UNITÉ D ’ENSEIGNEMENT (UE) : DÉVELOPPEMENT MOBILE

Ch6 : Threads

Importance des threads sous Android

- **L'interface doit toujours être (ré)active !**
  - pas de calculs longs, bloquants ou infinis
  - pas d'accès aux ressources coûteuses
- **Exemples :**
  - ressources coûteuses : réseau, internet
  - calcul bloquant : attente d'une entrée de l'utilisateur, attente de connexion
  - calcul infini : écoute sur un réseau

Threads sous Android

- **Thread principal, ou UI thread :**
  - le thread dans lequel s'exécute l'application au démarrage
  - gère tout ce qui est interface
  - seul thread à avoir accès à l'interface
  - possibilité de faire des calculs peu coûteux
- **Autres threads, ou worker threads :**
  - au programmeur de les gérer
  - effectuent les calculs coûteux, les accès aux ressources coûteuses (dont le réseau)
  - aucun accès à l'interface ⇒ communication avec le UI thread

But atteint

- **Fluidité de l'interface :**
  - un thread pour l'interface, n'attendant pas de résultats ou de ressources coûteux
  - d'autres threads en tâches de fond pour les résultats et ressources coûteuses

  → **répartition des tâches obligatoire**
- **Autres applications :**
  - naturellement, on peut utiliser les threads pour toutes leurs autres applications (parallélisme)
  - efficacité, mais pas d'obligation

Implantations

- **Threads de Java :**

  + toute la généralité des threads (parallélisme)

  - toute la complexité des threads (ressources partagées)

  - communication avec le thread difficile

- **La classe `AsyncTask` :**

  + haut niveau

  + garanties de synchronisation

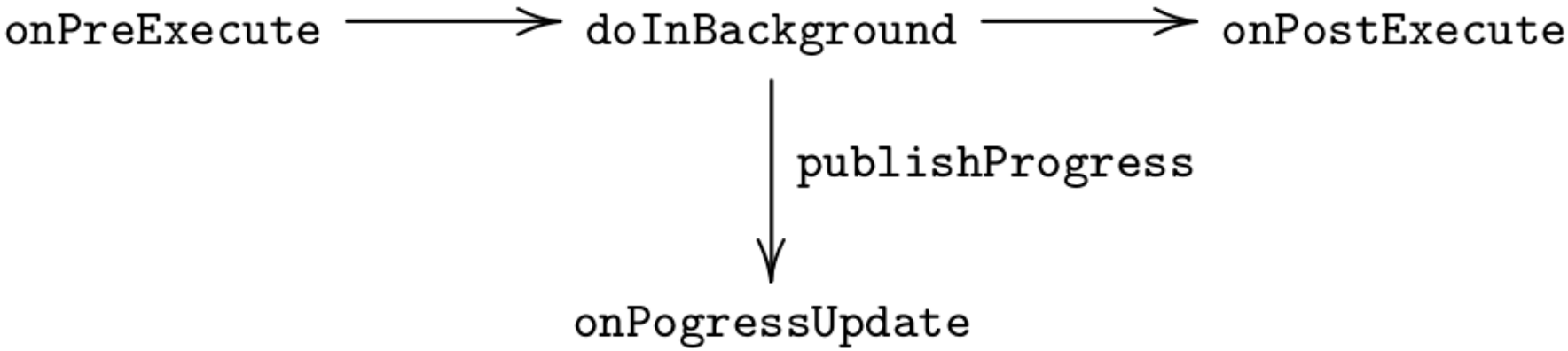
  - parallélisme uniquement avec le thread principal (pas entre `AsyncTasks`)

La classe AsyncTask

- **class AsyncTask<Params, Progress, Result>**
  - Params : type des paramètres envoyés à la tâche (entrée)
  - Progress : type des résultats fournis au fur et à mesure de l'exécution de la tâche (sortie)
  - Result : type du résultat de l'exécution de la tâche (sortie)
- **Principe :**
  - faire une classe Tache héritant de `AsyncTask`
  - surcharger quelques méthodes
  - lancer la tâche (ex : `new Tache().execute()`)

Méthodes à surcharger

- **code à exécuter par la tâche :** `doInBackground`
- **code à exécuter pour mettre à jour l'interface :**
  - avant l'exécution de la tâche : `onPreExecute`
  - pendant l'exécution de la tâche : `onPogressUpdate`
  - après l'exécution de la tâche : `onPostExecute`



Inconvénient

- **AsyncTask inclus dans Activity**

  ⇒ AsyncTask non utilisable ailleurs

- **Solution :**
  - Créer AsyncTask indépendant
  - Créer interface implémentée par Activity
  - AsyncTask.onPostExecute envoie les données à Activity via l'interface

Exemple

```
public class AttenteThread extends AsyncTask <Void, Integer, Integer> {
    private final int wait = 5000;
    private final int number = 6;
    @Override
    protected void onPreExecute() {
      affichage.setText("Lancement du thread..."); }
    @Override
    protected Integer doInBackground(Void... voids) {
      for (int count = 0; count < number; count++) {
        Thread.sleep(wait);
        publishProgress(count+1);    }
      return number;   }
    @Override
    protected void onProgressUpdate(Integer... counts) {
      int time = counts[0] * wait / 1000;
      affichage.setText("Le thread s'execute depuis " + time + " secondes");    }
    @Override
    protected void onPostExecute(Integer res) {
      int time = res * wait / 1000;
      affichage.setText("Le thread a fini ; il s'est execute pendant " + time + " secondes");  } }
```

Interruption d’une tâche

- **On peut vouloir interrompre une tâche en cours d’exécution :**
  - quand l’utilisateur quitte l’activité
  - quand l’utilisateur appuie sur un bouton
  - au bout d’une trop longue durée (timeout)
  - ...
- **On signale à la tâche qu’on souhaite l’interrompre :**
  - Méthode : `tache.cancel(true)`
  - à elle ensuite de véritablement s’interrompre

Interruption d’une tâche

- **Une tâche peut déterminer si elle doit s’interrompre :**
  - la méthode `isCancelled`() renvoie true
- **Lorsque c’est le cas :**
  - la tâche doit arrêter son calcul
  - elle peut ensuite encore effectuer quelques opérations
  - Ex dans le corps d’une boucle infinie :
    ```
    if(isCancelled())  break;
    ```

Pas de parallélisme entre AsyncTasks

```
tache1.execute();
tache2.execute();
tache2.doInBackground
tache1.doInBackground
```

appelée que lorsque a fini

Multi-threading

- **Autres possibilités : utiliser les threads Java et les Runnable, ou bien les HandlerThread de Android**
- **Problématique : gérer le nombre de threads lancées en parallèle et s'assurer que le système va pouvoir y répondre**
- **Android propose une file d'attente pour éviter de multiplier les threads**

La classe Thread

- **Séparation avec le UI thread :**
  - code qui sera exécuté par la tâche de fond
  - pour mettre à jour l'interface : "poster" des messages au thread principal
- **Principe :**
  - faire une classe `Fils` héritant de `Thread` (ou implantant Runnable)
  - surcharger la méthode `run`
  - lancer la tâche (ex :`new Fil().start()`)

Interruption d’un fils

- **Même principe :**
  - on signale au fils de s’interrompre
  - à lui de gérer cela pour s’arrêter
- **Autre syntaxe :**
  - signalement : appel à `interrupt()`
  - savoir si on est interrompu : test `interrupted()`
- **`Exemple` :**

```
fils1.start();
fils2.start();
```

Les deux fils sont lancés et s’exécutent “simultanément” (ordre d’exécution non déterministe)

Conclusion

- **Utilisation de worker threads :**
  - dès qu'on a une tâche longue
  - attention à la mise à jour de l'interface
- **Choix de l'implantation :**
  - **`AsyncTask`** : simple à utiliser, facile de modifier l'interface en cours de route
  - **Thread** : si on a besoin de parallélisme entre plusieurs worker threads