin
fun main() = launchThreads(1_000_000)  // Compliqué après quelques milliers de threads

fun launchThreads(amount: Int) {
    val jobs = mutableListOf<Thread>()
    repeat(amount) {
        jobs += thread {
            println("Lancement du thread $it : ${Thread.currentThread().name}")
            Thread.sleep(10_000)
        }
        println("Thread $it démarré depuis ${Thread.currentThread().name}")
    }
    jobs.forEach(Thread::join)
    println("Travail terminé !")
}

suspend fun main() = launchCoroutines(1_000_000)  // Aucun soucis, coroutine plus léger

// Attention, juste une démo, ne pas lancer les coroutines comme ça dans un code en production
suspend fun launchCoroutines(amount: Int) {
    val jobs = mutableListOf<Job>()
    repeat(amount) {
        jobs += GlobalScope.launch {
            println("Lancement de la coroutine $it : ${Thread.currentThread().name}")
            delay(10_000)
        }
        println("Coroutine $it démarré depuis ${Thread.currentThread().name}")
    }
    jobs.joinAll()
    println("Travail terminé !")
}
```

Apport des coroutines

- Structuration de code classique
- Utilisation des threads mais sans les bloquer
- Amélioration des performances

- Les coroutines (et la bibliothèque Kotlin associée) sont utiles dans plusieurs cas principaux sous Android :

1. Les tâches longues qui peuvent notamment bloquer le main thread
2. Préserver la « *main-safety* » (s'assurer que les fonctions `suspend` puissent être appelées depuis le main thread)
3. Assurer la bonne gestion (annulation, échec, ...) des tâches longues en fonction du cycle de vie des composants

Réfléchir à l'utilisation

1. Algo utilisant beaucoup le CPU (*CPU bound*)
   - ▷ Profite du parallélisme
   - ▷ Ne profite pas vraiment de la concurrence en mono thread
   - ▷ Peut même en souffrir à cause des changements de contexte

2. Algo faisant beaucoup d'entrées/sorties (*IO bound*)
   - ▷ Profite beaucoup du parallélisme et de la concurrence (si E/S indépendantes)
   - ▷ Sera quasiment toujours mieux que du séquentiel

- On essaye de tirer partie des deux mondes
- Coroutines utilisent judicieusement les threads

Utilisation des coroutines en Kotlin

1 Création

- Utilisation de *coroutine builders*
- Les plus courants :
  - `launch()` : crée une coroutine qui ne renvoie pas de résultat (*fire-and-forget*); retourne le `Job` représentant la coroutine
  - `async()` : crée une coroutine permettant d'obtenir un résultat de type `T`; retourne une `Deferred<T>` qui est un job et une *future* qui encapsule le résultat
  - `runBlocking()` : crée une coroutine et bloque le thread courant tant qu'elle n'est pas terminée; utilisée pour faire le lien entre du code bloquant et du code non bloquant (entre le `main` et les *suspending functions* par exemple)

Utilisation des coroutines en Kotlin

2 Utilisation

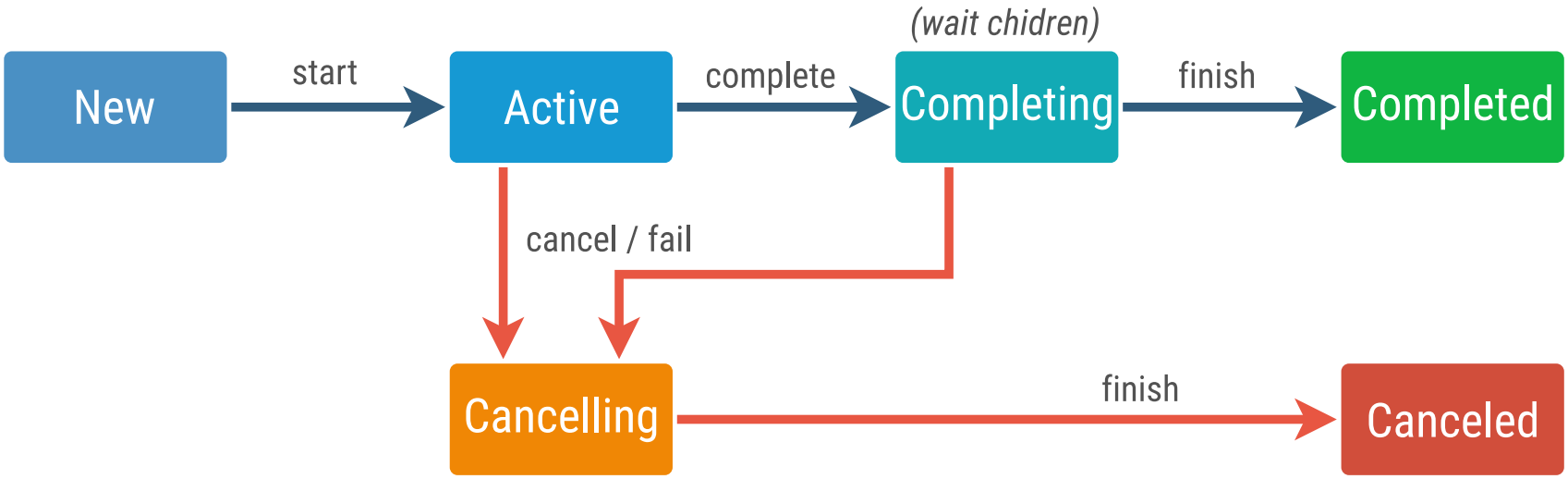
- Appel de fonctions `suspend` : travail long, qui peut être interrompu (suspendu), mais non bloquant pour le thread
- Fonctions `suspend` appelables uniquement dans coroutine ou autre fonction `suspend`
- Attente de terminaison d'une coroutine `join()`
- Attente d'un résultat `await()`, `awaitAll()`
- Annulation d'un coroutine `cancel()`, `cancelAndJoin()`
- Lancement d'autres coroutines ou modification du contexte `withContext()`, `coroutineScope()`, ...
- Laisser poliment la main à d'autres coroutines `yield`

Exemple de création de coroutines

```
suspend fun getName(): String {
    delay(1000)
    return "John"
}
suspend fun getLastName(): String {
    delay(1000)
    return "Doe"
}
fun main() {
    lateinit var job: Job
    val time = measureTimeMillis {
        job = GlobalScope.launch {
            val name = async { getName() }
            val lastName = getLastName()
            println("Hello, ${name.await()} ${lastName}")
        }
    }
    println("Execution took $time ms")
    runBlocking {
       println("Join took " + measureTimeMillis { job.join() } + " ms")
    }
}
```

Le Job

- Coroutines représentées à travers le concept de `Job`
- Si résultat : `Deferred<T>`, hérite de `Job`
- Gère le cycle de vie, l'annulation et les relations de parenté des coroutines



- Permet de mettre en place le mécanisme de *structured concurrency*

Structured concurrency

- Lien de parenté entre les coroutines
- Coroutine C2 lancée dans une coroutine C1 : `Job` de C2 est fils du `Job` de C1
- De manière générale :
  - ▷ Coroutine mère annulée => ensemble de sa hiérarchie de coroutines filles annulée aussi
  - ▷ Coroutine fille annulée => n'annule pas sa coroutine mère
  - ▷ Coroutine fille en échec => annule sa coroutine mère

- Garant de la bonne gestion des coroutines
- Indispensable sous Android pour prendre en compte de cycle de vie (être *lifecycle aware*)

- Peut-on lancer des coroutines de puis n'importe où ?

CoroutineScope

- Interface ne contenant qu'un contexte (`CoroutineContext`)
- Permet de définir des portées de code gérant l'existence de coroutines
- Appui la *structured concurrency* (un `Job` est un `CoroutineContext`)
- Les builders (hormis `runBlocking()`) sont des fonctions d'extension de `CoroutineScope`

```
fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    start: CoroutineStart = CoroutineStart.DEFAULT,
    block: suspend CoroutineScope.() -> Unit
): Job
```

CoroutineContext

- Une map d'éléments
- Les éléments permettent de customiser en partie la coroutine
- `CoroutineName`, `CoroutineDispatcher`, `CoroutineExceptionHandler`
- Modifiable en les mixant grâce à l'opérateur `+` (les éléments de même clé sont remplacés)

```
launch(Dispatcher.IO +
       CoroutineName("DBAcces") +
       CoroutineExceptionHandler(…) ) {
  …
}
```

Exemple sous Android

```
class MyActivity : AppCompatActivity(), CoroutineScope {
  private lateinit var job: Job
  override val coroutineContext: CoroutineContext
    get() = Dispatchers.Main + job

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    job = Job()
  }

  override fun onDestroy() {
    super.onDestroy()
    job.cancel()
  }

  fun loadDataFromUI() = launch {
    val ioData = async(Dispatchers.IO) { /* opération d'E/S */ }
    // ici des choses se font en concurrence avec ioData
    val data = ioData.await() // attente du résultat d'E/S
    updateUI(data) // mise à jour de la vue dans le main thread
  }
}
```

CoroutineDispatcher

- Permet de définir sur quels threads s'exécutent les coroutines
- Plusieurs implémentations existent de base :
  - `Dispatchers.Default` : celui utilisé par défaut quand rien n'est spécifié. C'est un pool de threads partagés. Approprié pour les coroutines qui consomment du CPU (parser du Json, trier une liste, ...)
  - `Dispatchers.IO` : utilise un pool partagé de thread créés à la demande. Parfait pour les opérations d'E/S bloquantes (lecture fichier, accès BdD, requête réseau, ...)
  - `Dispatchers.Unconfined` : coroutine démarrée dans le thread qui lance la coroutine, puis si coroutine suspendue, redémarrée dans le thread courant (pas forcément le même). *Ne devrait pas être utilisé normalement dans votre code*

CoroutineDispatcher

- Pour les frameworks ayant un main thread (ou UI thread) :
  - `Dispatchers.Main` (JavaFx, Android, ...) : utilisé pour des opérations légères (appeler des fonctions `suspend` main-safe, mettre à jour l'UI, ...)
- Possibilité de créer ses propres dispatchers : `newSingleThreadContext`, `newFixedThreadPoolContext`
  - alloue des ressources, donc attention à bien libérer la ressource
  - conseil : utiliser `use`
- Possibilité de convertir un `Executor` en dispatcher avec `asCoroutineDispatcher`

Modifications de contexte

- On peut modifier le contexte d'une coroutine sans lancer une autre coroutine : `withContext(…)`
- On peut créer un scope qui utilise le contexte d'une fonction appelante : `coroutineScope()` (*parallel decomposition*)

```
suspend fun showSomeData() = coroutineScope {
  val data = async(Dispatchers.IO) {
    // on charge des données en arrière plan
  }

  withContext(Dispatchers.Main) {
    // on met à jour la vue sur le thread d'affichage
    doSomeWork()
    val result = data.await()
      display(result)
  }
}
```

Coroutines en interne (pour les plus curieux)

- Fonctionnent sur le principe de *Continuation Passing Style*
- Utilisation d'un type spécial : `Continuation`
- Permet de stocker les informations pour continuer une fonction `suspend`
- Une machine à état pour :
  - ▷ Découper la fonction `suspend` suivant certains points de césure (appels `suspend`)
  - ▷ Sauvegarder les infos vitales (valeur de retour, paramètres, ...)

```
suspend fun createPost(token: Token, item: Item): Post { … }
```

devient pour la JVM

```
Object createPost(Token token, Item item, Continuation<Post> cont) { … }
```

Pour plus d'infos détaillés : https://resources.jetbrains.com/storage/products/kotlinconf2017/slides/2017+KotlinConf+-+Deep+dive+into+Coroutines+on+JVM.pdf

Scopes sous Android (pour plus tard)

- On pourrait utiliser la méthode décrite plus haut en exemple de `CoroutineScope`
- Mais il faudrait écrire le même code souvent pour gérer ces `CoroutineScope`
- Les fonctions d'extension Kotlin relatives au cycle de vie évitent ce boilerplate code
- Elles exposent des scopes pour des composants classiques (`lifecycleScope`, `viewModelScope`, `liveData`, ...)
- Il faudra penser à mettre les dépendance KTX adéquates dans le `build.gradle` de votre module

Petit retour sur Retrofit

- Retrofit est compatible avec les coroutines \😍/
- Permet une déclaration et une utilisation plus naturelle des requêtes (cf. TP CodeFirstExplorer)
- Il suffit de déclarer les fonctions de l'interface `suspend`
- On peut enlevé l'utilisation des `Call` et retourner simplement la valeur

```
interface ReqResService {
  @GET("users")
  suspend fun getUsers(): UsersResponse

  // Au lieu de
  // @GET("users")
  // fun getUsers(): Call<UsersResponse>
}
```

Communication entre coroutines

- Quelquefois nécessaire de communiquer entre coroutines
- On peut passer par des variables (mémoire) partagées
- Attention aux erreurs de concurrence dues à des coroutines s'exécutant sur différents threads
- On préférera souvent échanger des données
- C'est le rôle de `Channel`
- Equivalent aux `BlockingQueue` Java, mais dans une version suspensible

Channel

- On peut envoyer dans et recevoir d'un `Channel`

```
interface SendChannel<in E> {
  suspend fun send(element: E)
  fun close(): Boolean
}

interface ReceiveChannel<out E> {
  suspend fun receive(): E
}

interface Channel<E> : SendChannel<E>, ReceiveChannel<E>
```

- Les coroutines qui les appellent peuvent être suspendues (car `send()` et `receive()` sont `suspend`)
- Suspension dépend de l'implémentation du `Channel`

Channel

```
val rendezVousChannel = Channel<String>()
```

- Canal sans buffer : `send()` suspendu tant que pas de `receive()` et vice versa

```
val bufferedChannel   = Channel<String>(10)
```

- Canal avec buffer de taille fixe : `send()` suspendu une fois le buffer plein

```
val unlimitedChannel  = Channel<String>(UNLIMITED)
```

- Canal sans limite : `send()` n'est jamais suspendu, `OutOfMemoryException` si plus de mémoire

```
val conflatedChannel  = Channel<String>(CONFLATED)
```

- Canal d'au plus un élément : `send()` n'est jamais suspendu, l'élément est remplacé par le plus récent `send()`

Channel rendez-vous : exemple

```
fun log(msg: Any?) = println("${Thread.currentThread()} : $msg")

fun main() = runBlocking<Unit> {
    val channel = Channel<String>()
    launch {
        channel.send("A1")
        channel.send("A2")
        log("A done")
    }
    launch {
        channel.send("B1")
        log("B done")
    }
    launch {
        repeat(3) {
            val x = channel.receive()
            log(x)
        }
    }
}
```

Kotlin flows

- Pas forcément besoin de parallélisme
- Pas forcément besoin de communication entre coroutines
- On veut juste consommer des données qui arriveraient « au fur et à mesure »

```
fun main() = runBlocking {
  val data = (1..10).asSequence().onEach { Thread.sleep(500) }
  println("Start…")
  launch { data.forEach(::log) }
  repeat(10) {
    log("Doing another stuff")
    delay(200)
  }
}
```

- Problème : on bloque le thread pendant qu'on consomme la séquence

Kotlin flows

- On voudrait profiter des fonctions `suspend` qui sont non bloquantes
- Pour cela Kotlin propose le `Flow`
- Comme une séquence, mais accepte l'utilisation de fonctions `suspend`

```
fun main() = runBlocking {
    val data = (1..10).asFlow().onEach { delay(500) }
    println("Start…")
    launch { data.collect(::log) }
    repeat(10) {
        log("Doing another stuff")
        delay(200)
    }
}
```

Kotlin flows

- On peut interpréter un flow de différentes manières :

1. Un flux de données qui arrivent au fur et à mesure
2. Une fonction `suspend` qui peut retourner plusieurs valeurs
3. Une sorte d'observable qui met à jour une certaine donnée

- Un flow est séquentiel !
- Juste, il permet de ne pas bloquer les threads grâce aux fonctions `suspend`

Construction d'un flow

- Plusieurs `builders` à disposition :
- `flowOf(…)` : crée un flow qui produit les valeurs passées en paramètre

```
val mots = flowOf("Une", "phrase", "décomposée", "en", "mots")
```

- `asFlow()` : crée un flow qui produit les valeurs de la collection/séquence/intervalle/fonction sur laquelle on l'appelle

```
val chiffres = (1..10).asFlow()
```

Construction d'un flow

- `flow { … }` : crée un flow qui produit les valeurs émises par la lambda

```
val fibonnacci = flow {
    var previous = 0.toBigInteger()
    var current = 1.toBigInteger()
    while (true) {
        delay(500)
        emit(previous)
        val next = current + previous
        previous = current
        current = next
    }
}
```

- `emit(…)` permet de dire au flow de produire une valeur

Utilisation d'un flow

- On peut ensuite utiliser le flow

1. En appliquant des traitements intermédiaires (comme sur les séquences) : `onEach()`, `transform()`, `combine()`, `map()`, `distinctUntilChanged()`, `filter()`, etc. (cf. documentation pour une liste exhaustive)

```
val mots = flowOf("Une","phrase","décomposée","en","mots")
val motsLongsEnMajuscule = mots.filter { it.length > 3 }
                               .map(String::uppercase)
```

- Retourne un flow, mais n'exécute aucune opération (*cold flow*)

Consommation d'un flow

- On peut ensuite utiliser le flow
2. En appliquant une opération terminale : `collect()`, `count()`, `last()`, `emitAll()`, `fold()`, etc.

```
println(motsLongsEnMajuscule.count())        // Affichera 3
motsLongsEnMajuscule.collectIndexed { pos, mot ->
    delay(500)
    println("$mot est en position $pos dans le flow")
}
```

- Le flow est évalué seulement lorsqu'une opération terminale est appelée
- Le flow est réévalué à chaque consommation

Ordonnancement d'un flow

- On peut demander à ce que différentes opérations du flow soient exécutées sur différents *dispatchers*

```
fun main() = runBlocking<Unit> {
    val chiffres = flowOf(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
    launch {
        chiffres.filter { it % 2 == 0 }
                .onEach { log("Attente…"); delay(500) }
                .map { it * it }
                .flowOn(Dispatchers.IO)
                .collect { log("Valeur collectée : $it") }
    }
}
```

- L'ordonnancement demandé par `flowOn` s'applique pour les opérations antérieures à son appel !
- On peut le spécifier plusieurs fois pour différents ordonnancements

Cold flow, hot flow

- Un flow est froid (*cold*) par nature :
  - Il ne « mémorise » pas de données
  - Rien n'est évalué tant qu'une opération terminale n'est pas appelée
  - Un consommateur du flow reçoit toutes les valeurs émises

- Il est aussi possible de faire des flows chauds (*hot*) : `SharedFlow` et `StateFlow`
  - Le flow produit des données même s'il n'y a pas de consommateur
  - Beaucoup utilisés dans Android
  - Souvent une alternative plus simple aux channels (qui sont *hot*) si on ne veut pas communiquer entre coroutines

Hot flows

1. `SharedFlow`
   - un flow qui peut avoir plusieurs collecteurs abonnés
   - tous recevront les valeurs émises (une sorte de *broadcast*)
   - on peut obtenir un `SharedFlow` depuis un flow froid grâce à `shareIn(…)`
2. `StateFlow` (implémente `SharedFlow`)
   - un flow qui représente un seul état
   - une mise à jour de l'état émet la nouvelle valeur de l'état aux collecteurs
   - on peut obtenir la valeur de l'état grâce à la propriété `value`
   - possède toujours une valeur initiale
   - agit comme un pattern *Observer*
   - on peut obtenir un `StateFlow` depuis un flow froid grâce à `stateIn(…)`

- Leur utilisation sera vue dans la partie Android de la ressource