Applications client/serveur

Chantal Keller

17 septembre 2015

Plan

Rendu des TDs 2 et 3

TD2 :

- 12/16 (+ 1 excusé)
- interfaces et interaction utilisateur bien compris
- attention à l'algo même si ce n'est pas le sujet du cours (en testant, pas mal d'erreurs repérables)

TD3 :

- 13/16
- quelques applications rendues crashent... !
- lancement et vie des activités bien compris
- attention à l'arrêt d'une activité

Rappel : arrêt d'une activité

- On la dépile pour revenir à l'activité précédente (celle qui l'a lancé)
- Pas d'utilisation de `startActivity`
- Mais appel à la méthode `finish`

Projet

- Énoncé d'ici la fin de la semaine
- Groupes de deux : liste à faire pour vendredi 25/9 (essayer d'équilibrer)
- Avez-vous des demandes ? On peut voir ensemble pendant ces deux séances, ou par mail avant vendredi 25/9

Plan

Au sein d'une activité

- Décrire l'interface statique en XML
- Récupérer au niveau Java les objets de l'interface ("vues") : `findViewById`, ...
- Appeler des méthodes sur ces objets pour les faire évoluer dynamiquement
- Associer des actions à des objets, notamment des boutons : `public void action(View v) { ... }`
- Vie : cycles de création/destruction, mise en pause/reprise, arrêt/reprise

Légende : maîtrisé, abordé

Plusieurs activités

- Lancement : `startActivity`
- Arrêt : `finish`
- Passage de messages : intentions, associent des valeurs (de type quelconque) à des clés (de type `String`)
- Demande de réponse : lancement d'une activité en posant une question (de type `int`)

À venir

Applications client/serveur :

- Serveur proposant un service
- Client utilisant ce service
- Souvent, par le réseau

Aspects abordés

- Réseau : ouverture/fermeture de ports, connexion
- Système : threads Android

Plan

Généralités

Machines sur un réseau :

- identifiées par leur adresse IP (ex : 192.168.1.247) (un DNS peut leur associer un nom)
- offrant des ports de connexion selon le protocole, par ex :
  - HTTP : 80
  - HTTPS : 443
  - SMTP : 25
  - SSH : 22

Application client/serveur

Le serveur :

- possède une adresse IP
- écoute sur un port donné (choisi par le protocole)

Les clients :

- établissent une connexion sur ce port
- échangent des données avec le serveur *via* cette connexion

En Java (et Android)

Connexion :

- *sockets* serveur et client
- avec un flux sortant et un flux entrant

Plan

Généralités sur les threads

Deux buts principaux :

- parallélisme (tirer profit des machines multi-cœurs) : effectuer plusieurs calculs simultanément
- tâches de fond communiquant avec la tâche principale

Difficultés :

- modèle d'exécution parallèle (*scheduler*)
- non déterminisme
- points de synchronisation

Threads en Android

But principal :

- tâches de fond communiquant avec la tâche principale

API Android :

- classe Java/Android de haut niveau pour manipuler des threads
- Java garantit certains points de synchronisation

↪ démo

Le thread principal et les autres

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 (ex. TD2 : addition de deux entiers)

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**

But

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

Principe

Android fournit une classe `AsyncTask` avec des méthodes ayant pour but de :

- définir le code à exécuter par le *worker thread*
- définir le code à exécuter pour mettre à jour l'interface avant, pendant et après l'exécution du *worker thread*

Principe :

- faire une classe `NewThread` héritant de `AsyncTask`
- surcharger les méthodes ci-dessus
- lancer le thread (ex : `new NewThread().execute()`)

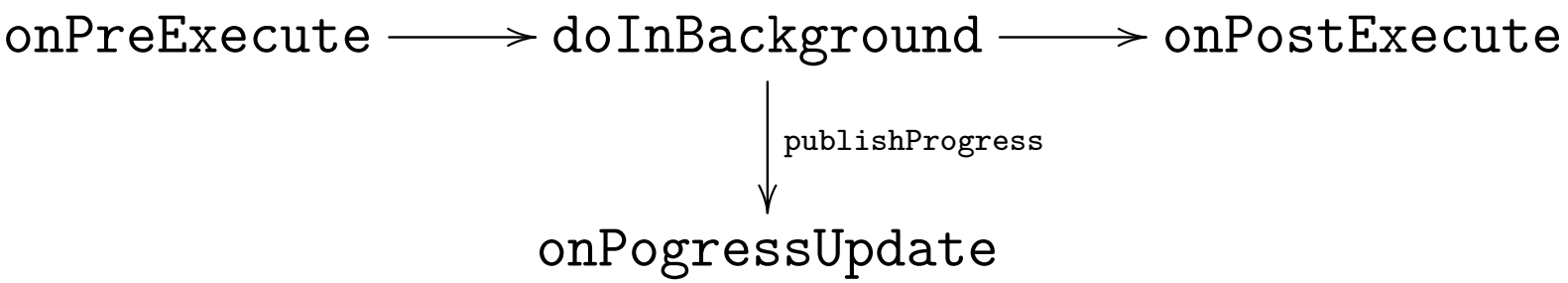
↪ similaire à l'idée de surcharger `onCreate`, `onStart`, ... dans les classes héritant de `Activity`

Détails

Android fournit une classe `AsyncTask` permettant de :

- définir le code à exécuter par le *worker thread* : `doInBackground`
- définir le code à exécuter pour mettre à jour l'interface :
  - avant l'exécution du worker thread : `onPreExecute`
  - pendant l'exécution du worker thread : `onPogressUpdate`
  - après l'exécution du worker thread : `onPostExecute`

Garanties de synchronisation :



Un peu de généricité

```
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)

Répercussion sur la signature des méthodes

```
class AsyncTask<Params, Progress, Result>
```

Méthodes à surcharger :

- `Result doInBackground (Params... params)`
- `void onPreExecute ()`
- `void onProgressUpdate (Progress... values)`
- `void onPostExecute (Result result)`

Méthodes qu'on peut appeler :

- `void publishProgress (Progress... values)`

Exemple

```
private class WaitingThread 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'un thread

Deux méthodes de la classe `AsyncTask` que l'on peut utiliser :

- à l'extérieur du thread :

  ```
  boolean cancel (boolean mayInterruptIfRunning)
  ```

  interrompt le thread
- à l'intérieur du thread :

  ```
  boolean isCancelled ()
  ```

  teste si le thread a été interrompu

Plan

Conseil

Lire les TDs en entier avant de les commencer :

- avoir un aperçu de l'application qu'on cherche à écrire
- vrai dans n'importe quelle matière et dans votre travail

Aujourd'hui

TD :

- Client de chat (dont le serveur va être projeté)
- Puis une application implantant le serveur correspondant

Pour utiliser le serveur projeté :

- réseau local
- utiliser les machines de l'IUT
- enregistrer dans `Z:\`