DAA - Développement d'applications Android

Threads & Coroutines

01 January 2026

Table des matières

1 Threads

Les threads permettent à une application d'exécuter plusieurs tâches simultanément. Chaque thread représente un flux d'exécution distinct, ce qui permet à une application de rester réactive tout en effectuant des opérations en arrière-plan.

Dans Android, le thread principal (ou UI thread) est responsable de la gestion de l'interface utilisateur. Il est crucial de ne pas bloquer ce thread avec des opérations longues, car cela entraînerait une interface utilisateur non réactive. C'est pourquoi une exception est levée si une **opération réseau** est effectuée sur le thread principal (UI-Thread).

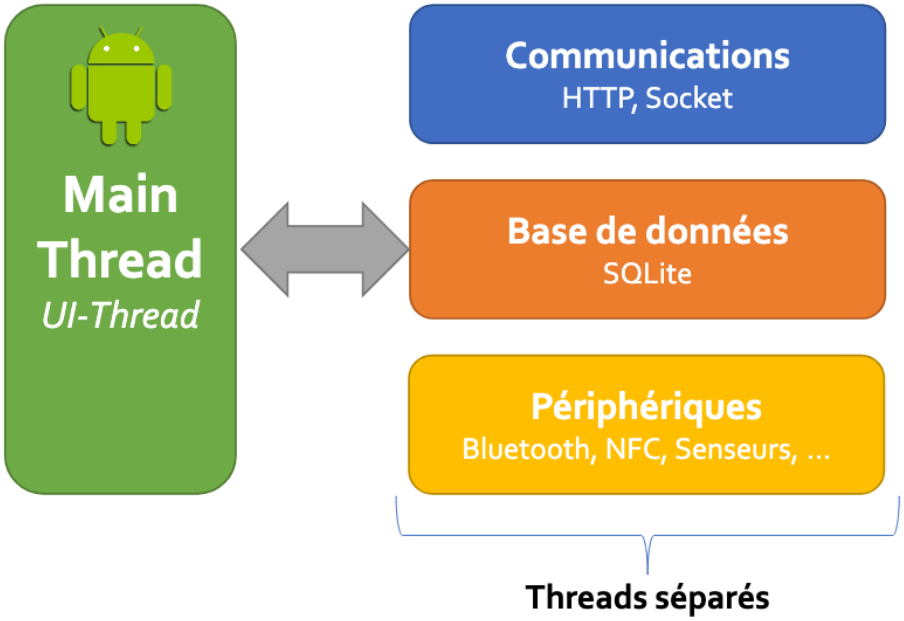

Fig. 1. – Capture des slides du cours – Séparation des threads dans une application Android

1.1 Handler

Un **Handler** est un composant qui permet de gérer la communication entre différents threads. Il est souvent utilisé pour poster des tâches à exécuter sur le thread principal depuis un thread en arrière-plan.

Voici un exemple de code qui illustre l'utilisation d'un thread pour télécharger une image depuis une URL et mettre à jour une ImageView sur le thread principal en utilisant un Handler :

```
class MyDownloader(private val url : URL,
        private val myImage : ImageView,
        private val handler : Handler) : Thread() {

  override fun run() {
    val inputStream = url.openConnection().getInputStream()
    val bmp = BitmapFactory.decodeStream(inputStream)
    inputStream.close()

    handler.post(object : Runnable {
      override fun run() {
        myImage.setImageBitmap(bmp)
      }
    })
  }
}
```

Depuis le thread principal, on peut démarrer le téléchargement de l'image comme suit :

```
val handler = Handler(Looper.getMainLooper())
val downloader = MyDownloader(URL("https://example.com/image.png"), myImageView, handler)
downloader.start()
```

1.1.1 Sucre syntaxique avec Kotlin

Kotlin offre une syntaxe plus concise pour poster des tâches sur un Handler en utilisant des lambdas. Voici comment cela peut être fait :

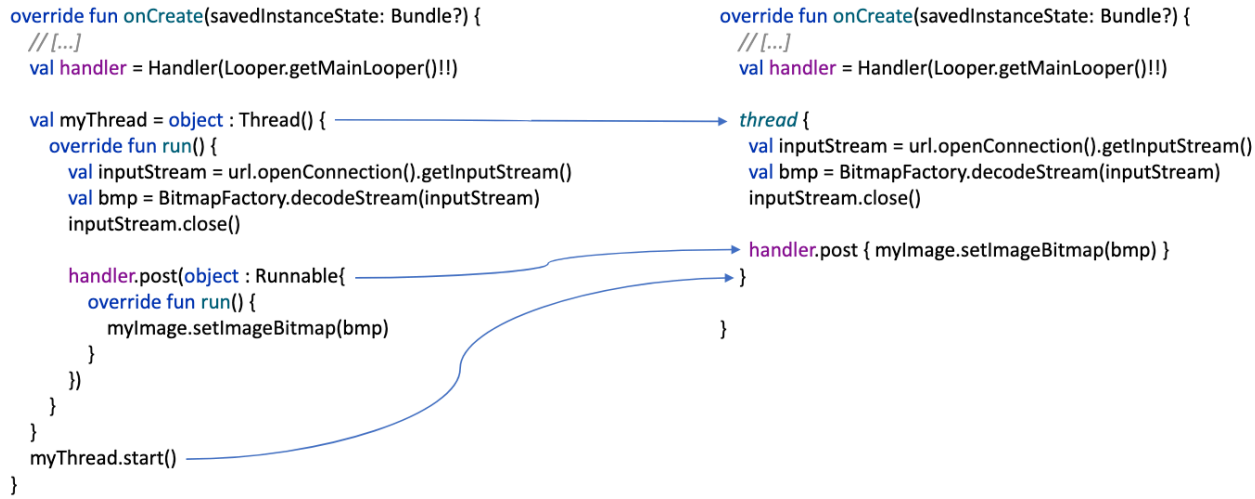

Fig. 2. – Capture des slides du cours – Utilisation de lambdas avec Handler

1.1.2 Handler dans une activité

La classe Activity du SDK possède une méthode utilitaire `runOnUiThread` qui permet d'exécuter du code sur le thread principal sans avoir à créer explicitement un Handler. Donc, dans une activité, il n'est pas nécessaire de créer un Handler pour mettre à jour l'interface utilisateur depuis un thread en arrière-plan. Voici un exemple d'utilisation de `runOnUiThread` :

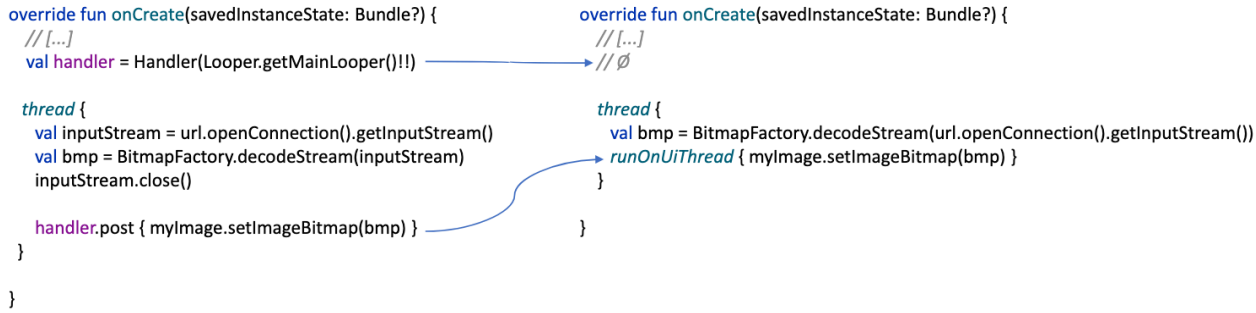

Fig. 3. – Capture des slides du cours – Utilisation de runOnUiThread dans une activité

1.2 Limites des threads

L'utilisation de Threads est bien adaptée pour des tâches courtes et ponctuelles. Cependant, il existe des cas dans lesquels les threads ne sont pas idéaux :

- Les threads ne sont pas conscients du cycle de vie des composants Android.
  - On ne souhaite pas qu'une tâche en arrière-plan continue à s'exécuter si l'activité ou le fragment associé est détruit.
  - Si un thread possède une référence vers une Activité, celle-ci ne pourra pas être libérée par le garbage collector, ce qui peut entraîner des fuites de mémoire.
- Si on lance plusieurs threads en parallèle, la gestion de la synchronisation (concurrence) entre eux peut devenir complexe.

1.2.1 Exemples

1.2.1.1 Utilisation d'une sous-classes lambda

- Les sous-classes lambda sont liées à l'instance de l'objet dans lequel elles ont été définies.
- Par exemple dans le code suivant:

```
thread {
  val bmp = BitmapFactory.decodeStrwam(url.openConnection().getInputStream())
  runOnUiThread { myImage.setImageBitmap(bmp) }
}
```

- On va créer une instance d'une sous-classe anonyme de Thread et une instance d'une classe anonyme implémenantant Runnable, toutes deux associées à l'instance de l'Activité dans laquelle elles ont été définies.
- Les Acrtivités Android peuvent être détruites par le système (rotation de l'écran par exemple), mais les Threads vont continuer leur exécution. L'instance de l'Activité détruite ne pourra pas être garbage collectée tant que toutes les instances des classes et sous-classes anonymes continuent d'exister. Donc, nous avons une **fuite mémoire (memory leak)**.

Pour éviter ce problème, on peut utiliser des références faibles (WeakReference) pour référencer l'instance de l'Activité dans les sous-classes anonymes.

```
class MyImageDownloader(private val url: URL,
                        private val handler: Handler) : Thread () {

    lateinit var callback: WeakReference<(bitman: Bitmap?) -> Unit>

    fun start(callBack : (bitmap : Bitmap?) -> Unit) {
      this.callback = WeakReference(callBack)
      super.start()
    }

    override fun run() {
      val bmp = try {
        BitmapFactory.decodeStream(url.openConnection().getInputStream())
    } catch (e : IOException) {
      Log.w("MyImageDownloader", "Exception while downloading image", e)
      null
    }
    handler.post { callback.get()?.let { it(bmp) } }
  }
}
```

1.3 Alternatives aux threads

Pour surmonter les limitations des threads, Android propose plusieurs alternatives modernes:

- Les Coroutines Kotlin
- La librairie WorkManager

2 Coroutines

Une **coroutine** est une unité légère d'exécution qui permet de gérer des tâches asynchrones de manière plus simple et plus efficace que les threads traditionnels. Les coroutines sont gérées par le runtime Kotlin et permettent de suspendre et de reprendre l'exécution d'une fonction sans bloquer le thread sous-jacent.

2.1 Suspending functions

Les fonctions suspendues (suspending functions) sont des fonctions spéciales qui peuvent être suspendues et reprises ultérieurement. Elles sont définies à l'aide du mot-clé `suspend` et ne peuvent être appelées que depuis une coroutine ou une autre fonction suspendue.

```
suspend fun backgroundTask(param: Int): Int {
  // long running operation
}
```

Cette approche permet d'écrire du code asynchrone de manière séquentielle, ce qui le rend plus lisible et plus facile à maintenir. Cependant, il s'agit uniquement d'un sucre syntaxique, le compilateur va convertir automatiquement ces méthodes en réintroduisant les callbacks nécessaires.

```
fun backgroundTask(param: Int, callback: Continuation<Int>): Int {
  // long running operation
}
```

> ⚠ **Warning**
>
> Il ne faut pas confondre les méthodes dites **supsensives** introduites par Kotlin avec les méthodes **bloquantes** classqiues déjà présentes en Java. Les méthodes bloquantes empêchent le thread d'exécution de continuer tant qu'elles n'ont pas terminé leur travail, tandis que les méthodes suspendues permettent de libérer le thread pendant l'attente, améliorant ainsi la réactivité de l'application.

2.2 Contextes d'exécution

Les coroutines s'exécutent dans des contextes spécifiques appelés **Dispatchers**. Un Dispatcher détermine le thread ou le pool de threads sur lequel une coroutine s'exécute.

Cela est nécessaire car juste signifier une fonction comme suspendue ne garantit pas qu'elle devienne « suspensive ».

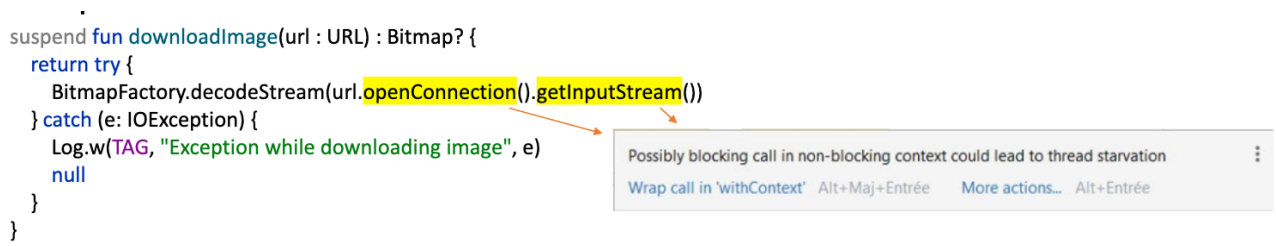

Fig. 4. – Capture des slides du cours – Potentiel blocage avec les fonctions suspendues

C'est pourquoi il est important de spécifier le contexte d'exécution approprié pour les coroutines, en utilisant des Dispatchers.

```
suspend fun downloadImage(url : URL) : Bitmap? = withContext(Dispatchers.IO) {
  try {
    BitmapFactory.decodeStream(url.openConnection().getInputStream())
  } catch (e: IOException) {
    Log.w(TAG, "Exception while downloading image", e)
    null
  }
}
```

2.2.1 Principaux Dispatchers

- `Dispatchers.Main` : Utilisé pour exécuter des coroutines sur le thread principal (UI thread). Idéal pour les opérations qui interagissent avec l'interface utilisateur.
- `Dispatchers.IO` : Conçu pour les opérations d'entrée/sortie (I/O) telles que les appels réseau ou l'accès aux fichiers. Utilise un pool de threads optimisé pour les tâches I/O. (Nbr de thread alloués dynamiquement, max = 64)
- `Dispatchers.Default` : Utilisé pour les opérations de calcul intensif qui ne nécessitent pas d'accès à l'interface utilisateur ou aux ressources I/O. Utilise un pool de threads optimisé pour les tâches CPU-intensives. (Nbr de thread = nbr de coeurs CPU)

> **i Info**
>
> Pour chaque coroutine, nous allons choisir le dispatcher le plus approprié en fonction de la nature de la tâche à accomplir.
>
> Une tâche complexe sera alors divisée en une multitude d'opérations simples, chacune de ces opérations deviendra une coroutine s'exécutant dans le contexte le plus adapté.

2.2.2 Exemple

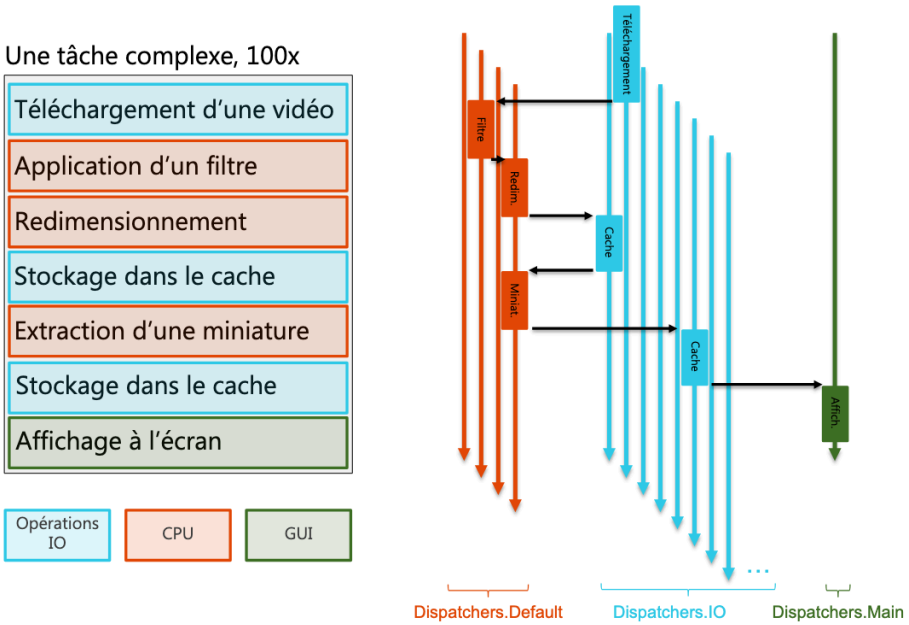

Fig. 5. – Capture des slides du cours – Utilisation des coroutines avec différents dispatchers

On va donc séparer les différentes opérations en fonction de leur nature:

```
suspend fun downloadImage(url : URL) : ByteArray? = withContext(Dispatchers.IO) {
  try {
    url.readBytes()
  } catch (e: IOException) {
    Log.w(TAG, "Exception while downloading image", e)
    null
  }
}

suspend fun decodeImage(bytes : ByteArray?) : Bitmap? = withContext(Dispatchers.Default) {
  try {
    BitmapFactory.decodeByteArray(bytes, 0, bytes?.size ?: 0)
  } catch (e: IOException) {
    Log.w(TAG, "Exception while decoding image", e)
    null
  }
}

suspend fun displayImage(bmp : Bitmap?) = withContext(Dispatchers.Main) {
  if(bmp != null)
    myImage.setImageBitmap(bmp)
  else
    myImage.setImageResource(R.drawable.error_placeholder)
}
```

Le téléchargement de l'image dans le dispatcher *IO*

Le décodage de l'image dans le dispatcher "*CPU*"

L'affichage de l'image dans l'*UI-Thread*

Fig. 6. – Capture des slides du cours – Séparation des opérations en coroutines selon leur nature

2.3 Scopes

Les différents Scopes de coroutine permettent de définir le cycle de vie des coroutines. Un scope est lié à un composant Android (activité, fragment, service) et garantit que les coroutines lancées dans ce scope sont annulées lorsque le composant est détruit.

- **`GlobalScope`** : Les coroutines sont lancées dans le scope de l'application. C'est au développeur de s'assurer de les annuler lorsqu'elles ne sont plus nécessaires, par exemple lorsqu' une Activité est détruite (sujet aux memory leaks, donc son utilisation est déconseillée)
- **`LifeCycleScope`** : Scope associé à un objet possédant un cycle de vie, par exemple une Activité ou un Fragment. Les coroutines lancées dans ce scope seront automatiquement stoppées à la fin du cycle de vie de l'objet parent
- **`ViewModelScope`** : Equivalent au précédent mais pour les ViewModels

Pour lancer la coroutine principale depuis une activité, on peut utiliser le scope `lifecycleScope`, qui est lié au cycle de vie de l'activité.

```
lifecycleScope.launch {
  val bytes = downloadImage(url)
  val bmp = decodeImage(bytes)
  displayImage(bmp)
}
```

Le contenu entre les `{}` est une coroutine lambda qui va être lancée dans le scope `lifecycleScope`. Par défaut, cette coroutine s'exécute sur le dispatcher `Dispatchers.Main`, ce qui est approprié pour les opérations qui interagissent avec l'interface utilisateur. Pour spécifier l'utilisation d'un autre Dispatcher :

```
lifecycleScope.launch(Dispatchers.Default) {
  //...
}
```

2.4 Stopper une coroutine

Les coroutines peuvent être annulées à tout moment en utilisant la méthode `cancel()`. Lorsqu'une coroutine est annulée, elle arrête son exécution et libère les ressources associées. Par exemple, si une activité est détruite, toutes les coroutines lancées dans son scope seront automatiquement annulées.

2.4.1 delay vs Thread.sleep

Pour introduire une pause dans une coroutine, on utilise la fonction `delay()`, qui est une fonction suspendue. Contrairement à `Thread.sleep()`, qui bloque le thread d'exécution, `delay()` suspend la coroutine en libérant le thread, permettant ainsi à d'autres coroutines de s'exécuter pendant ce temps.

```
suspend fun example() {
  println("Start")
  delay(1000L) // Pause de 1 seconde sans bloquer le thread
  println("End")
}
```

3 Jetpack WorkManager

Les coroutines sont idéales pour des tâches asynchrones courtes. Cependant, pour des tâches plus longues ou périodiques, Android propose la librairie WorkManager, qui permet de planifier et d'exécuter des tâches en arrière-plan de manière fiable, même si l'application est fermée ou le dispositif redémarré.

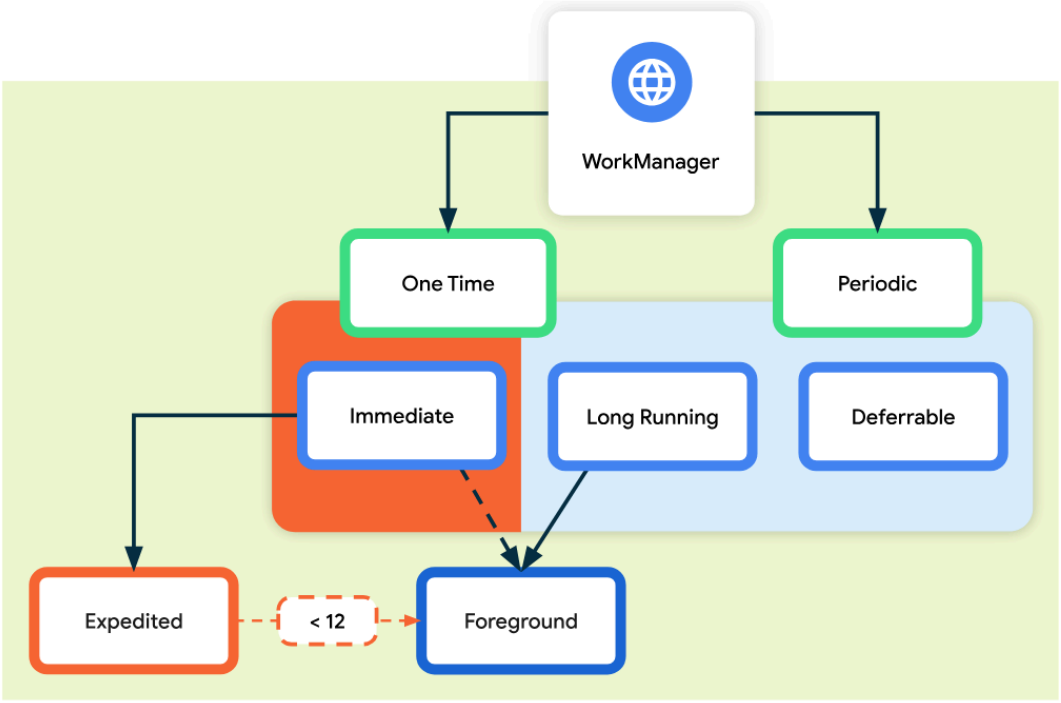

Fig. 7. – Capture des slides du cours – Architecture de WorkManager

Imaginions que nous souhaitions se connecter toutes les x minutes pour synchroniser des données avec un serveur distant. WorkManager permet de définir des contraintes pour l'exécution de ces tâches, telles que la disponibilité du réseau ou le niveau de batterie.

3.1 Tâche périodique

Lorsque le téléphone n'est pas connecté à un chargeur et que son écran est verouillé, le mode *Doze* est activé pour économiser la batterie. Dans ce mode, les tâches en arrière-plan sont différées jusqu'à ce que le téléphone sorte de ce mode. Le système sort périodiquement de sa veille profonde pour permettre aux applications de synchroniser leurs données. Cependant, la fréquence de ces fenêtres d'activité est contrôlée par le système et peut varier en fonction de l'utilisation du téléphone et de l'état de la batterie.

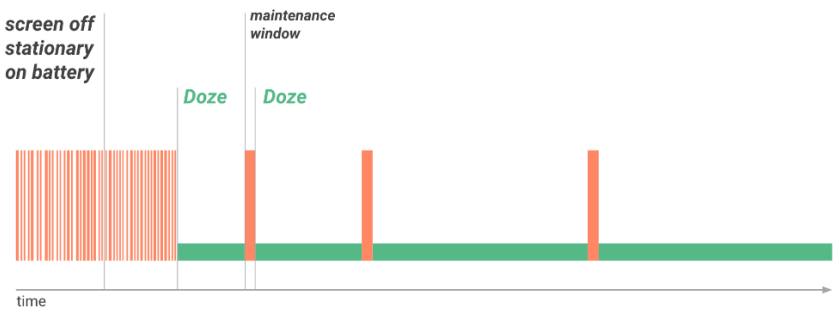

Fig. 8. – Capture des slides du cours – Tâche périodique avec WorkManager

> **i Info**
>
> Le système va classer les applications en fonction de leur utilisation, les catégories sont les suivantes:
> - Active bucket: En cours d'utilisation
> - Working set bucket: Utilisation quotidienne
> - Frequent bucket: Utilisation hebdomadaire
> - Rare bucket: Utilisation mensuelle ou moins fréquente
> - Restricted bucket: Utilisation très rare ou jamais

3.2 Utilisation de WorkManager

Pour utiliser WorkManager, il faut d'abord définir une classe qui étend `Worker` et implémente la méthode `doWork()`, qui contient le code de la tâche à exécuter en arrière-plan.

```
class MyWork(appContext: Context, workerParams: WorkerParameters) : Worker(appContext,
workerParams) {
  override fun doWork(): Result {
    // do something
    return Result.success()
  }
}
```

Ensuite, nous pouvons lancer notre tâche en utilisant `WorkManager`. Pour une tâche à lancer une seule fois :

```
val workManager = QWorjManager.getInstance(applicationContext)
val myWorkRequest = OneTimeWorkRequestBuilder<MyWork>().build()
workManager.enqueue(myWorkRequest)
```

3.2.1 Utilisation de manière périodique

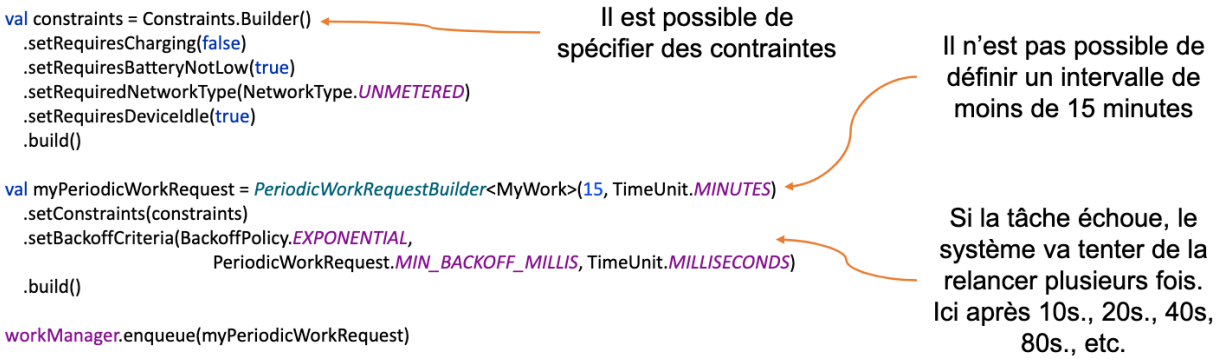

Fig. 9. – Capture des slides du cours – Utilisation de WorkManager pour une tâche répétée