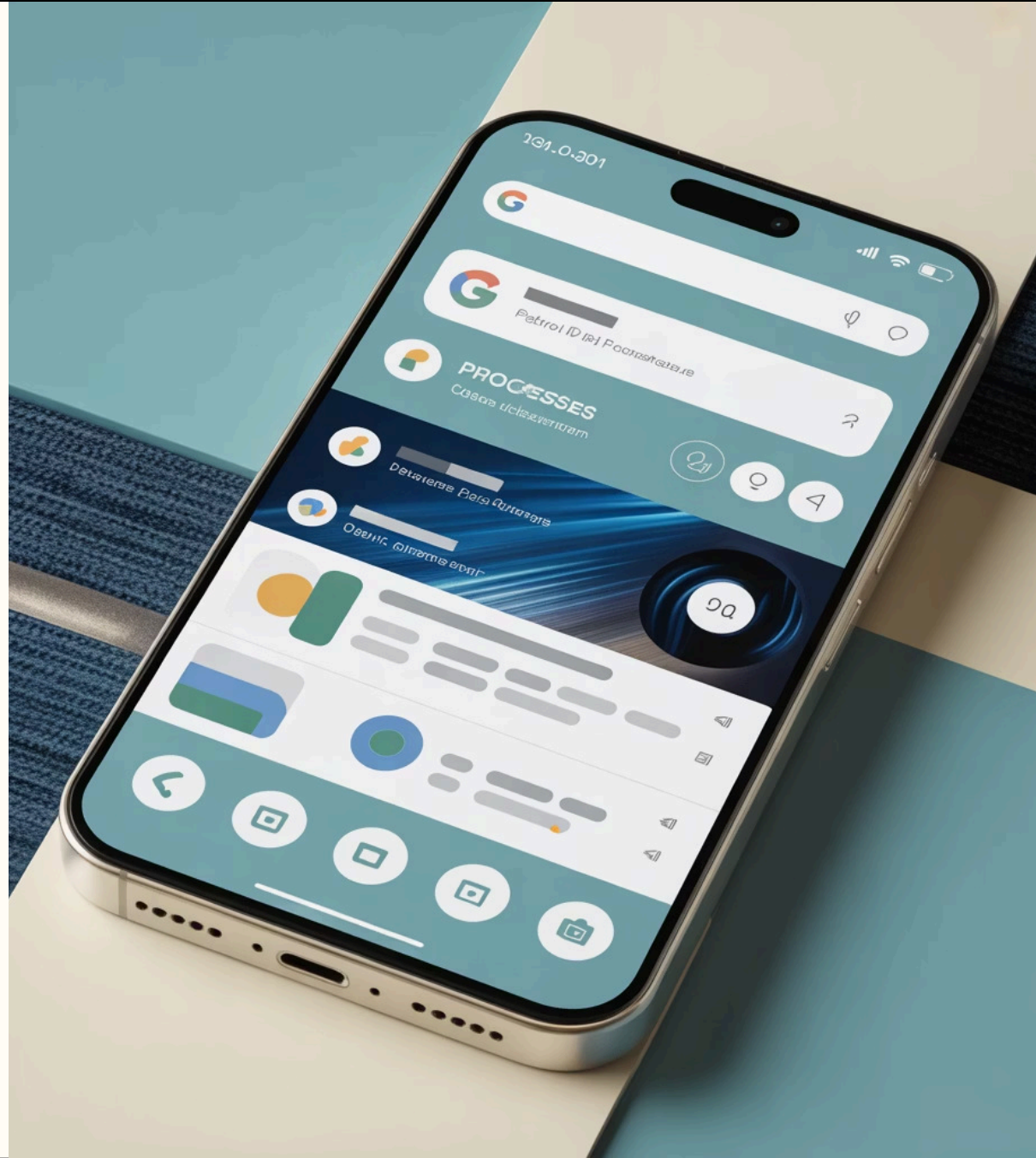
# Chapitre 8 : Threads sous Android



# Threads sous Android

Une application Android moderne repose sur plusieurs composants coopérant de manière concurrente afin d'assurer performance, réactivité et continuité des traitements. Ce chapitre présente les mécanismes de concurrence en Android, en mettant l'accent sur la gestion des processus, des threads et des composants dédiés aux tâches asynchrones.

01

## Threads et Runnables en Android

02

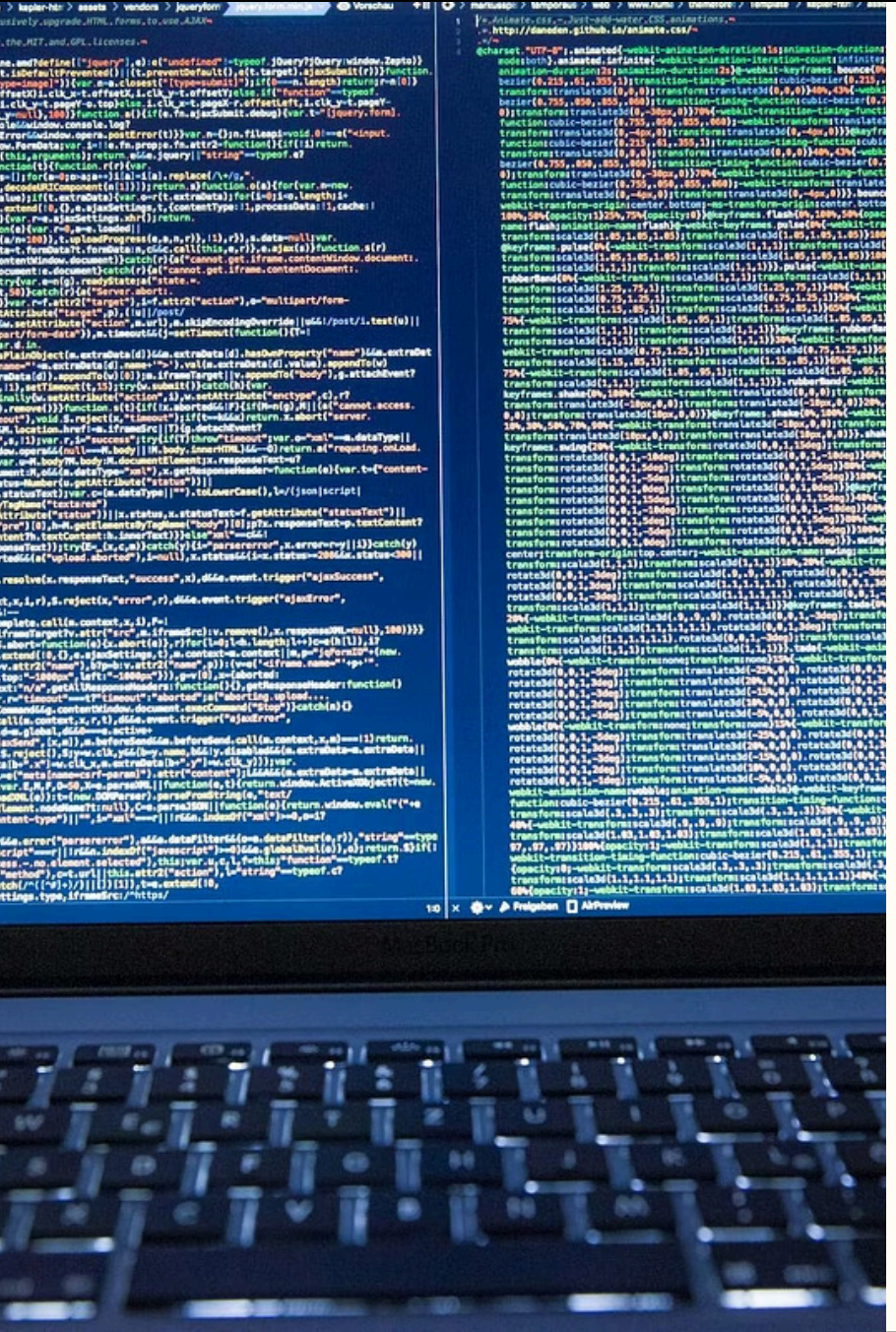
## Qestion du thread principal (UI Thread)

03

## Services Android et traitements en arrière-plan

04

## Bonnes pratiques et erreurs courantes en programmation concurrente Android



# Threads et Runnables en Android

# Threads et Runnables en Android

Android repose sur un modèle d'exécution mono-thread pour l'interface utilisateur. Toute opération longue exécutée sur le thread principal (UI Thread) provoque un gel de l'interface et peut entraîner une ANR (Application Not Responding).
Pour garantir la réactivité de l'application, Android met à disposition des mécanismes de programmation concurrente, principalement les Threads et les Runnables.

## Le thread principal (UI Thread)

Le **UI Thread** est responsable de :

- l'affichage de l'interface graphique,
- la gestion des événements utilisateur (clics, saisie),
- le dessin des vues.

> **Toute opération longue (réseau, accès disque, calcul intensif) doit être exécutée en dehors du UI Thread.**

Sinon, l'application risque :

- des ralentissements,
- une ANR après ~5 secondes de blocage.

# Threads et Runnables en Android

## Les Threads en Android

Un Thread est un flux d'exécution indépendant permettant d'exécuter du code en parallèle avec le thread principal.

En Android, les threads sont basés sur ceux que vous connaissez de Java.

### Exemple simple

```
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        // Tâche longue (réseau, calcul, lecture fichier)
        doLongTask();
    }
});
thread.start();
```

Problème : accès à l'UI depuis un thread : Un thread secondaire **ne peut pas modifier l'interface graphique**.

## Mise à jour de l'UI depuis un thread

### Solution 1 : `runOnUiThread()`

```
runOnUiThread(() -> {
    textView.setText("Task finished");
});
```

### Solution 2 : `Handler` et `Looper`

```
Handler handler = new Handler(Looper.getMainLooper());

handler.post(() -> {
    textView.setText("Updated");
});
```

Le UI Thread fonctionne avec :

- une **Message Queue**
- un **Looper**

Chaque action UI est ajoutée à la file et traitée séquentiellement, si une tâche bloque la file, toutes les autres attendent.

**Exemple typique :** Imaginez télécharger une image sur un thread secondaire. Une fois le téléchargement terminé, vous utiliseriez un Handler ou runOnUiThread() pour afficher cette image dans un ImageView sur l'écran.

# Threads et Runnables en Android

## Les Runnables

Un Runnable est une interface représentant une tâche à exécuter, indépendante du thread qui l'exécute.

```
Runnable task = new Runnable() {
    @Override
    public void run() {
        // Code à exécuter
    }
};
```

**Pourquoi utiliser Runnable ?**

- Séparation tâche / thread
- Réutilisable
- Plus flexible
- Compatible avec :
    - Thread
    - Handler
    - ExecutorService

# Threads et Runnables en Android

## Qestion des threads avec `ExecutorService`

Problème des threads manuels:

- Création excessive
- Difficulté de gestion
- Fuites mémoire possibles

**Solution : ExecutorService**

```
ExecutorService executor = Executors.newSingleThreadExecutor();

executor.execute(() -> {
    // tâche en arrière-plan
});
```

## Mise à jour de l'UI avec `ExecutorService`

Toujours via le thread principal :

```
ExecutorService executor = Executors.newSingleThreadExecutor();
Handler handler = new Handler(Looper.getMainLooper());

executor.execute(() -> {
    // tâche en arrière-plan
    handler.post(() -> {
        textView.setText("Task finished");
    });
});
```

Avantages :

- Pool de threads: évite de créer/détruire des threads à chaque tâche
- Meilleure gestion des ressources : moins de surcharge mémoire
- Recommandé pour les tâches répétitives: (requêtes réseau, accès BD, traitements)

# Threads et Runnables en Android

## Exécution concurrente avec Executor et Thread Pools

### Executor

`Executor` abstrait la gestion des threads.

```
Executor executor =
Executors.newSingleThreadExecut
or();

executor.execute(() -> {
    // tâche en arrière-plan
});
```

### Thread Pool

```
ExecutorService executor =
Executors.newFixedThreadPool(4);
```

Avantages :

- Réutilisation des threads
- Meilleure performance
- Limitation du nombre de threads

### Retour vers le UI Thread

```
Handler handler = new
Handler(Looper.getMainLooper());

executor.execute(() -> {
    String result = downloadData();
    handler.post(() ->
textView.setText(result));
});
```

# Threads et Runnables en Android

## View.post(Runnable)

```
textView.post(() -> {
    textView.setText("Hello");
});
```

### Ce que ça fait

- Le `Runnable` est ajouté à la queue du thread UI
- Le code est exécuté quand l'UI est prête
- Autorisé pour modifier l'interface

### Équivalent conceptuel

```
new Handler(Looper.getMainLooper()).post(() -> {
    textView.setText("Hello");
});
```

## View.postDelayed(Runnable, long)

```
textView.postDelayed(() -> {
    textView.setText("After 2 seconds");
}, 2000);
```

### Ce que ça fait

- Exécute le code sur le thread UI
- Après un délai (en millisecondes)

**C'est exactement le même objectif que :** `runOnUiThread()` et `Handler.post()`, **mais avec une syntaxe plus simple**

# Threads et Runnables en Android

## Cycle de vie et UI Thread

### Problème courant

Un thread continue à travailler alors que :

- l'Activity est détruite
- l'utilisateur change d'écran

Android ne l'arrête pas automatiquement

Si l'Activity est détruite (`onDestroy()`), le thread :

  - continue à s'exécuter
  - peut tenter d'accéder à une UI inexistante

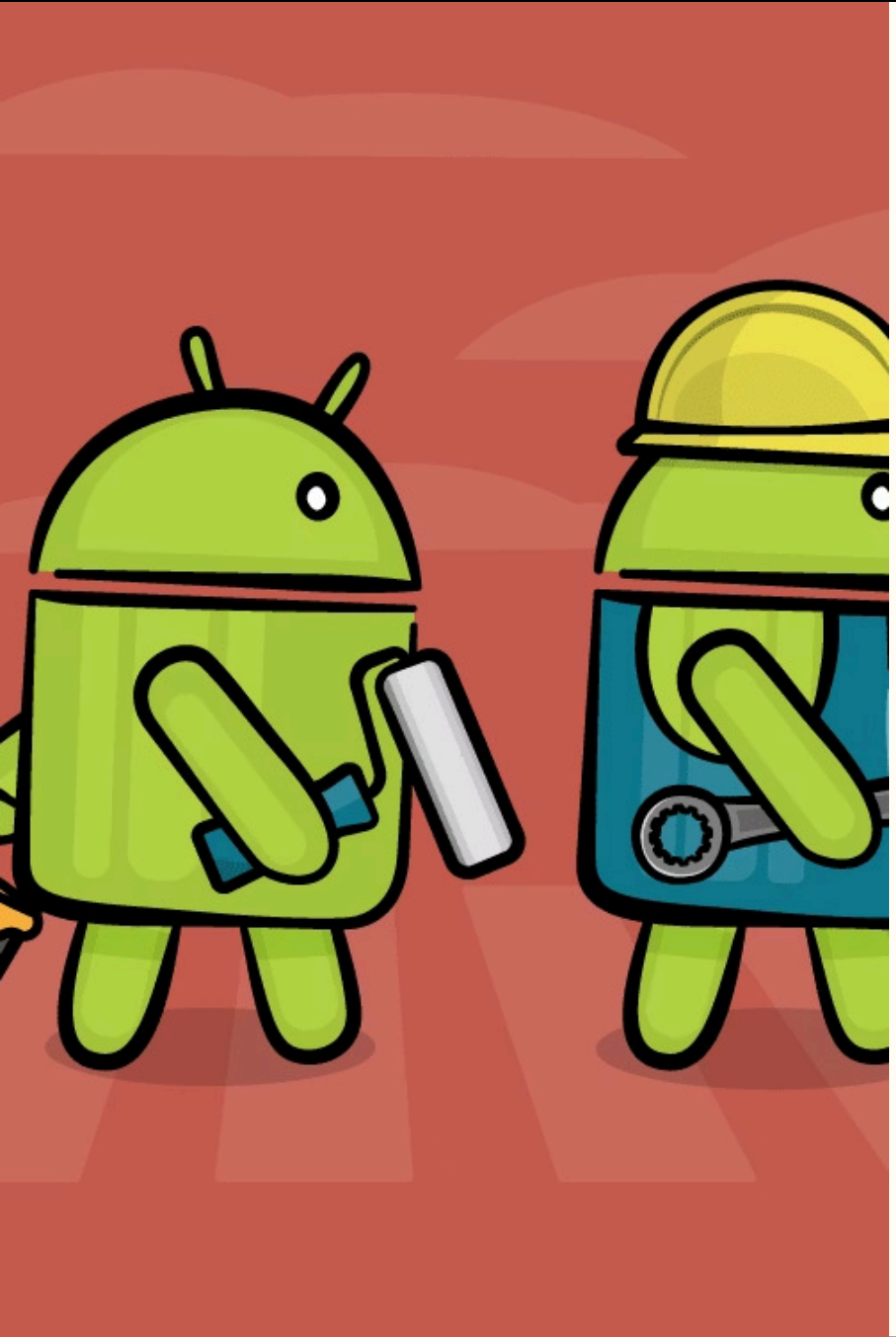
**Cas typique** :

- rotation de l'écran
- navigation vers une autre Activity
- fermeture de l'app

### Solution

- Vérifier l'état de l'Activity avant mise à jour UI
- Annuler les tâches longues dans `onDestroy()`

```
if (!isFinishing()) {
    runOnUiThread(() -> updateUI());
}
```



# Services Android

# Services Android

Dans une application Android, certaines tâches doivent s'exécuter **indépendamment de l'interface utilisateur**, même lorsque l'utilisateur change d'écran ou que l'application passe en arrière-plan. Les **Services Android** sont des composants conçus pour gérer ce type de traitements longue durée ou persistants, sans interaction directe avec l'UI.

## Un **Service** est un composant Android qui :

- s'exécute en arrière-plan
- ne fournit pas d'interface graphique
- peut continuer à fonctionner même si l'Activity est détruite

Un Service ne s'exécute pas automatiquement dans un thread séparé. Par défaut, il s'exécute dans le thread principal (UI Thread).

## Pourquoi utiliser un Service ?

Un Service est utilisé lorsque :

- une tâche doit continuer après la fermeture de l'Activity
- un traitement doit durer plusieurs secondes ou minutes
- une action doit être exécutée périodiquement ou en continu

## Exemples concrets

- lecture de musique
- téléchargement de fichiers
- synchronisation de données
- suivi de localisation
- communication avec un serveur

# Cycle de vie d'un Service

## Méthodes principales

```
onCreate()     // Initialisation
onStartCommand() // Démarrage
onDestroy()    // Nettoyage
```

Schéma logique :

1. `onCreate()`
2. `onStartCommand()`
3. Exécution du traitement
4. `onDestroy()`

Un Service se déclare dans le fichier `AndroidManifest.xml`, à l'intérieur de la balise `<application>` :

```
<service android:name=".MyService" />
```

## Cycle de vie d'un Bound Service

1. `onCreate()`
2. `onBind()`
3. Interaction Activity / Service
4. `onUnbind()`
5. `onDestroy()`

Rôle de `onBind()`

```
public IBinder onBind(Intent intent)
```

`onBind()` est appelée lorsque :

- un composant (Activity, Fragment) se lie à un Service
- via `bindService()`

Son rôle :

- retourner un objet `IBinder`
- permettant la communication entre l'Activity et le Service

`onBind()` s'exécute sur le UI Thread, toute opération lourde à l'intérieur : provoque un ANR. Le Service doit créer ses propres threads pour les traitements longs.

# Types de Services Android

1

## Service démarré (Started Service)

Un service est démarré explicitement et continue à s'exécuter jusqu'à son arrêt.

```
startService(new Intent(this, MyService.class));
```

Caractéristiques :

- indépendant de l'Activity
- continue même si l'utilisateur quitte l'application
- doit être arrêté manuellement

```
stopService(new Intent(this, MyService.class));
```

2

## Service lié (Bound Service)

Un service est lié à un ou plusieurs composants (Activity, Fragment).

```
bindService(intent, serviceConnection,
Context.BIND_AUTO_CREATE);
```

Caractéristiques :

- actif uniquement tant qu'un composant est lié
- permet une communication directe (méthodes publiques)

Utilisé lorsque l'UI doit interagir avec le service (ex. lecteur audio).

3

## Service au premier plan (Foreground Service)

Un service visible par l'utilisateur via une notification persistante.

```
startForeground(notificationId, notification);
```

Caractéristiques :

- priorité élevée
- moins susceptible d'être arrêté par le système
- obligatoire pour certaines tâches (localisation, musique)

Obligatoire à partir d'Android 8 (API 26) pour les services longue durée en arrière-plan.

# Comment démarrer un Service Android

## Créer un Service

```
public class MyService extends Service {

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        // Tâche en arrière-plan
        Log.d("MyService", "Service démarré");
        return START_NOT_STICKY;
    }

    @Override
    public IBinder onBind(Intent intent) {
        return null; // Service non lié
    }
}
```

## Déclarer le Service dans le Manifest

```
<service
    android:name=".MyService"
    android:exported="false" />
```

## Démarrer le Service depuis une Activity

```
Intent intent = new Intent(this, MyService.class);
startService(intent);
```

Le système appelle automatiquement `onStartCommand()`.

# Comment démarrer un Service Android

Lorsqu'un Service redéfinit la méthode `onStartCommand()`, il doit retourner une valeur entière indiquant au système comment réagir si le service est tué de manière inattendue (par exemple par manque de mémoire).

Les valeurs possibles sont :

- `START_STICKY`
  Le service est recréé après avoir été tué par le système, mais l'`Intent` initial n'est pas redélivré (l'`Intent` reçu est `null`). Ce mode est adapté aux services devant fonctionner en continu (ex. lecteur audio).
- `START_NOT_STICKY`
  Le service n'est pas redémarré après un arrêt par le système. Ce mode est utilisé pour des tâches ponctuelles déclenchées par un `Intent` spécifique.
- `START_REDELIVER_INTENT`
  Le service est redémarré et le dernier `Intent` est redélivré, garantissant que la tâche interrompue pourra reprendre correctement.

Depuis Android 8, l'utilisation des services en arrière-plan est limitée. Pour les tâches garanties ou différées, Foreground Service ou WorkManager sont recommandés.

# Comment démarrer un Foreground Service

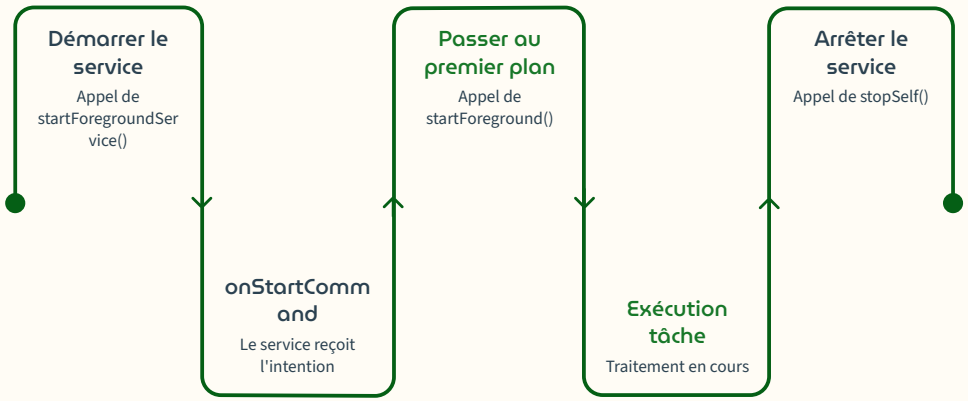
## Étape 1 – Déclaration dans le manifeste

```
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>
<service
   android:name=".MyForegroundService"
   android:exported="false" />
```

## Étape 3 – Démarrage depuis une Activity

```
Intent intent = new Intent(this, MyForegroundService.class);
ContextCompat.startForegroundService(this, intent);
```

- Service visible avec notification , Priorité élevée, Obligatoire pour tâches longues (Android 8+)




## Étape 2 – Création du service

```
public class MyForegroundService extends Service {
   @Override
   public int onStartCommand(Intent intent, int flags, int startId) {
      createNotification();
      new Thread(() -> {
         doLongTask();
         stopSelf();
      }).start();
      return START_NOT_STICKY;}
   private void createNotification() {
      Notification notification =
            new NotificationCompat.Builder(this, "channel_id")
             .setContentTitle("Foreground Service")
             .setContentText("Task running...")
             .setSmallIcon(R.drawable.ic_launcher_foreground)
                .build();
      startForeground(1, notification);}
   @Override
   public IBinder onBind(Intent intent) {
      return null;
   }
}
```

# Services et Threading

## Erreur courante

Exécuter un traitement long directement dans un Service :

```
public int onStartCommand(...) {
    doLongTask(); // BLOQUE le UI Thread
}
```

Provoque :

- ANR (Application Not Responding)
- gel de l'application

## Bonne pratique

Utiliser un **Thread**, **Executor** ou **Coroutine** à l'intérieur du Service.

```
new Thread(() -> doLongTask()).start();
```

# JobScheduler

`JobScheduler` est une API Android (API 21+) qui permet de planifier des tâches en arrière-plan (*jobs*) à exécuter selon certaines conditions, afin d'économiser la batterie et les ressources.

`JobScheduler` permet au système :

- de regrouper les tâches
- d'optimiser la consommation
- de décider du meilleur moment pour exécuter le travail

## Un Job :

- n'est pas exécuté immédiatement
- attend que les conditions soient remplies

Exemples de conditions :

- connexion réseau disponible
- appareil en charge
- batterie suffisante
- stockage non saturé

# JobScheduler

## a) Créer un JobService

```
public class SyncJobService extends JobService {

    @Override
    public boolean onStartJob(JobParameters params) {
        // tâche en arrière-plan
        return false; // job terminé
    }

    @Override
    public boolean onStopJob(JobParameters params) {
        return true; // relancer si interrompu
    }
}
```

## b) Planifier le Job

```
ComponentName componentName =
        new ComponentName(this, SyncJobService.class);

JobInfo jobInfo = new JobInfo.Builder(1, componentName)
        .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
        .setRequiresCharging(true)
        .build();

JobScheduler scheduler =
        (JobScheduler) getSystemService(JOB_SCHEDULER_SERVICE);

scheduler.schedule(jobInfo);
```

# WorkManager

`WorkManager` est une API Android destinée à exécuter des tâches en arrière-plan de manière fiable,
même si :

- l'application est fermée
- le téléphone redémarre
- le système tue le processus

Contrairement à un `Service`, le système garantit l'exécution de la tâche.

## Problème des Services

- Le système peut arrêter un Service à tout moment
- Forte consommation de batterie
- Restrictions sévères depuis Android 8+

## Solution

`WorkManager` choisit automatiquement le meilleur mécanisme :

- `JobScheduler`
- `AlarmManager`
- `Foreground Service` (si nécessaire)

# WorkManager

**Exemple**

## a) Créer un Worker

```
public class SyncWorker extends Worker {

    public SyncWorker(@NonNull Context context,
                      @NonNull WorkerParameters params) {
        super(context, params);
    }

    @NonNull
    @Override
    public Result doWork() {
        // tâche en arrière-plan
        return Result.success();
    }
}
```

## b) Lancer le travail

```
WorkRequest workRequest =
        new OneTimeWorkRequest.Builder(SyncWorker.class).build();

WorkManager.getInstance(this).enqueue(workRequest);
```

## Exemples concrets

Synchroniser des données avec un serveur

Sauvegarde automatique

Uploade d'images en arrière-plan

Nettoyage de cache périodique

***Même si l'utilisateur quitte l'application, la tâche sera exécutée.***

# Bonnes pratiques et erreurs courantes en programmation concurrente Android

## Bonnes pratiques et erreurs courantes en programmation concurrente Android

1. **Toujours exécuter les tâches lourdes hors du thread UI**
   - Thread, ExecutorService, WorkManager pour éviter blocage de l'UI.
2. **Mettre à jour l'UI uniquement depuis le thread principal**
   - Utiliser `runOnUiThread()`, `Handler.post()`, `View.post()`.
3. **Utiliser un pool de threads plutôt que de créer de nombreux threads**
   - `ExecutorService` ou `Executors.newFixedThreadPool()`.
4. **Préférer WorkManager pour les tâches différées ou fiables**
   - Garantit l'exécution même après redémarrage du téléphone.
5. **Libérer les ressources correctement**
   - Arrêter les services (`stopService()`, `stopSelf()`)
   - Shutdown les `ExecutorService` avec `shutdown()`.
6. **Gérer le cycle de vie Android**
   - Ne pas laisser un thread continuer après la destruction d'une Activity si non nécessaire.

## Bonnes pratiques et erreurs courantes en programmation concurrente Android

| Besoin | Solution |
| --- | --- |
| Tâche courte liée à l'UI | Thread / Executor |
| Tâche longue visible | Foreground Service |
| Tâche différée, fiable | WorkManager |
| Tâche périodique | WorkManager |

# Conclusion

| Solution | Description | Utilisation typique | Avantages | Limites |
|---|---|---|---|---|
| **Thread / Runnable** | Fil d'exécution simple | Tâches courtes, calculs, I/O ponctuelle | Simple, direct | Ne gère pas la durée de vie, pas adapté pour UI ou tâches longues |
| **Executor / ExecutorService** | Gestion de pool de threads | Tâches répétitives ou parallèles | Gestion automatique des threads, réutilisation de threads, meilleure performance | Toujours manuel pour la mise à jour de l'UI |
| **Service** | Composant Android pour tâches en arrière-plan | Tâches longues, indépendantes de l'UI | Fonctionne même si activité fermée, cycle de vie Android intégré | Ne garantit pas l'exécution si le système ferme l'app, pas optimisé batterie |
| **Foreground Service** | Service visible à l'utilisateur (notification persistante) | Musique, GPS, téléchargement | Ne peut pas être tué facilement par le système, tâche longue et critique | Doit afficher une notification, consommateur de batterie |
| **WorkManager** | API haut-niveau pour tâches différées et fiables | Synchronisation, sauvegarde, upload, tâches périodiques | Fiable même si app fermée ou redémarrage, choisit automatiquement le mécanisme interne (JobScheduler / AlarmManager / Foreground Service) | Moins précis en timing exact, pas pour tâches temps réel strictes |