Développement d'applications nomades

(Cours 4)

Références utilisées pour ce cours

- `android.developper.com` :
  - Guide de développement
  - Référence
- Développement d'applications professionnelles avec `Android 2`, Reto Meier, Editions Pearson

Composants d'une application

Composants d'une application 1/2

- **Activité :**
  - chaque "écran" de l'application correspond à une activité (une instance d'une classe dérivée de la classe `Activity`)
  - une activité utilise des instances de classes dérivées de la classe `View` pour interagir avec l'utilisateur.
- **Service :**
  - un service (une instance d'une classe dérivée de la classe `Service`) correspond à une fonctionnalité "non visible" de l'application. Il peut être exécuté même lorsque l'application n'interagit pas avec l'utilisateur.

Une application peut

- être constituée de plusieurs activités/services
- ne contenir que des activités ou que des services
- faire appel aux activités/services d'une autre application

Composants d'une application 2/2

- **Intents et Intent Filters:**

  L'utilisation d'une activité ou d'un service se fait par une requête auprès du système `android`.

  - chaque activité ou service déclare les fonctionnalités pour lesquelles il est disponible (*intent filter*)
  - lorqu'une application désire utiliser une fonctionnalité
    - elle déclare son intention de le faire (*intent*)
    - le système `android` détermine et démarre l'activité/service correspondant.

- Broadcast Receivers, Content Providers, Widgets, Notification (non abordés dans ce cours)

Exemple

Un lecteur de MP3

- une activité qui permet de choisir un morceau, démarrer ou arréter la lecture
- un service qui gére la lecture du fichier MP3
- un intent pour démarrer le service, depuis l'activité ou depuis une autre application.

Déclarations

- chaque activité ou service d'une application doit impérativement être déclaré dans le fichier `AndroidManifest.xml`
- la déclaration de chaque activité/service contient la liste des *intents* auxquelles il doit répondre.

Exécution d'une application

- Les composants d'une application
  - sont à l'écoute des changements d'état de l'application et réagissent en conséquence.
  - doivent répondre rapidemment sous peine d'être interrompus
    - les opérations consommatrices de temps doivent être réalisées par des tâches de fond exécutées en parallèle
    - l'introduction du parallèlisme conduit à des comportements subtils qu'il faut maitriser.
- Par défaut, chaque application est exécutée par un processus distinct exécutant une instance distincte de la machine `Dalvik`.

Il est cependant possible, bien que peu courant, de spécifier un autre mode d'exécution.

Gestion des ressources

- les ressources sont gérées de manière agressive, l'application a peu de contrôle sur cet aspect et le système peut décider de tuer un processus en fonction de ses besoins.
- L'ordre dans lequel les processus sont tués est basé sur des priorités

priorité critique	**priorité haute**	**priorité basse**
Processus actif	Processus visible	Processus d'arrière plan
	Processus de service en cours d'exécution	Processus vide

- La définition de l'état d'un processus sera donnée par la suite.

Intent et Intent Filters

Intents and intent filters

Intents

- Les applications `android` sont isolées mais peuvent se solliciter mutuellement en utilisant des *intents*,
- utilisation des *intents*
  - en déclarant une intention de faire exécuter une action à une activité ou un service non connu (*late binding*)
  - en demandant de démarrer/lié explicitement un service ou une activité donné

Dans les deux cas, cela se fait en utilisant un *intent*, i.e. une instance de la classe `Intent`

```
Intent
```

La classe `Intent`

Une instance de la classe `Intent` contient, entre autre, les éléments suivants

- **composant :** le composant explicitement nommé
- **action :** l'action a effectuer
- **data :** les données sur lesquelles effectuer l'action

Remarque : Ces informations sont toutes facultatives mais peuvent être cumulées , le système `android` a la responsabilité de trouver, si possible, l'activité ou le service correspondant.

La classe `Intent`

D'autres informations sont disponibles

- **category :** type de composant, affine le filtrage.
- **extras :** paires (clé, valeur) pour des informations additionnelles
- **flags :** instructions pour le système concernant la gestion du composant sélectionné par le système

c.f. Intents and intent filters

action

Une action est représentée par une chaîne de caractère

- la classe `Intent` contient un certain nombre d'action prédéfinies, comme par exemple `Intent.ACTION_CALL` pour réaliser un appel.
- il est possible de créer de nouvelles actions, le nom doit alors être préfixé par le *package* de l'application

`"irad.m1.projet.MON_ACTION''`

data

La spécification du type de donnée auquel un *intent* est associé repose sur :

- les URI (wikipedia)
- les type MIME (wikipedia)

La classe URI de Java permet de manipuler des `URI`.

La classe `Intent`

- Les constructeurs donnés ci-dessous permettent de créer une instance de la classe `Intent` en fonction du type de résolution souhaité.
  - implicitement, en donnant l'action souhaité
  - explicitement, en donnant le nom de la classe
- Les méthodes `getAction` et `getData` permettent au service ou à l'activité concerné de récuperer les informations de l'`intent`.

```
Intent(String action)          Intent(String action, Uri uri)

Intent(Context packageContext, Class<?> cls)

Intent(String action, Uri uri, Context packageContext, Class<?> cls)

getAction()          getData()
```

- d'autres constructeurs sont disponibles
- les informations peuvent être ajoutées à posteriori

consultez la doc. de la classe `Intent`

Filters

- Pour pouvoir traiter un *intent*, une activité/service doit déclarer un *intent filter* spécifiant les caractéristiques (action, data et catégory) des intents auxquels une activité/service peut répondre.

  Rmq : pour nos *intents* personnalisés, on se contentera de la catégorie `android.intent.category.DEFAULT` que l'on utilisera systématiquement[a] (voir doc. pour les autres catégories)

- cette déclaration est faite dans le manifeste de l'application

[a] en l'absence de catégorie, le composant ne pourra pas être sélectionné

Exemple

```
<activity android:name=''MyService' ...  >'
   ...
   <intent-filter>
    <action android:name=''td.irad.Play'' />
    <category android:name=android.intent.category.DEFAU
      android:scheme=''http''/>
   </intent-filter>
   <intent-filter>
    <action android:name=''td.irad.MyAction1''>
    <action android:name=''td.irad.MyAction2''>
    <category android:name=android.intent.category.DEFAU
   ...
</activity>
```

Remarques

- une activité/un service peut contenir plusieurs *intent filters*
- plusieurs occurences des champs category et data peuvent être présents dans le même composant.
- pour être sélectionnée une activité/un service doit contenir au moins un *intent filter*
  - qui satisfait la contrainte sur le nom de composant ou l'action
  - dont tous les champs category sont satisfaits
- si le nom du composant est spécifié dans un *intent*, les autres champs ne sont pas pris en compte.
- si plusieurs activités correspondent à un *intent*, l'utilisateur devra en choisir une.

Activités

Cycle de vie d'une activité

Une activité peut un être dans un des états suivants

- **en cours d'exécution (resumed/running)**
- **suspendu (paused)**
- **arrêtée (stopped)**

Activité en cours d'exécution

L'activité est au premier plan et intéragit avec l'utilisateur

Activité suspendue

- une autre activitité est en cours d'exécution mais celle-ci est toujours visible. Son état mémoire est maintenue par le système et elle est toujours attachée au gestionnaire de fenêtre.
- une telle activité ne sera tuée qu'en cas de très faible niveau de ressources disponibles.

Activité arrêté

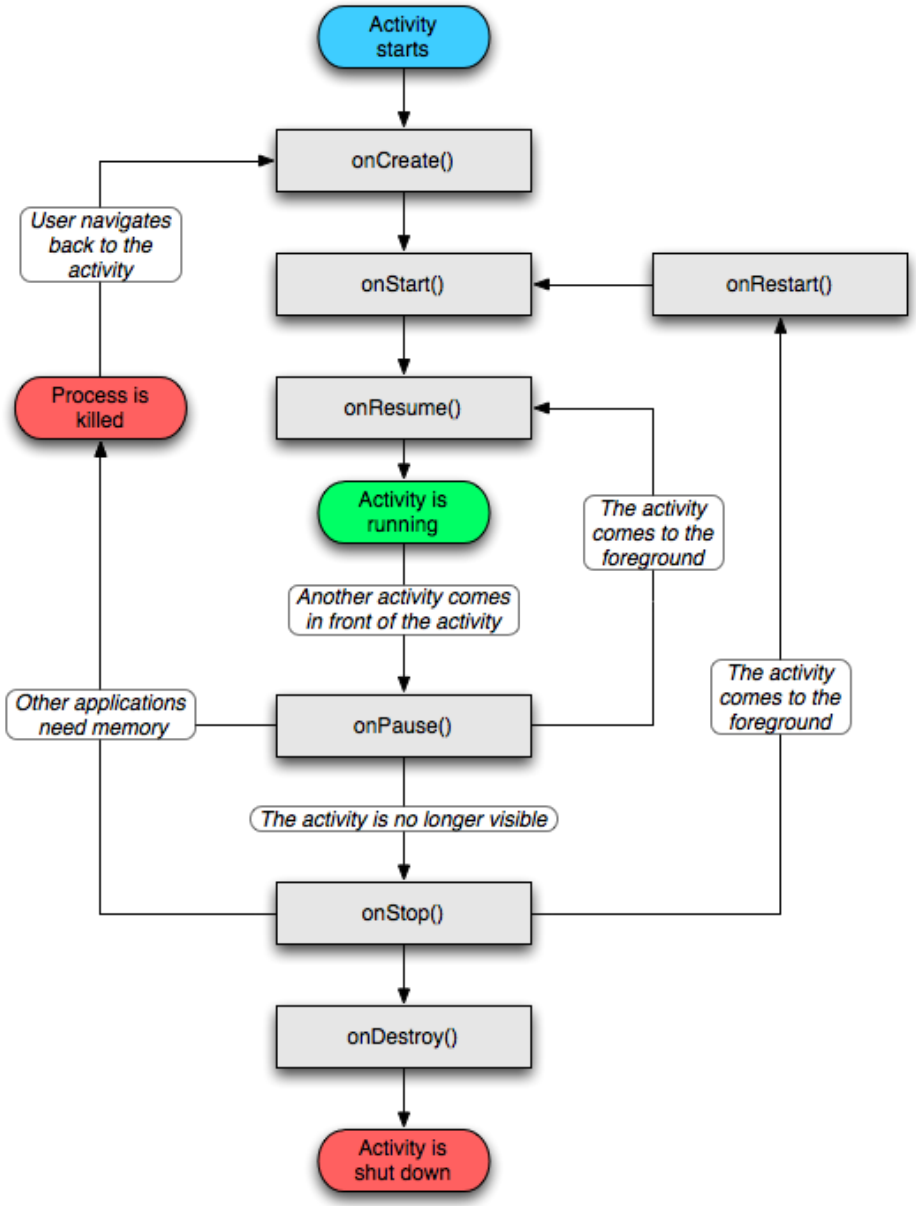
- l'activité est complétement masquée par une autre activité. Son état mémoire est maintenue par le système mais elle n'est plus attachée au gestionnaire de fenêtre.
- une telle activité peut être tuée pour libérer ses ressources.

La classe Activity

Pour définir une activité, il faut étendre la classe Activity et implanter les méthodes permettant de gérer son cycle de vie :

```
public class MyActivity extends Activity {

  // Methodes a implanter pour gerer le cycle de vie
  ////////////////////////////////////////////////

  //creation de l'activite
  public void onCreate(Bundle savedInstanceState);

  //l'activite est sur le point de devenir visible
  protected void onStart();

  // l'activite est visible
  protected void onResume();

  //une autre activite passe au premier plan
  protected void onPause();

  //l'activite n'est plus visible
  protected void onStop();

  // l'activite est sur le point d'etre detruite
  protected void onDestroy();
}
```

Activity



Exemple : Démarrer explicitement une activité

- Créer une nouvelle activité avec sa propre mise en page

```
public class NextActivity extends Activity {
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.next);
  }
}
```

- Créer un Intent pour lancer cette activité

```
Intent intent = new Intent(this,NextActivity.class);
if (intent !=null) this.startActivity(intent);
```

Rappel : l'activité doit être déclarée dans le manifeste.

Utilisation d'*intents* prédéfinis

ex : démarrer le navigateur

```
Intent intent = new Intent(Intent.ACTION_WEB_SEARCH,Uri.
    parse("http://www.google.fr"));
startActivity(intent);
```

Récupérer le résultat d'une activité

- Il est également possible de récupérer le résultat de l'exécution d'une (sous) activité
  - en utilisant la méthode `startActivityForResult` au lieu de `startActivity`
  - en utilisant la méthode `setResult` lorsque la sous activité termine
  - en implantant la *callback* `onActivityResult` de l'activité parent

```
public void startActivityForResult (Intent intent, int requestCode)

public final void setResult (int resultCode)

protected void onActivityResult (int requestCode, int resultCode, Intent data)
```

Démarrer une application depuis l'écran Application

Le système démarre une application en utilisant également les intents, pour qu'une application puisse être démarrée depuis l'écran Applications, une de ses activités doit déclarer l'*intent filter* suivant

```
<intent-filter>
    <action android:name =
        "android.intent.action.MAIN"/>
    <category android:name =
        "android.intent.category.LAUNCHER"/>
</intent-filter>
```

Services

Services

- Les Services correspondent à des composants dédiés à la prise en charge de fonctionnalités devant être exécutées de façon invisible, sans interface utilisateurs.
- Un service peut s'exécuter :
  - en mode **lié** *(bound)*
    - interface client pour les composants qui lui sont liés.
    - Lorsque plus aucun composant n'est lié il est détruit.
  - en mode **non lié** *(unbound)*
    - il ne fournit généralement pas de résultat au composant qui l'a crée.
    - il peut s'exécuter indéfiniment en **arrière plan** [a] même si le composant qui l'a démarré est détruit.

[a] attention à cette notion, on parle ici du point de vue du système et non de l'application, contrairement aux threads que nous verrons par la suite.

Attention : dans les deux cas, les services s'exécutent ...

... dans le **thread principal** de l'application!!!

source : `developper.android.com`

> *A services runs in the same process as the application in which it is declared and in the main thread of that application, by default. So, if your service performs intensive or blocking operations while the user interacts with an activity from the same application, the service will slow down activity performance. To avoid impacting application performance, you should start a new thread inside the service.*

La classe Service

Pour définir un service, il faut étendre la classe Service et implanter les méthodes permettant de gérer son cycle de vie :

```
// Commun aux deux modes
//////////////////////////

public void onCreate();     // Creation du service
public void onDestroy(); // Destruction du service

// mode non lie
//////////////////////

// Demarrage du service
public int onStartCommand(Intent intent,
                int flags, int startId);

//mode lie
//////////

// Liaison au service
public IBinder onBind(Intent intent);

// Liberation du service
public boolean onUnbind(Intent intent);

// re-liaison ???
public void onRebind(Intent intent);
```

N.B :

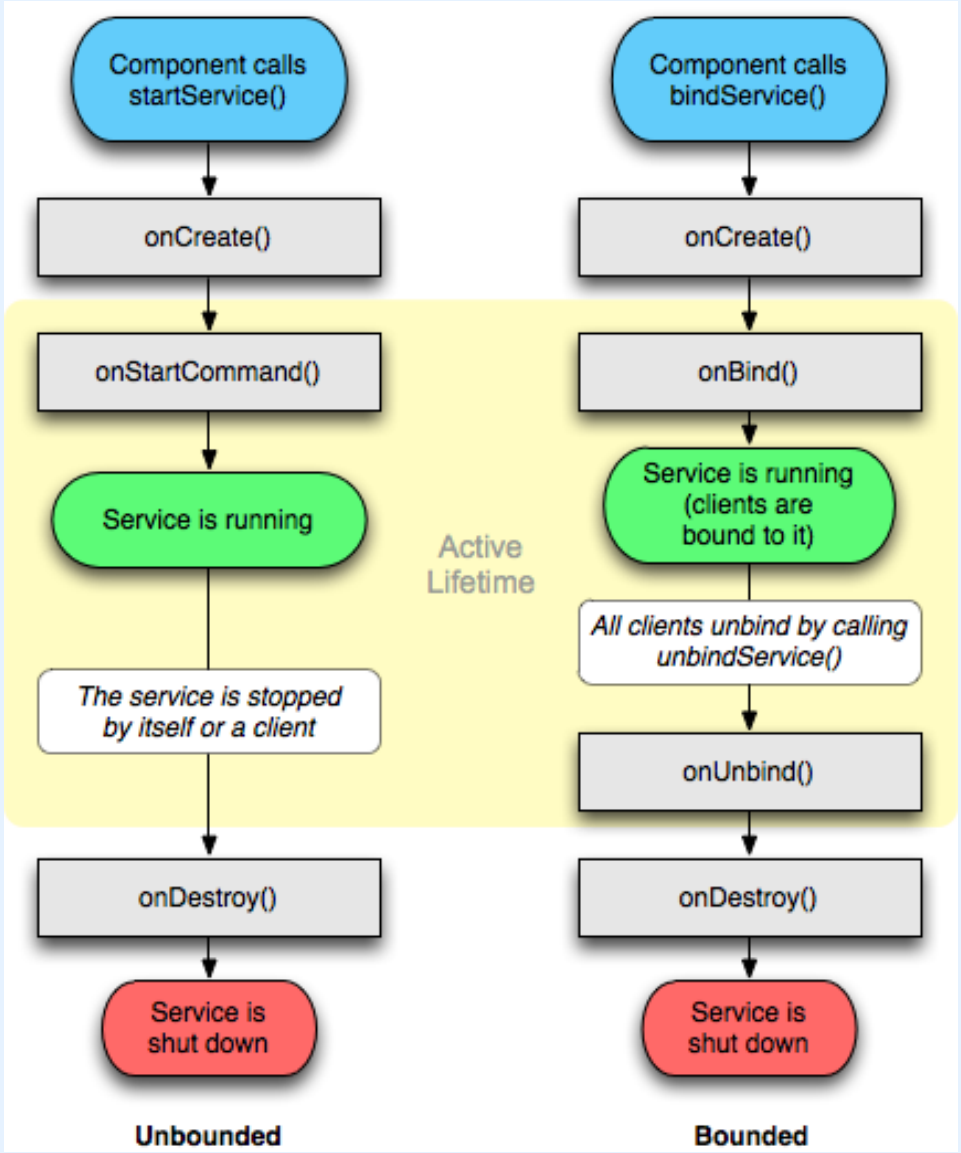
un même service peut offrir les deux modes de

Méthodes d'accès aux services

- `startService()` : démarrage en mode non lié
- `bindService()` : liaison au service
- `unbindService()` : destruction du lien vers le service

La liaison à un service provoque la création de celui-ci s'il n'existe pas déja.

Cycle de vie d'un service (source developpers.android.com)



Mode non lié

Un composant

- démarre un service en mode non lié en appelant la méthode `startService`
- arrête un service en mode non lié en appelant la méthode `stopService`

```
public class MyActivity{
  ...
  //in a given method
  ...
  startService(new Intent(this, MyService.class));
  ...
  ...
  stopService(new Intent(this, MyService.class));
}

public class MyService{
  public int onStartCommand(Intent intent, int flags, int
      startId){
    // do some work
    // run in application main thread
  }
}
```

Mode non lié

Un service peut demander lui même son arrêt en appelant la méthode `stopSelf()` [a]

```
public class MyService{
  ...
  //in some method
  ...
  stopSelf()
  ...
}
```

[a] Utile pour améliorer les performances car un service a peu de chance d'être tué.

Mode lié

On distingue deux cas, selon que

1. le service est local à l'application,

   dans ce cas il est possible d'accéder "directement" à l'objet rendant le service

2. le service peut être utilisé par une autre application,

   dans ce cas il est nécessaire d'utiliser un mécanisme de passage de message [a]

Dans les deux cas, l'interface avec le service passe par un objet dont la classe dérive de l'interface `IBinder`.

[a] chaque application à son propre *process*

Liaison à un service

Coté service : on implante la méthode `onBind` qui

- renvoie un objet servant d'interface entre le client/service
- sera utilisée par le système pour établir la liaison

La nature de cet objet dépend du type de liaison, il peut

- encapsuler le service
- permettre le passage de message

Liens

`IBinder onBind(Intent intent)` `IBinder`

Liaison à un service

Coté client : le client appelle la méthode `bindService` qui réalise la connection par l'intermédiaire d'un objet de la classe `ServiceConnection` dont il faut implanter les méthodes

- `onServiceConnected`, exécutée au moment de la liaison
- `onServiceDisconnected`, exécutée au moment de la libération

- `boolean bindService (Intent intent, ServiceConnection conn, int flags)`
- `ServiceConnection`
  - `void onServiceConnected(ComponentName className, IBinder service)`
  - `void onServiceDisconnected(ComponentName className)`

Liaison à un service local : exemple

On définit une liaison triviale à un objet local

- le *service* n'a aucune fonctionnalité
- le *binder* permet simplement de récupèrer un pointeur sur l'objet

L'exemple est trivial, en pratique

- définir une classe encapsulant le service
- faire retourner une instance de cette classe au *binder*

Coté service

```
public class MyBoundedService extends Service {

  // Creation du binder

  private final IBinder mBinder = new LocalBinder();

  // La classe LocalBinder qui derive de la classe Binder
  // (implantation par defaut de IBinder)

  public class LocalBinder extends Binder{

    // On propose simplement une methode qui retourne
    // le service lui-meme

    MyBoundedService getService(){

      return MyBoundedService.this;
    }
  }

  // on retourne le binder au client

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

Binder

Coté client

```
public class BoundedServiceBasicActivity extends Activity {

  MyBoundedService mService;
  boolean mBound = false;
  private ServiceConnection mConnection =
  new ServiceConnection()
  {
    public void onServiceConnected(ComponentName className,
    IBinder service){
      LocalBinder binder = (LocalBinder) service;
      mService = binder.getService();
      mBound = true;
    }

    public void onServiceDisconnected(ComponentName className)
        {
      mBound=false;
    }
  };

  protected void onStart(){ /* next slide */}

  protected void onStop(){ /* next slide */}
}
```

Coté client

```
protected void onStart(){
  super.onStart();
  Intent intent = new Intent(this,MyBoundedService.class);
  bindService(intent, mConnection,
  Context.BIND_AUTO_CREATE);
}

protected void onStop(){
  if (mBound){ unbindService(mConnection);mBound=false;}
}
```

Liaison à un service "distant"

- la classe `Messenger` propose des *binders* permettant le passage de messages.
  - coté client :
    1. génèrer une instance de `Messenger` à partir du paramètre de type `IBinder` de la méthode `onServiceConnected`.
    2. envoyer des messages en utilisant la méthode `send` de cette instance.
  - coté service :
    1. génèrer une instance de `Messenger` à partir d'une instance d'une classe qui dérive de `Handler` et permet de traiter les messages.
    2. faire retourner par la méthode `onBind` du service le *binder* donné par un appel à la méthode `getBinder` de cette instance.

```
Messenger

Messenger(IBinder binder)          public void send (Message message)

Messenger(Handler target)          public IBinder getBinder ()
```

Message

- un message contient (entre autre) une description et un objet arbitraire.
- utiliser la méthode statique `obtain` pour créer un nouveau message (voir documentation).

Handler

- implanter la méthode `handleMessage` pour traiter les messages reçus.

```
Message        public static Message obtain ()

Handler        handleMessage(Message msg)
```

IntentService

- Dans un service le travail sera généralement réalisé par un nouveau thread pour ne pas bloquer le thread principal (dans lequel s'exécute le service).
- S'il n'est pas nécessaire que le service traite plusieurs requêtes simultanément, on pourra étendre la classe `IntentService` qui gérera les requêtes les unes après les autres.

c.f. : Services/Extending the IntentService Class

Remarque

- de manière générale, on évitera de démarrer une activité ou un service depuis une des *callbacks* de gestion du cycle de vie d'une activité ou d'un service.
- cette pratique est source d'erreur (voir exemple suivant et expérimentation en T.D).

MyActivity.class

```
public void onCreate(Bundle savedInstanceState) {
  ...
  Log.i(this.getClass().getName(), "step 1");
  startService(new Intent(this, MyService1.class));
  startService(new Intent(this, MyService2.class));
  Log.i(this.getClass().getName(), "step 2");
}
```

MyService1.class

```
public int onStartCommand(Intent intent, int flags, int
    startId){
  Log.i(this.getClass().getName(), "step 3");
}
```

MyService2.class

```
public int onStartCommand(Intent intent, int flags, int
    startId){
  Log.i(this.getClass().getName(), "step 4");
}
```

Exécution

irad.td.MyActivity	step 1
irad.td.MyActivity	step 2
irad.td.MyService1	step 3
irad.td.MyService2	step 4

Retour sur les priorités

processus actif (ou de premier plan)

Un processus est actif s'il intéragit avec l'utilisateur, i.e il contient

- une activité avec laquelle l'utilisateur interagit (méthode `onResume` appelée)
- un service lié à une activié avec laquelle l'utilisateur intéragit
- un service s'exécutant au premier plan (méthode `startForegroun`
- un service exécutant une de ses méthodes de gestion de son cycle de vie (`onCreate`, `onStart` ou `onDestroy`)
- un Broadcast Receiver exécutant sa méthode `onReceive`

En général, peu de processus sont actifs en même temps. Ces processus ne sont tués qu'en dernier recours.

processus visible

Un processus est visible s'il contient des composants qui ne sont pas au premier plan mais peuventt affecter ce que voit l'utilisateur, i.e il contient

- une activité visible, i.e. qui n'est pas au premier plan mais qui toutefois visible par l'utilisateur (sa méthode `onPause` a été appelée). Par exemple, une activité qui affiche une boite de dialogue.
- un service lié à une activité visible.It hosts a Service that's bound to a visible (or foreground) activity.

Un processus visible ne sera tué que si cela est nécessaire pour maintenir la réactivité d'un processus actif.

processus de service en cours d'exécution

Un processus de service est un processus hébergeant des services en cours d'exécution (la méthode `startService` a été appelée) et qui n'appartient pas à l'une des catégories précédentes.

- bien que ces processus ne contiennent pas de composant visible par l'utilisateur, ils effectuent généralement des tâches importantes pour ce dernier (e.g. lecture de musique).
- ils ne seront tués que pour assurer la réactivité d'un processus actif ou visible.

processus d'arrière plan

Un processus d'arrière plan est un processus qui contient une activité qui n'est pas visible par l'utilisateur (la méthode `onStop` a été appelée).

- Ces processus n'ont pas d'impact sur l'utilisateur et le système peut réclamer leurs ressources.
- ils sont généralement nombreux et ordonnés par le système en fonction de la durée de leur inactivité (le plus ancien est le premier tué)

processus vide

- les processus vides sont tous les processus qui ne tombent dans aucune des catégories précédentes.
- ils ne sont gardés en mémoire que pour accélerer leur redémarrage.

Contraintes supplémentaires

- Si deux processus ont la même priorité, celui s'étant exécuté le plus longtemps à une priorité plus basse.
- Si une application A dépend d'un service ou d'un content provider fournit par l'application B, B recevra une priorité au moins aussi haute que A.

Une application reste en mémoire et s'éxécute tant que le système n'a pas besoin de réclamer ses ressources.

Tâches en arrière plan

Tâches en arrière plan

- Une application `android` doit être réactive, sous peine d'être interrompues.
  - Les tâches lourdes doivent être placées en arrière plan.
  - Deux solutions sont possibles,
    - les tâches asynchrones
    - les threads.

Attention

Dans tous les cas, l'exécution de tâches en arrière plan nécessitent une synchronisation avec l'interface.

Tâches asynchrones

- Les tâches asynchrones permettent de s'abstraire des problèmes de synchronisation en offrant des mécanismes permettant de mettre à jour l'interface
  - au cours de la progression de la tâche
  - lorsqu'elles terminent
- Les threads sont à utiliser uniquement lorsque l'expressivité des tâches asynchrones est insuffisante.

La classe `AsyncTask`

- Paramètres de généricité
  - Le type des paramètres d'entrées donnés à la tâche
  - le type de la progression, publiés pendant l'exécution en tâche de fond.
  - le type du résultat de l'exécution en tâche de fond
- méthodes à redéfinir
  - `doInBackground` : prend en entrée un nombre variable de paramètres de type `Input` et s'exécute dans le thread d'arrière plan. Elle ne doit pas tenter d'interagir avec l'interface.
  - `onProgressUpdate` : permet de signaler dans l'interface la progression du thread. Synchronisé avec l'interface.
  - `onPostExecute` : reçoit en paramètre la valeur de retour de la méthode `doInBackground`. Synchronisé avec l'interface.

```
AsyncTask<Params, Progress, Result>

protected abstract Result doInBackground (Params ... params)

protected void onProgressUpdate (Progress ... values)

protected void onPostExecute (Result result)
```

Exemple

```
public class MyTask extends AsyncTask<Context,Void,Void> {

  Context context; // Lien vers l'activity
  int cpt = 0; // compteur de taches

  @Override
  protected Void doInBackground(Context... input) {
    context = input[0];
    while(...){
      // execute une tache et incremente le compteur
      publishProgress();}
    return null;
  }

  @Override
  protected void onProgressUpdate(Void... values){
    super.onProgressUpdate(values);
    Toast.makeText(context, "Task␣"+cpt+"␣Done", Toast.
        LENGTH_SHORT).show();
  }

  @Override
  protected void onPostExecute(Void result){
    super.onPostExecute(result);
    toast = Toast.makeText(context, "All␣tasks␣done", Toast.
        LENGTH_SHORT).show();
  }
}
```

Threads

- L'utilisation des **Tâches asynchrones** permet un accès simplifier à l'interface utilisateur depuis une tâche s'exécutant à l'arrière plan.
- Toutefois, celles-ci ne permettent pas toujours une interaction suffisament fine et il est alors nécessaire d'utilliser directement les **threads** Java et de les synchroniser avec l'interface utilisateur.

Création de threads

Les threads proposés dans `android` sont les threads standards du langage `java` (`java.lang.thread`).

Synchronisation avec l'interface

Pour utiliser des threads dans un environnment disposant d'une interface utilisateur, il est nécessaire de les synchroniser avec le thread principal de l'application. Ceci peut se faire de deux manières :

- en forçant le thread a s'exécuter dans le même thread que l'interface.

```
public final void runOnUiThread (Runnable action)
```

- en utilisant un `Handler` sur le thread principal

Utilisation d'un Handler sur le thread principal

Un handler permet de poster des messages et des tâches (Runnable) dans la file de messages du thread dans lequel il a été crée qui aura alors la responsabilité de les traiter.

```
// Dans le thread principal
private Handler handler = new Handler();

private Runnable updater = new Runnable (){
  public void run(){update()}}

public void update(){
  // Mise a jour de l'interface
}

// Depuis le thread
handler.post(updater);
```

Handler

public final boolean post (Runnable r)