# DAA-L5-AsyncTask

*Auteurs : Bugna, Slimani & Steiner*

## Réponses aux questions

### 3

#### 3.1

*Veuillez expliquer comment votre solution s'assure qu'une éventuelle Couroutine associée à une vue (item) de la RecyclerView soit correctement stoppée lorsque l'utilisateur scrolle dans la galerie et que la vue est recyclée.*

En stoquant la référance du Job on peut le `cancel` en utilisant `job?.cancel()`, dans le `onViewRecycled`, de la classe englobante appelé automatiquement, on va appeler à l'aide du holder la méthode du `ImageViewHolder` permettant de `cancel` le job.

#### 3.2

*Comment pouvons-nous nous assurer que toutes les Coroutines soient correctement stoppées lorsque l'utilisateur quitte l'Activité ? Veuillez expliquer la solution que vous avez mise en œuvre, est-ce la plus adaptée ?*

Les Coroutines sont associées au scope du cycle de vie de l'activité via lifecycleScope. Ce dernier est automatiquement annulé lorsque l'activité est détruite. Cela fonctionne car les Coroutines lancées à l'intérieur de ce scope (par exemple via launch { … }) s'exécutent dans un contexte lié à l'activité. Ainsi, elles sont automatiquement annulées lorsque le cycle de vie de l'activité prend fin, évitant tout risque de fuite de ressources.

Dans notre code, nous transmettons le lifecycleScope de l'activité à l'Adapter. Celui-ci va l'utiliser pour lancer des Coroutines de manière sécurisée, comme dans l'exemple suivant : lifecycleOwner.lifecycleScope.launch { … }

#### 3.3

*Est-ce que l'utilisation du Dispatchers.IO est la plus adaptée pour des tâches de téléchargement ? Ou faut-il plutôt utiliser un autre Dispatcher, si oui lequel ? Veuillez illustrer votre réponse en effectuant quelques tests.*

Pour les tâches intensives en I/O comme le téléchargement de fichiers ou la lecture depuis le réseau, `Dispatchers.IO` est le choix recommandé. Il est optimisé pour les opérations de blocage (telles que les requêtes réseau ou la lecture/écriture de fichiers) et gère efficacement un pool de threads dédiés à ces tâches. Néanmoins au vu de la taille des images, la différence n'est pas sensé être significative.

`Dispatchers.Default` est optimisé pour des tâches de calcul (CPU-bound) et ne gère pas bien les opérations de blocage. Cela pourrait également bloquer des threads nécessaires à d'autres calculs.

Les moyennes des mesures sont :
- Default : de 59 ms à 181 ms
- IO : de 2912 ms à 3330 ms

Ce qui est plutôt étonnant, en répétant plusieurs fois les calculs, il semble que le Default soit 20x plus rapide. On s'attendait plutôt a ne pas voir de différences, vu la taille très faible des images.

### 3.4

*Nous souhaitons que l'utilisateur puisse cliquer sur une des images de la galerie afin de pouvoir, par exemple, l'ouvrir en plein écran. Comment peut-on mettre en place cette fonctionnalité avec une RecyclerView? Comment faire en sorte que l'utilisateur obtienne un feedback visuel lui indiquant que son clic a bien été effectué, sur la bonne vue.*

Pour permettre à l'utilisateur de cliquer sur une image et d'afficher une vue en plein écran, il faut implémenter un "écouteur" de clic dans le `ImageViewHolder`.

Pour le feedback on peut par exemple afficer un toast à l'utilisateur de la manière suivante:

```
// Gestion du clique sur une image TEST
imageView.setOnClickListener {
    Log.d("Click", "Clique sur l'image : $imageUrl")
    Toast.makeText(
        itemView.context, // Utilise le contexte de la vue
        "Clique sur l'image : $imageUrl",
        Toast.LENGTH_SHORT
    ).show()
}
```

Il faut évidemment l'Activité correspondante ainsi que le(s) layout(s) nécessaire(s) (paysage, etc...).

Finalement il faut créer et lancer l'intent de l'activité de plein écran depuis le `setOnClickListener`.

Par exemple :

```
val intent = Intent(itemView.context, FullScreenActivity::class.java)
intent.putExtra("IMAGE_URL", imageUrl)
itemView.context.startActivity(intent)
```

## 4

### 4.1

*Lors du lancement de la tâche ponctuelle, comment pouvons-nous faire en sorte que la galerie soit rafraîchie ?*

On pourrait faire en sorte que lorsque le cleaner effectue sa tâche on appelle en plus un rafraichissement, comme suit :

```
override fun doWork(): Result {
    val cacheDir = applicationContext.cacheDir
    cacheDir.listFiles()?.forEach { it.delete() }
    Log.d(applicationContext.getString(R.string.log_tag_worker),
applicationContext.getString(R.string.log_cache_cleaned))

    // Envoyer un Broadcast pour notifier l'Activity ou Fragment que le cache est
vidé
    val intent = Intent("com.example.app.CACHE_CLEARED")
    applicationContext.sendBroadcast(intent)

    return Result.success()
}
```

Et écouter le broadcastReceiver (initialisé dans la mainactivity par exemple) dans l'activité afin de notifier l'adaptateur de recharger les images.

Sinon on peut simplement changer d'adapteur dansd le `reloadData()` de la `mainActivity` :

```
val recyclerView = findViewById<RecyclerView>(R.id.recyclerView)
recyclerView.adapter = null // Retirer l'adaptateur actuel
recyclerView.adapter = Adapter(this) // Réassigner un nouvel adaptateur
```

Cette solution est beaucoup plus simple mais je ne suis pas sûr que cela ne crée pas de soucis avec les coroutines, il me semble que les coroutines étant lié au lifeCycle de la `MainActivity` et non à celui de l'adapteur, techniquement elles ne sont possiblement pas terminées, risque de faire que cela ne respecte plus les réponses aux questionx du point 3. Peut-être qu'un appel au destroy des job pourrait alors être effectué avant de modifier l'adapteur.

**4.2**

*Comment pouvons-nous nous assurer que la tâche périodique ne soit pas enregistrée plusieurs fois ? Vous expliquerez comment la librairie WorkManager procède pour enregistrer les différentes tâches périodiques et en particulier comment celles-ci sont ré-enregistrées lorsque le téléphone est redémarré.*

WorkManager permet de gérer les tâches périodiques en garantissant qu'elles ne soient pas enregistrées plusieurs fois. Et cela en spécifiant un identifiant unique pour chaque tâche périodique. Notamment en utilisant `enqueueUniquePeriodicWork()` en lui fournissant un `ExistingPeriodicWorkPolicy.KEEP` afin que la tâche soit conservée et pas enregistrée à nouveau.

Le WorkManager s'occupe de gérer le tâches périodique, y compris leur persistance, en utilisant un emplacement de stockage interne. Il enregistre son état dans une DB, SQLite il me semble, de cette manière au redémarrage il est capable de retrouver son état. au redémarrage il lit cette DB et enregistre les tâches dans le planificateur du system. C'est en fait le system android qui redémarre les services nécessaires.