Eclipse – Android SDK
Les services

Les services

- Les services permettent la prise en charge d'opérations sans interface utilisateur.
- Ils disposent d'une priorité supérieure à celle des activités inactives, ils seront donc nettoyés en dernier en cas de besoin en ressources.
- Un service « tué » par le système peut être configuré pour redémarrer automatiquement dès que les ressources deviennent suffisantes.
- Android dispose de deux classes de base pour la création d'une classe « service métier » :
  - la classe **Service**,
  - la classe **IntentService**.

Les services

- La classe **Service** :
  - Par défaut le service s'exécute sur le thread principal de l'application, le traitement est synchrone.
  - Si l'on désire déléguer le traitement à un thread il faut le coder en utilisant la classe **Thread** ou **AsyncThread**.
  - Il est possible d'imposer l'exécution du service dans un processus indépendant, ce qui complique la mise en place de la communication entre l'activité client et le service.

- La classe **IntentService** :
  - Elle automatise le traitement asynchrone des requêtes au service.
  - Elle gère une file d'attente des requêtes et traite celles-ci dans des threads dédiés.

La classe Service

La classe Service à implémenter doit hériter de la classe **Service** et surcharger 4 méthodes .

```
public class Service1 extends Service {
    @Override
    public void onCreate() {
    // Création du service ; config des ressources
    }

    @Override
    public IBinder onBind(Intent intent) {
    // Crée un canal de communication vers le service (si besoin)
    return null;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
    // Démarrage du service
    return START_STICKY;
    }

    @Override
    public void onDestroy() {
    // Arrêt du service ; libération des ressources
    }
}
```

Appelée à la création du service (première utilisation)

Appelée pour créer un canal entre client et service (connexion du client)

Appelée à chaque démarrage du service

Appelée par le système pour (nettoyer) le service (inactif depuis trop longtemps ou besoin de ressources)

La classe Service

La méthode onStartCommand :

```
@Override
public int onStartCommand(Intent intent, int flags, int startId) {
return START_STICKY;
}
```

L'intention à l'origine du démarrage du service

Info complémentaire lié au démarrage (0 = normal), nouvelle tentative, etc.

Un numéro identifiant la séquence de démarrage, permet de stopper la séquence avec *stopSelfResult(int startId)*

La valeur de retour permet de contrôler le mode de redémarrage :

- **START_STICKY** : le service est automatiquement redémarré s'il est tué, intent est alors à null.
- **START_NOT_STICKY** : logique, pas de redémarrage auto...
- **START_FLAG_RETRY** : pour signaler une nouvelle tentative de démarrage avorté (lorsque flag = START_FLAG_RETRY)

La classe Service

Déclarer le service dans le manifeste :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="net.marie.michel.service1">
                <service android:name=".Service1" />
</manifest>
```

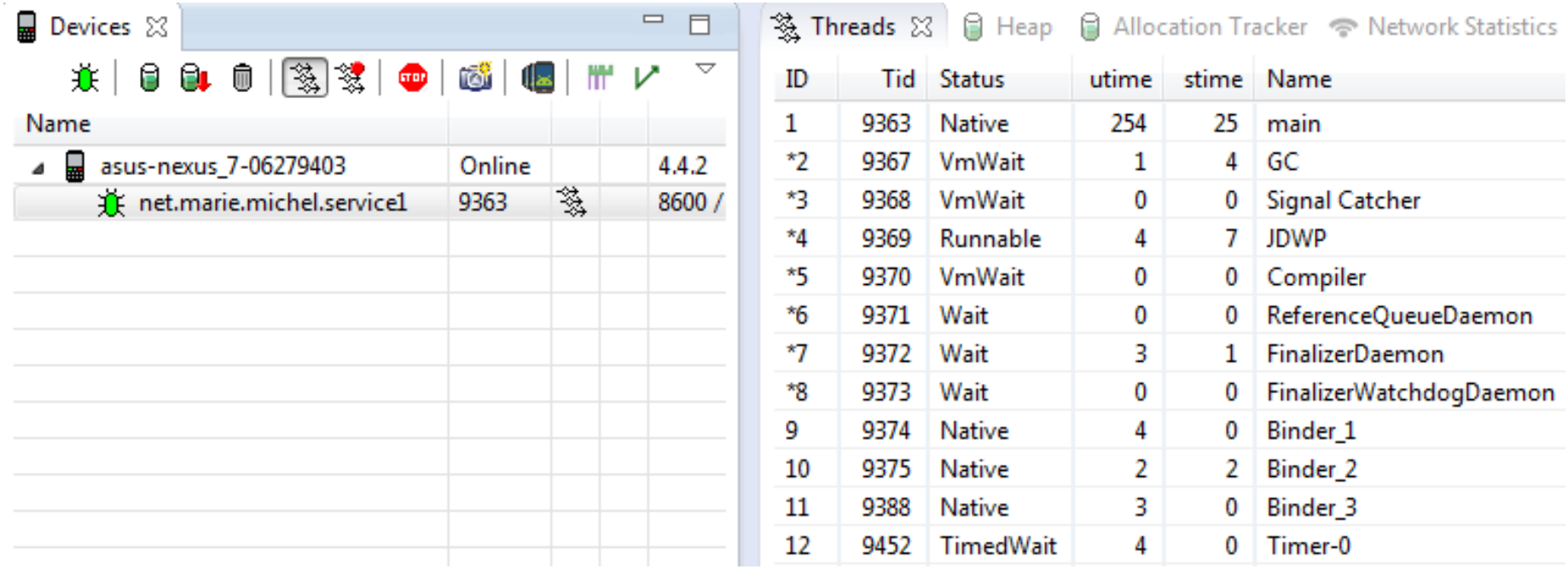
Démarrer/Arrêter le service :

```
Intent intent = new Intent(getApplicationContext(), Service1.class);
    startService(intent);
    …
    stopService(intent);
```

Ces méthodes sont à invoquer dans l'activité de l'application intégrant le service.
Le service est local à l'application, il va s'exécuter dans le thread principal de l'application.

La classe Service

L'onglet « Update Threads » de l'outils DDMS permet de visualiser les threads du processus unique de l'application de service :

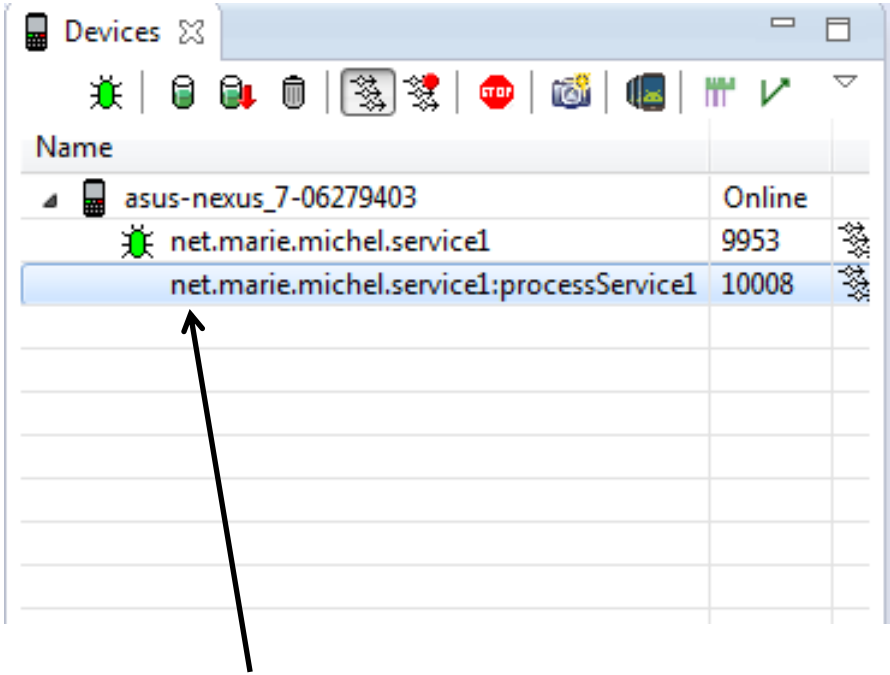


ID	Tid	Status	utime	stime	Name
1	9363	Native	254	25	main
*2	9367	VmWait	1	4	GC
*3	9368	VmWait	0	0	Signal Catcher
*4	9369	Runnable	4	7	JDWP
*5	9370	VmWait	0	0	Compiler
*6	9371	Wait	0	0	ReferenceQueueDaemon
*7	9372	Wait	3	1	FinalizerDaemon
*8	9373	Wait	0	0	FinalizerWatchdogDaemon
9	9374	Native	4	0	Binder_1
10	9375	Native	2	2	Binder_2
11	9388	Native	3	0	Binder_3
12	9452	TimedWait	4	0	Timer-0

Activer le service dans son propre processus

Il suffit de modifier le manifeste :

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        package="net.marie.michel.service1">
<service android:name=".Service1" android:process=":processService1"/>
</manifest>
```



Threads | Heap | Allocation Tracker | Network Statistics

ID	Tid	Status	utime	stime	Name
1	10008	Native	21	7	main
*2	10012	VmWait	0	0	GC
*3	10013	VmWait	0	0	Signal Catcher
*4	10014	Runnable	0	2	JDWP
*5	10015	VmWait	0	0	Compiler
*6	10016	Wait	0	0	ReferenceQueueDaemon
*7	10017	Wait	0	0	FinalizerDaemon
*8	10018	Wait	0	0	FinalizerWatchdogDaemon
9	10019	Native	1	0	Binder_1
10	10020	Native	0	1	Binder_2
11	10021	TimedWait	0	0	Timer-0

Petit jeu : tuer le processus de service et constater que l'activité est toujours active ; et qu'elle peut redémarrer un service.

L'accès au thread UI depuis le service…Aie !

Votre service veut accéder au thread UI pour afficher un message :

- **Votre service n'a pas défini de processus dédié, le code de *onStartCommand* n'est pas concurrent** : pas de problème car **onStartCommand()** s'exécute sur le thread principal de l'application donc sur le thread UI.
- **Votre service n'a pas défini de processus dédié, mais le code de *onStartCommand* est concurrent** :

OU

- **Votre service a défini un processus dédié** :

Ouh la la !! Il va falloir poster une demande de traitement dans la file d'attente du processus gérant le thread UI.

Exemple d'exécution périodique concurrente, depuis un processus dédié ou non.

Pour poster un message exécutable au processus parent en charge du threadUI

```
Timer timer = new Timer();
TimerTask task;


public int onStartCommand(Intent intent, int flags, int startId) {
   final Handler handler = new Handler();
   task = new TimerTask() {
     public void run() {
     handler.post(new Runnable() {
                  public void run() {
                  Toast.makeText(Service1.this, "Coucou : Service1 !",
                  Toast.LENGTH_SHORT).show();
                  }
         });
   }};
   timer.schedule(task, 0, 5000);
 return (flags==START_FLAG_RETRY)?START_FLAG_RETRY:START_STICKY;
}
```

Le bloc à faire exécuter par le threadUI

Démarrage automatique du service

Android, une fois le système démarré, diffuse un message (broadcast d'une intention) pour signaler « *c'est bon, vous pouvez démarrer vos applications de service* »…
Le message en question : `android.intent.action.BOOT_COMPLETED`

Pour démarrer automatiquement le service il suffit :

- de donner la permission `android.permission.RECEIVE_BOOT_COMPLETED`
- de capturer cette intention `BOOT_COMPLETED`,
- de lui associer une classe récepteur, classe **BroadcastReceiver**, en filtrant sur l'intention précédente,
- de surcharger la méthode **onReceive()**, méthode de la classe **BroadcastReceiver** déclenchée lorsque l'intention est reçue,
- de faire en sorte que la méthode **onReceive()** démarre le service.

Exemple : Capturer le message diffusé

Le Manifeste :

```
<uses-permission
      android:name="android.permission.RECEIVE_BOOT_COMPLETED" />
```

Récepteur d'intention (message Broadcast)

La classe en charge du message

```
<receiver android:name=".AutoStartService">
   <intent-filter>
     <action android:name="android.intent.action.BOOT_COMPLETED" />
   </intent-filter>
</receiver>
```

Le message accepté par le filtre

Le filtre à appliquer aux messages reçus

Exemple : La classe BroadcastReceiver

Classe BroadcastReceiver en charge du message

Méthode déclenchée à la réception du message

```
public class AutoStartService extends BroadcastReceiver
{
  public void onReceive(Context context, Intent intent)
  {
    Intent startServiceIntent =
            new Intent(context, Service1.class);
    context.startService(startServiceIntent);
  }
}
```

Démarrage du service Service1

Capture d'intention dans une activité

L'activité doit alors déclarer un filtre d'intention et installer un récepteur sur ce filtre en spécifiant l'objet qui prendra en charge son traitement.

- La classe **IntentFilter** permet la création d'un filtre.
- L'objet en charge du traitement de l'intention de type **BroadcastReceiver** sera un objet intégré à l'activité.
- La méthode **Activity.registerReceiver()** permet l'enregistrement du récepteur pour un filtre et un objet de traitement passés en paramètres.
- Cette configuration est réalisée dans la méthode **onCreate()** de l'activité.

Exemple

L'activité

```
public class MainActivity extends Activity {
private BroadcastReceiver ProgressReceiver;

@Override
protected void onCreate(Bundle savedInstanceState) {
        IntentFilter action = new
        IntentFilter("net.marie.michel.service.progressdownload");

        ProgressReceiver = new BroadcastReceiver() {
        @Override
        public void onReceive(Context context, Intent intent) {
                Bundle bundle = intent.getExtras();
                if (bundle != null) {
                        ?...?
                }
          }
        };
registerReceiver(ProgressReceiver, action);
}
}
```

L'intention diffusée à filtrer

L'objet intégré de traitement de l'intention

Le traitement à réaliser si l'intention est capturée…

Enregistrement du récepteur d'intention

Diffusion de l'intention

La diffusion de l'intention (broadcast) se fait simplement à l'aide de la méthode **sendBroadcast().**

```
public static final String NOTIFICATION_PROGRESS =
                        "net.marie.michel.service.progressdownload";

        Intent intent = new Intent(NOTIFICATION_PROGRESS);
        intent.putExtra("FILE_LENGTH", fileLength);
        intent.putExtra("TOTAL", total);
        intent.putExtra(RESULT, result);

        sendBroadcast(intent);
```

L'action de l'intention

Diffusion de l'intention

Configuration de l'intention

Communiquer entre Activité et Service

- L'activité peut être liée au service afin de permettre la communication entre celle-ci et le service au cours de son exécution.
- Pour ceci une solution consiste à :
  - définir une **interface** (contrat de communication entre activité et service, quelles méthodes sont autorisées),
  - utiliser une classe **Binder** qui fournit les outils de communication **RPC** entre activité et service,
  - implémenter la méthode **onBind()** de la classe de service qui retourne la référence à l'objet **Binder** (qui retourne **null** autrement).
- Cette solution n'est réalisable que si Activité et Service font partie du même processus, autrement il faudra utiliser une des deux solutions suivantes :
  - passer par un mécanisme de communication inter-processus avec définition des interfaces en **AIDL** (**Android IDL** pour ceux qui se souviennent avoir utilisé **COM** et **IDL**),
  - utiliser la classe **IntentService** beaucoup plus « cool », c'est ce que nous ferons plus tard…

Exemple côté Service : définir l'interface

Le service

L'interface définissant les méthodes qu'il expose au client, l'activité

```
public class Service2 extends Service {

// Méthodes accessibles du service
    public interface Service2Interface {
        public int getInfo();
        public Service2 getService();
    }
}
```

Les méthodes exposées

Exemple côté Service : la classe Binder

Le service

La classe Binder qui implémente l'interface précédemment définie et qui gère les échanges (c'est magique !)

```
public class Service2 extends Service {
 private int infoOfService2;

  public class Service2Binder extends Binder implements
      Service2Interface {
            public int getInfo() {
                  return infoOfService2;
            }
            public Service2 getService() {
                  return Service2.this;
            }
      }
}
```

Les méthodes imposées par l'interface

Exemple côté Service : la méthode onBind()

Le service

```
public class Service2 extends Service {

private Binder ib;

   public void onCreate() {
        ib = new Service2Binder();
        …
   }

   public IBinder onBind(Intent intent) {
        return ib;
   }
}
```

L'objet Binder, canal de communication avec le service

Création du Binder à la création du service

Méthode retournant le canal de communication à l'activité client

Communiquer entre Activité et Service

- Côté activité client, pour lier l'activité au service, un objet de la classe **ServiceConnection** doit être instancié. Il s'agit d'une classe « Callback » pour la gestion des événements de connexion/déconnexion entre l'activité et le service.
- C'est cet objet qui permettra de récupérer la référence à l'objet Binder, lien entre l'activité et le service.

- Deux méthodes de cet objet doivent être surchargées :
  - **onServiceConnected()** : appelée lorsque le service sera connecté à l'activité client et fournira la référence au Binder,
  - **onServiceDisconnected()** : qui permet de nettoyer (si nécessaire) les objets créés à la connexion.

Exemple côté Activité : l'objet ServiceConnection

L'activité

```
public class MainActivity extends Activity {
Service2Interface monService = null;

private ServiceConnection maConnexion = new
                              ServiceConnection() {

   public void onServiceConnected(ComponentName className,
                                  IBinder binder) {
       monService = (Service2.Service2Interface)binder;
    }
   public void onServiceDisconnected(ComponentName className)
    {
     monService = null;
    }
  };
}
```

La référence au service

L'objet callback

Méthode appelée à la connexion

Méthode appelée si déconnexion par un « crash » du service

Exemple côté Activité : utilisation du service

Connexion et démarrage du service

```
Intent intent = new Intent(getApplicationContext(),
                                        Service2.class);
bindService(intent, maConnexion, Context.BIND_AUTO_CREATE);
startService(intent);
```

Déconnexion et arrêt du service

```
Intent intent = new Intent(getApplicationContext(),
                                        Service2.class);
stopService(intent);
unbindService(maConnexion);
```

Appel de la méthode getInfo() du service

```
int infoFromService = monService.getInfo();
```

La classe IntentService

La classe de service à implémenter doit hériter de la classe **IntentService**. 2 méthodes seulement sont obligatoires :

- le constructeur de la classe qui doit simplement appeler le constructeur de la classe de base **IntentService** en lui passant un nom (celui du service…),
- la méthode surchargée **onHandleIntent()** qui reçoit les requêtes au service dans une file d'attente et les traite dans des Threads.

Le service est démarré à la réception d'une intention transmise par la méthode **startService(Intent)** et sera automatiquement stoppé lorsque les traitements de la file d'attente seront achevés.
La déclaration dans le manifeste se fait identiquement à la classe Service.

Exemple IntentService : Manifest

```
<manifest
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="net.marie.michel.service"
    android:versionCode="1"
    android:versionName="1.0" >
<application>
  <service android:name="net.marie.michel.service.DownloadService"
          android:exported="true">
          <intent-filter>
          <action
            android:name="net.marie.michel.service.downloadfile"/>
          </intent-filter>
     </service>
</application>
</manifest>
```

Classe IntentService de traitement du service

Utilisable depuis une autre application

L'action démarrant de service

Exemple IntentService : la classe de service

```
public class DownloadService extends IntentService {

    // Obligatoire pour IntentService
    public DownloadService() {
    super("DownloadService");
    }

    // Appel du service
    @Override
    protected void onHandleIntent(Intent intent) {
        String action = intent.getAction();
        String param1 = intent.getStringExtra(PARAM1_KEY);
        String param2 = intent.getStringExtra(PARAM2_KEY);

                etc…
    }
}
```

Le constructeur obligatoire

Le nom du service

Le traitement du service

Possibilité de plusieurs traitements en commutant selon l'action

Pause café ???