Developpement mobile – Thread, AsyncTask et servivce

AKKA ZEMMARI

ZEMMARI@LABRI.FR

Android et les tâches lentes

- Problèmes
  - Une application peut nécessiter une opération qui prend beaucoup de temps.
  - L'interface graphique (UI) est bloquée pendant toute la durée de l'opération.
  - Android détruit les UI qui ne répondent pas depuis trop longtemps.
- Objectif
  - Conserver l'UI interactive malgré l'opération en cours
- Solutions Android
  - Background thread : faire les opérations lourdes dans un thread
  - Background service : faire les opérations lourdes dans un service en tâche de fond, en envoyant des notifications à l'utilisateur à propos de la prochaine étape

Threads

Thread : modèle

- Un Thread est un unité d'exécution **concurrente**.
- Chaque thread a sa propre **pile d'appel** (*call stack*). La pile est utilisée à chaque appel de méthode, passage de paramètre et stockage pour les variables locales.
- Chaque instance de machine virtuelle à au moins un **thread principal**.
- Les Threads dans la même VM interagissent et se synchronisent en utilisant des **objets partagés** et des **moniteurs** associés à ces objets.

Thread : exécution

- Il y a 2 manières d'exécuter un code dans un Thread
  - Créer une nouvelle classe qui étend (extends) la classe Thread et surcharger la méthode run().

    ```
    Runnable myRunnable1 = new MyRunnableClass();
    ```

  - Créer une nouvelle instance de Thread en lui passant un objet Runnable.

    ```
    Thread t1 = new Thread(myRunnable1);
    t1.start();();
    ```

- Dans les 2 cas, la méthode start() doit être appelée pour exécuter réellement le nouveau Thread.

Thread : exécution

```
public class MainActivity extends Activity {

	@Override
	public void onCreate( Bundle savedInstanceState) {

		super.onCreate(savedInstanceState);

		setContentView(R.layout.main);
		// on peut aussi encapsuler le code dans une méthode privée

		new Thread(new Runnable() {

			@Override

			public void run() {

				try {
					Thread.sleep(2000);

					Log.i("Thread" , "coucou");

				} catch (InterruptedException e) {

					e.printStackTrace();

				}

			}

		}).start();

}//onCreate
```

Avantages du multithreading

- Les Threads partagent les ressources du processus mais peuvent s'exécuter en parallèle.
- Les responsabilités peuvent être partagées
  - Le thread principal exécute l'UI
  - Les tâches lentes sont envoyées dans des threads en tache de fond
- Le threading offre une abstraction utile pour la programmation concurrente
- Un système multithreadé s'exécute plus vite sur un système qui possède plusieurs CPU.

Inconvénients du multithreading

- Le code tend à être plus complexe
- Il faut détecter, éviter et résoudre les interblocages (deadlocks)

Interagir avec un Thread

- Le thread principal maintient une file de message qui gère les interactions avec les objets applicatifs (ex : activités) et les UI associées.
  - La file de message contient soit des objets de type Message, soit de type Runnable.
- On peut créer ses propres threads secondaires et les faire communiquer avec le thread principal en utilisant la classe Handler.
  - Le nouvel Handler est lié à la file d'attente du thread dans lequel il a été créé.
  - Il peut récupérer les objets de la file de messages

La classe Handler

- http://developer.android.com/reference/android/os/Handler.html
- Il y a 2 façons principales d'utiliser un Handler :
  - Pour envoyer un message destiné à déclencher l'exécution d'une action dans un autre thread.
  - Pour planifier des actions futures (runnables)

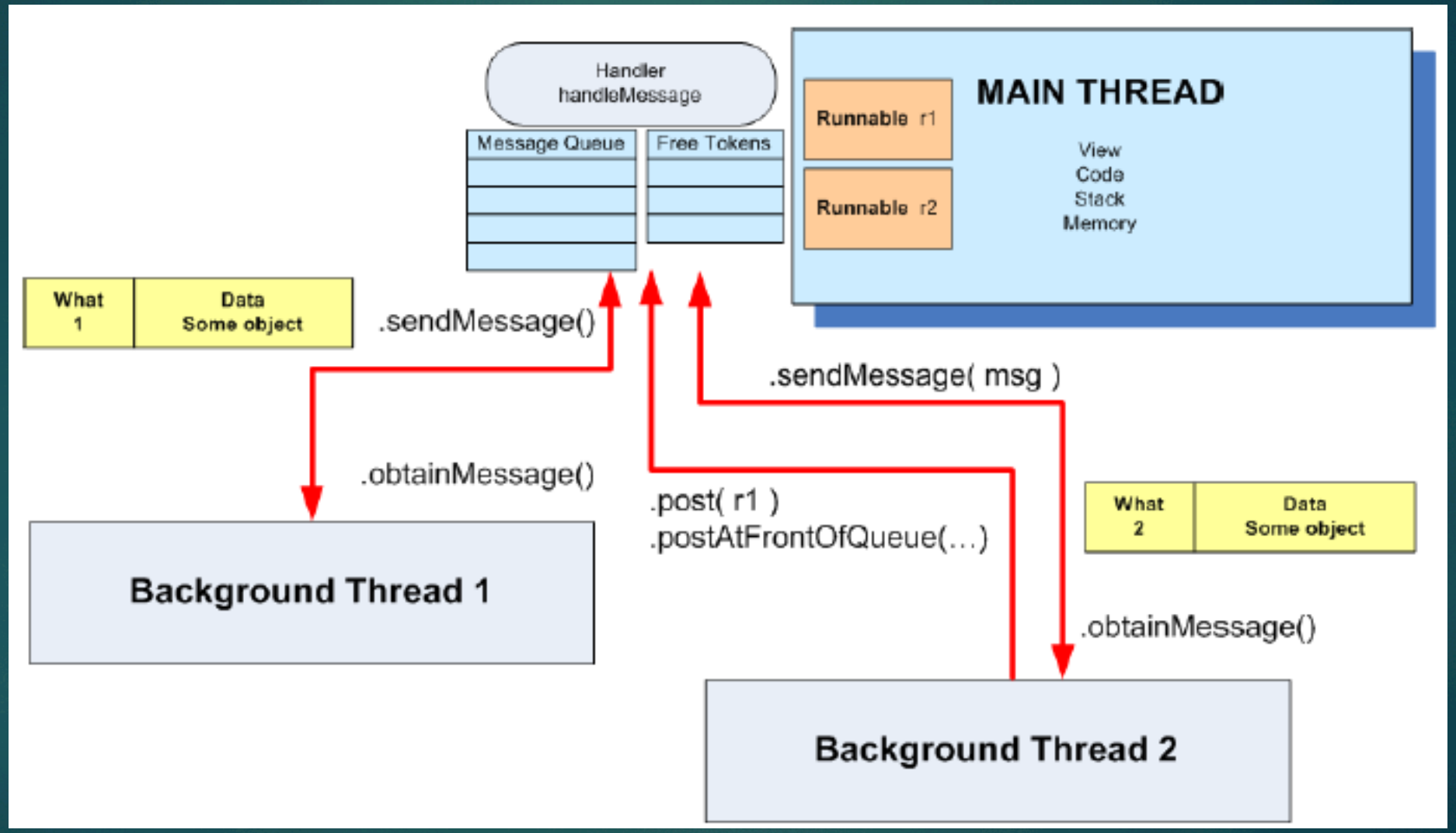
Thread et UI

- Les Threads secondaires ne sont pas autorisés à interagir avec l'interface graphique.
- Seul le processus principal peut accéder à la vue de l'activité
- Les variables globales de classe peuvent être vues et mises à jour dans les Threads

Communication entre Threads

- Thread secondaire
  - Un thread secondaire qui veut communiquer avec le thread principal doit créer un nouveau Message par la méthode obtainMessage().
  - Une fois obtenu, le thread secondaire peut remplir le message avec des données et l'insérer dans la file d'attente du handler par la méthode sendMessage().
- Handler du thread principal
  - Le handler utilise la méthode handleMessage() pour se mettre en attente de nouveaux messages à destination du thread principal.
  - Un message extrait de la file d'attente peut soit transporter des données à destination du thread principal, soit demander l'exécution d'objets exécutables (runnable) avec la méthode post().

Communication entre Threads



Communication par Message

Main Thread

```
...
Handler myHandler = new Handler() {

  @Override
  public void handleMessage(Message msg) {

     // do something with the message...
     // update GUI if needed!
     ...
  }//handleMessage

};//myHandler

...
```

En attente des messages

Background Thread

```
...
Thread backgJob = new Thread (new Runnable (){

  @Override
  public void run() {
   //...do some busy work here ...
   //get a token to be added to
    //the main's message queue
   Message msg = myHandler.obtainMessage();
     ...
    //deliver message to the
   //main's message-queue
   myHandler.sendMessage(msg);
  }//run

});//Thread

//this call executes the parallel thread
backgroundJob.start();
...
```

Envoie des messages

Communication par Message

- Les méthodes SendMessage
  - sendEmptyMessage() : crée un message vide et l'insère dans la file immédiatement
  - sendMessage() : insère un message « donnée » dans la file d'attente immédiatement
  - sendMessageAtFrontOfQueue() : insère le message en tète de la file d'attente (message prioritaire)
  - sendMessageAtTime(): insère le message dans la file à un moment futur, exprimé en millisecondes (SystemClock.uptimeMillis())
  - sendMessageDelayed() : insère le message dans la file d'attente après un délai, exprimé en millisecondes

Exemple d'activité avec un Thread

```
public class ThreadActivity extends Activity {

    static public final int BACKUP_OK = 0;

    private TextView contentView = null;
    private Button buttonStart = null;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        contentView = (TextView)this.findViewById(R.id.contentView);
        buttonStart = (Button)this.findViewById(R.id.button1);

        buttonStart.setOnClickListener(new OnClickListener() {
            @Override
            public void onClick(View v) {
                backup();
            }
        });

    }
```

Exemple d'activité avec un Thread

```
// abonnement aux messages du Thread
private Handler handler = new Handler() {

    public void handleMessage(android.os.Message msg) {
        if(msg.what == BACKUP_OK) {
            contentView.setText("Backup done.");
        }
    };

};

private void backup(){
    //on effectue le travail dans un Thread séparé
    new Thread(new Runnable() {
        @Override
        public void run() {
            //travail bidon
            for(int i=0;i<100000;i++){}
            // notification de la fin de tache au main thread
            handler.sendEmptyMessage(0);
        }
    }).start();
}
```

Communication par Runnable

Main Thread

```
...
Handler      myHandler = new Handler();
@Override
public void onCreate(
          Bundle savedInstanceState) {
  ...
 Thread myThread1 =
         new Thread(backgroundTask,
                    "backAlias1");
 myThread1.start();

}//onCreate

...
//this is the foreground runnable
private Runnable foregroundTask
   = new Runnable() {
  @Override
  public void run() {
   // work on the UI if needed
}
...
```

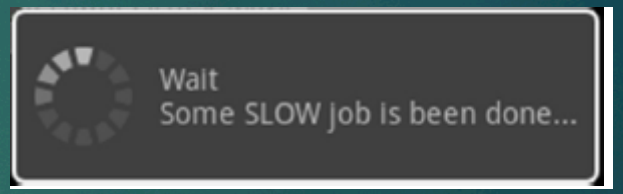
Background Thread

```
// this is the "Runnable" object
// that executes the background thread

private Runnable backgroundTask
                  = new Runnable () {
  @Override
  public void run() {
     ... Do some background work here
    myHandler.post(foregroundTask);

  }//run

};//backgroundTask
```

AsyncTask

AsyncTask : modèle

- http://developer.android.com/reference/android/os/AsyncTask.html
- La classe AsyncTask permet d'exécuter des opérations en tâche de fond et de publier les résultats dans l'interface graphique sans avoir à manipuler de thread ou de handler.
- Une tâche asynchrone est définie par:



3 Types Génériques	4 Etats	1 Méthode auxiliaire
Params, Progress, Result	**onPreExecute, doInBackground, onPostExecute**	**publishProgress**

La classe AsyncTask

- AsyncTask<Params,Progress,Result>
- Types génériques :
  - Params : le type des paramètres d'input envoyés à la tâche à l'exécution
  - Progress : le type des unités de progression publiées pendant l'exeécution en tâche de fond .
  - Result : le type de résultat de l'exécution en tâche de fond
- Remarque : Tous les types ne sont pas supportés par les tâches asynchrones. Pour marquer un type inutilisé, indiquer Void
- Note: la sytaxe "String ..." indique (Varargs) un tableau de String, (similaire à "String[]").

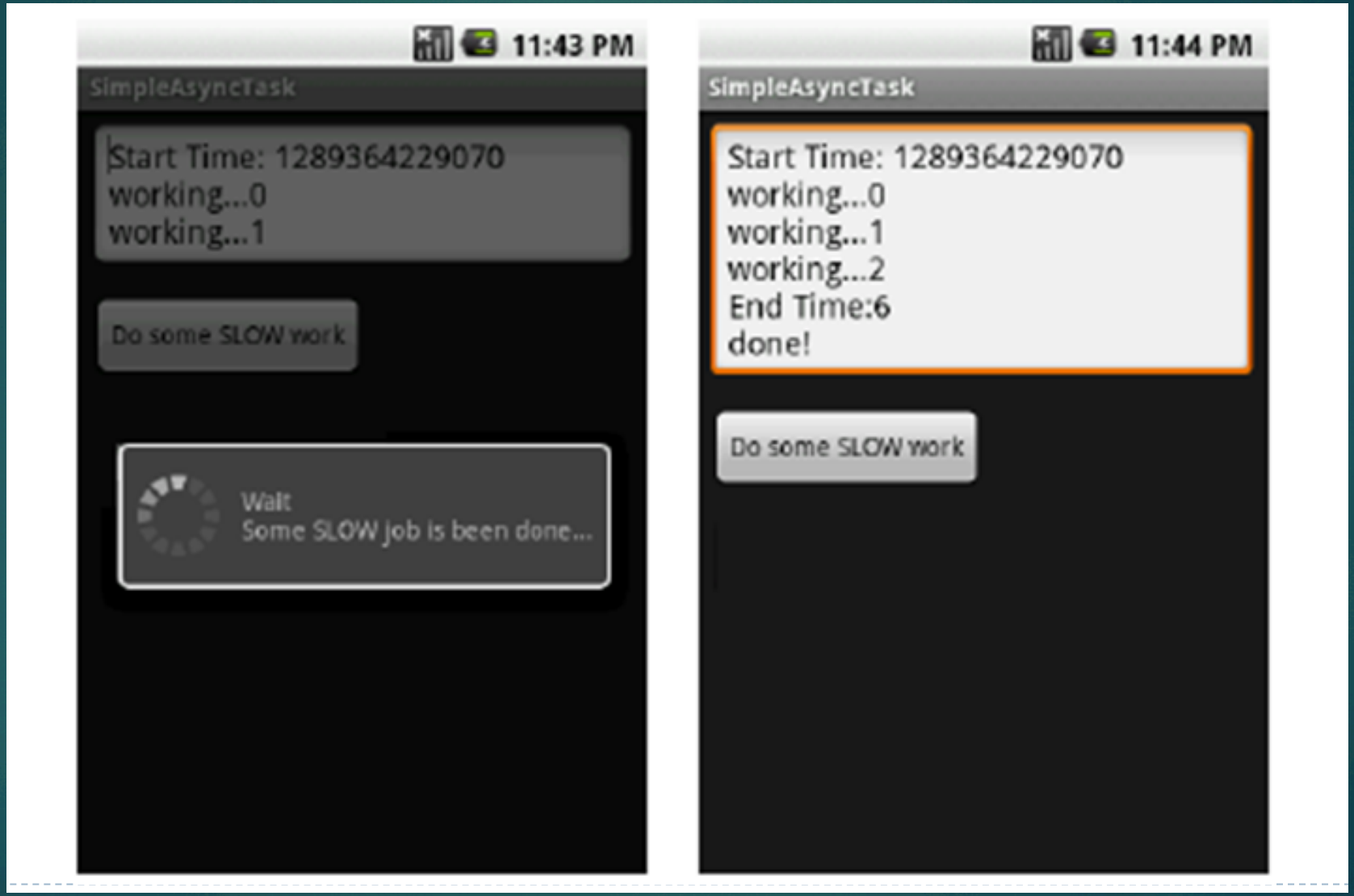
AsyncTask : méthodes

```
public private class VerySlowTask extends AsyncTask<String, Long,Void> {
// Démarre : initialise les données locales, peut accéder à l'UI
protected void onPreExecute() {}
// C'est la tâche de fond qui effectue l'opération en tâche de
//fond et ne peut pas changer directement l'UI

protected Void doInBackground(final String... args) {
    publishProgress((Long) someLongValue);
}
// mises à jour périodiques. Peut changer l'UI
@Override
protected void onProgressUpdate(Long... value) {}
// fin. Peut utiliser l'UI ici.
protected void onPostExecute(finalVoid unused) {} }
```

AsyncTask : méthodes

Les méthodes d'AsyncTask
onPreExecute() : invoquée au démarrage. Cette étape est normalement utilisée pour initialiser la tâche, par exemple en montrant une barre de progression à l'interface.
doInBackground(Params... : invoquée juste après la fin de *onPreExecute*(). Cette étape est utilisée pour effectuer l'action en tâche de fond à proprement parlé et peut prendre beaucoup de temps. Les paramètres de la tâche asynchrone sont passés à cette étape. Le résultat de l'opération doit être retourné par cette étape et seront passés à la dernière étape. Cette étape peut aussi utiliser *publishProgress(Progress...)* tpour publier 1 ou plusieurs unités de progression.Ces valeur sont publiées au thread de l'UI, à l'étape *onProgressUpdate(Progress...)*.
onProgressUpdate(Progress...) : invoqué par le thread de l'UI après un appel à la méthode *publishProgress(Progress...)*. Le timing de l'exécution est indéfinie.Cette méthode est utilisée pour afficher n'importe quelle forme de progression dans l'UI pendant que la tâche de fond est toujours en cours d'exécution., pour exemple pour animer une barre de progression ou pour montrer un log dans un champ texte.
onPostExecute(Result), : invoquée dans le thread de l'UI après la fin de l'exécution de la tâche. Le résultat de l'opération de tâchde de fond est passé à cette étape en paramètre.

AsyncTask : Exemple



Activité principale

```
public class AsyncActivity extends Activity {

    Button btnSlowWork;
    EditText txtMsg;
    Long startingMillis;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
        txtMsg = (EditText) findViewById(R.id.EditText01);
        // slow work...for example: delete all data from a database
        btnSlowWork = (Button) findViewById(R.id.Button01);
        this.btnSlowWork.setOnClickListener(new OnClickListener() {

            public void onClick(final View v) {
                new VerySlowTask().execute();
            }
        });
    }// onCreate

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        // Inflate the menu; this adds items to the action bar if it is present.
        getMenuInflater().inflate(R.menu.main, menu);
        return true;
    }
```

VerySlowTask

```
//--------------- inner class
private class VerySlowTask extends AsyncTask <String, Long, Void> {
    private final ProgressDialog dialog = new ProgressDialog(AsyncActivity.this);
    // can use UI thread here
    protected void onPreExecute() {
        startingMillis = System.currentTimeMillis();
        txtMsg.setText("Start Time: " + startingMillis);
        this.dialog.setMessage("Wait\nSome SLOW job is being done...");
        this.dialog.show();
    }
    // automatically done on worker thread (separate from UI thread)
    protected Void doInBackground(final String... args) {
        try {
            // simulate here the slow activity
            for (Long i = 0L; i < 3L; i++) {
                Thread.sleep(2000);
                publishProgress((Long)i);
            }
        } catch (InterruptedException e) {
            Log.v("slow-job interrupted", e.getMessage());
        }
        return null;
    }
```

VerySlowTask

```
    // periodic updates - it is OK to change UI
    @Override
    protected void onProgressUpdate(Long... value) {
        super.onProgressUpdate(value);
        txtMsg.append("\nworking..." + value[0]);
    }
    // can use UI thread here
    protected void onPostExecute(final Void unused) {
        if (this.dialog.isShowing()) {
            this.dialog.dismiss();
        }
        // cleaning-up, all done
        txtMsg.append("\nEnd Time:");
        txtMsg.append(""+(System.currentTimeMillis()-startingMillis)/1000);
        txtMsg.append("\ndone!");
    }
}// VerySlowClass
```

TP 04 : Tâches de fond

- A récupérer ici :
  - http://www.labri.fr/~zemmari/pam