Développement Android (4.3)

Composants Logiciels

BENJAMIN BILLET

WARNING

Le contenu de cette présentation est basé sur la documentation anglophone officielle d'Android, diffusée sous licence *Creative Commons Attribution 2.5* :

developer.android.com

La plupart des schémas qui composent ce cours proviennent de cette documentation et sont, par conséquent, soumis à cette même licence.

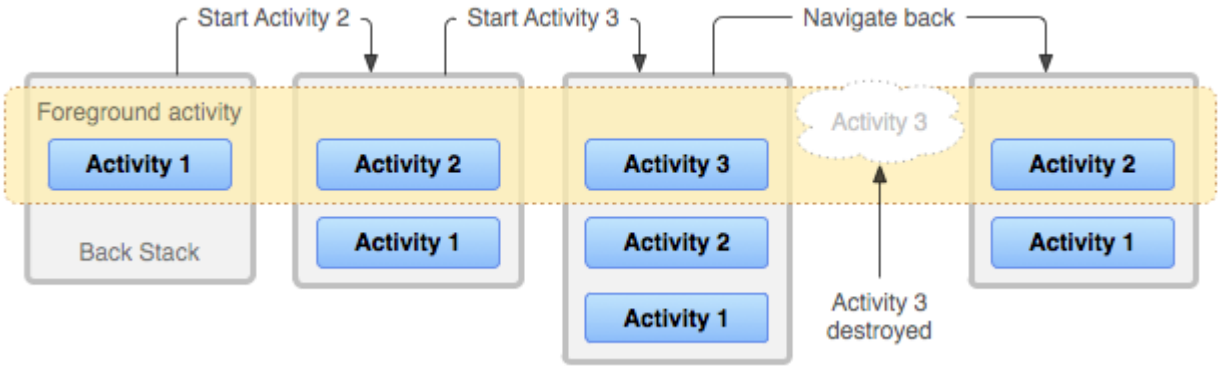
http://creativecommons.org/licenses/by/2.5/

Au Sommaire !

- Activités et Cycle de vie
- Intent Messaging
- Services
- Broadcast Receiver
- Runtime Reconfiguration

ACTIVITÉS ET CYCLE DE VIE

Le composant Activity



- L'activité est le composant principal, qui fournit un écran avec lequel l'utilisateur intéragit.
- Une application est composée d'activités, qui s'appellent les unes les autres.
- L'une des activités est dite "main activity" et est dévoilée à l'utilisateur lorsqu'il ouvre l'application.

Le composant Activity

```
public class HelloActivity extends Activity
{
    // Called when the activity is first created.
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
    }
}
```

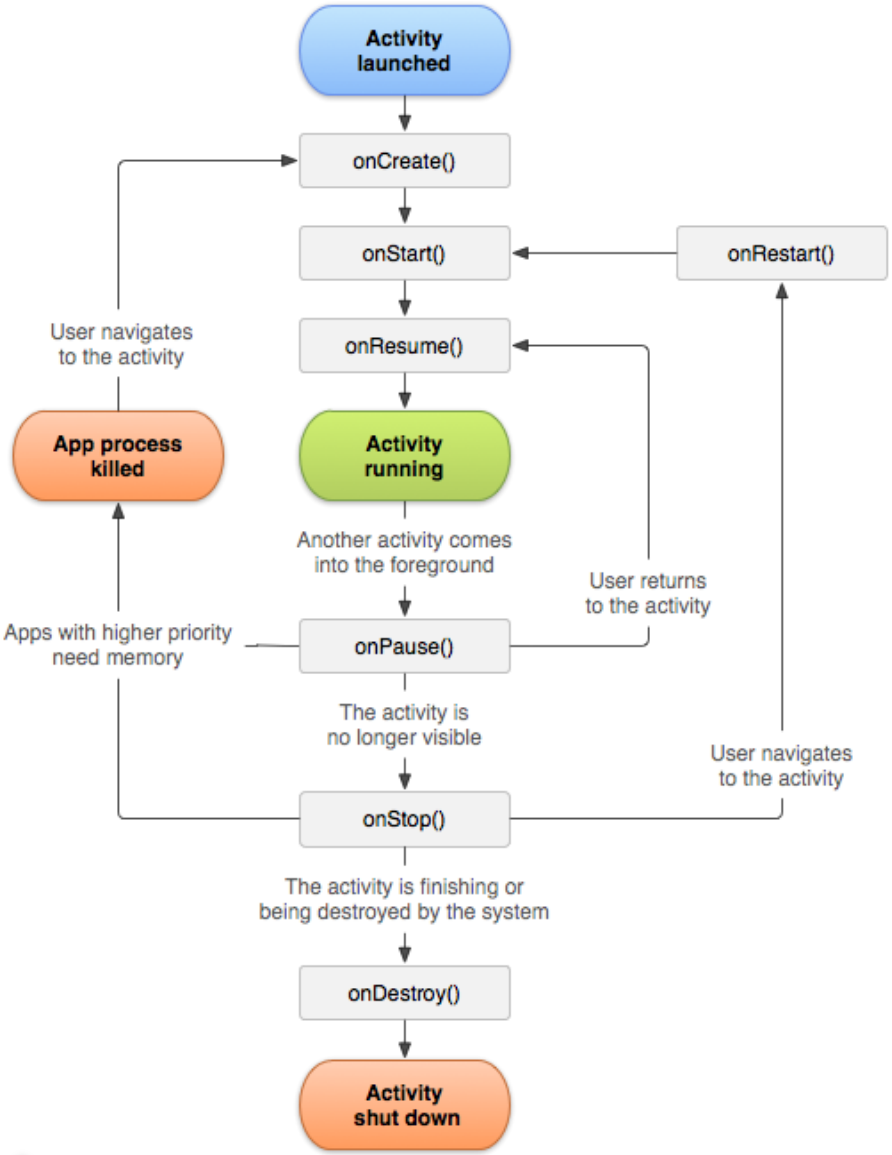
- Une activité étend la classe Activity.
- Lorsque l'activité est instanciée, onCreate() est invoquée.

Le composant Activity

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

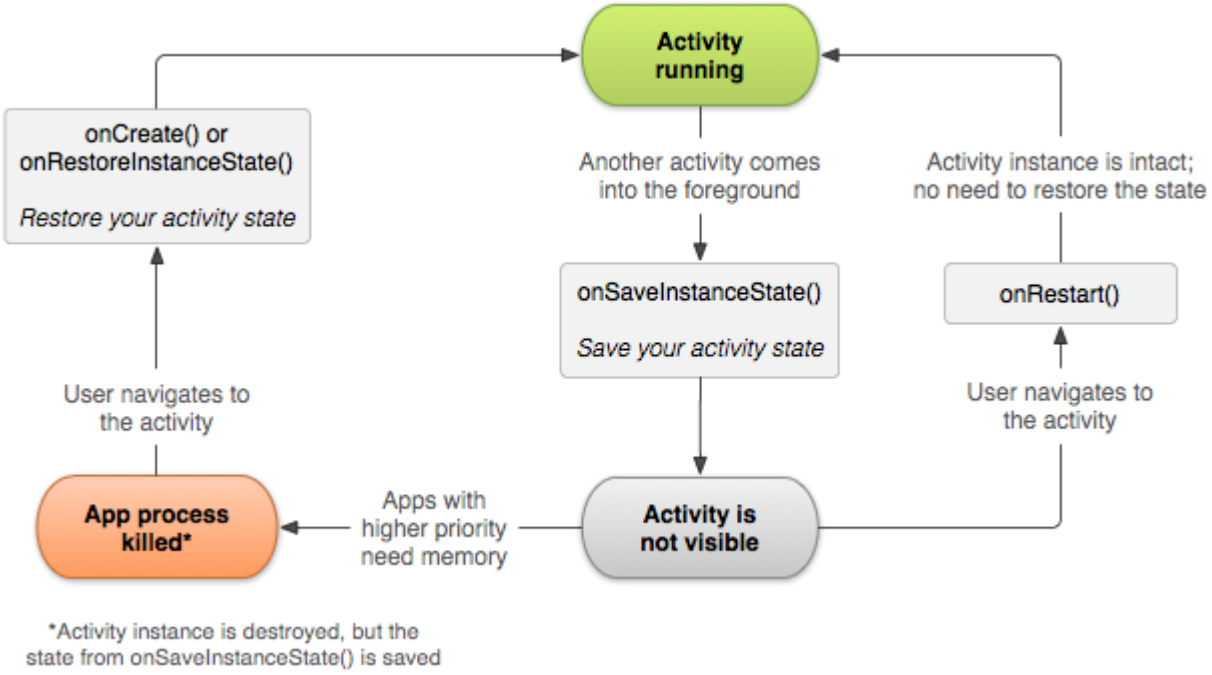
- Une activité déclarée dans le manifest est un point d'entrée potentiel.
- D'autres options permettent de configurer l'icône de l'application, son nom, etc.
- Les informations fournies dans intent-filter la désignent, ici, comme "main activity".

Le composant Activity



- Durée de vie totale : de onCreate() à onDestroy()
- Arrière plan : de onStart() à onStop(), l'application s'exécute en fond et n'est plus visible par l'utilisateur.
- Premier plan : de onResume() à onPause(), l'application est au premier-plan et l'utilisateur intéragit avec celle-ci.

LE COMPOSANT ACTIVITY



- Une activité effectue régulièrement des aller-retours entre le premier plan et l'arrière plan.
- Une activité en arrière plan peut être détruite à tout instant pour libérer la mémoire.

Le composant Activity

```
public class HelloActivity extends Activity
{
    @Override
    protected void onSaveInstanceState(Bundle state)
    {
        state.putString("key1", "myString");
        state.putIntArray("key2", new int[] { 0, 1, 2, 3 });
        state.putSerializable("key3", ...);

        super.onSaveInstanceState(state);
    }

    @Override
    protected void onCreate(Bundle state) // or onRestoreInstanceState(Bundle state)
    {
        super.onCreate(state);
        if (state == null)
            ...
        else
            ...
    }
}
```

- Remarque : si l'utilisateur ferme l'application avec "Back", onSaveInstanceState() n'est pas invoquée.

Changement d'activité

Le comportement d'Android est, sauf cas particulier, déterministe lorsqu'une activité A invoque une activité B :

1. A.onPause()
2. B.onCreate()
3. B.onStart()
4. B.onResume()
5. A.onStop()

INTENT MESSAGING

Intent Messaging

- L'échange de message entre les composants applicatifs se fait au travers de messages, appelés Intent.
- Un Intent est une structure de données qui décrit soit une opération à effectuer, soit un évènement.

ComponentName	La classe et le package d'un composant.
Action	Une action qui doit être, ou a été, réalisée.
Data	Une URI (tel:, http:, geo:, …) vers des données à traiter, et un type MIME.
Category	Une catégorie de composant.
Extras	Des paramètres (clé/valeur).
Flags	Options diverses.

Intent Messaging

- Android route les Intent d'un composant à un autre en exploitant les informations fournies dans le manifest.
- Un Intent qui définit un ComponentName est directement transmis (intent explicite).
- Les autres Intent (implicites) nécessitent de trouver les composants qui matchent les différents champs, avec trois issues possibles :
  - Un seul composant a été trouvé, le message est transmis.
  - Plusieurs composants ont été trouvé, une boîte de dialogue demande alors à l'utilisateur de faire un choix.
  - Aucun composant n'a été trouvé, une exception est levée.

Un simple Intent explicite

ActivityA

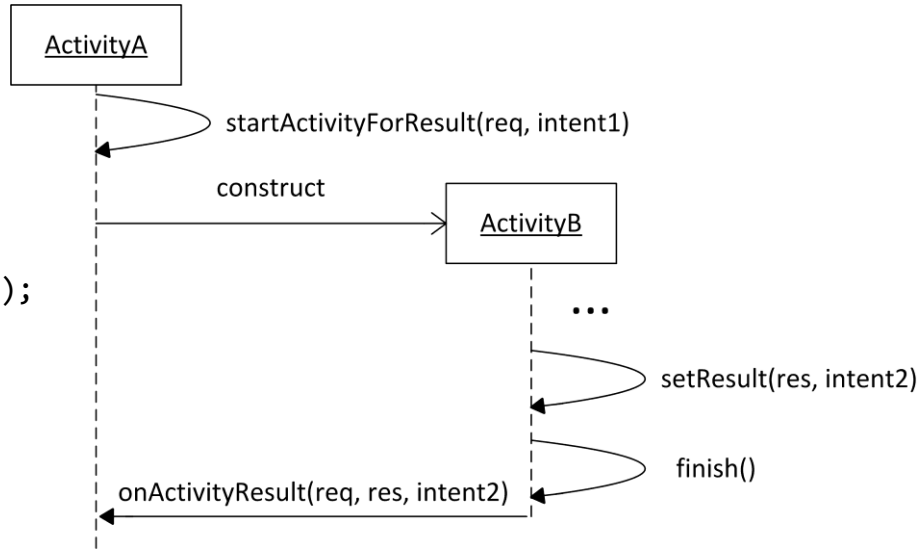
```
Intent i = new Intent(this, ActivityB.class);
i.putExtra("something", 10);
startActivityForResult(i, 42);

@Override
public void onActivityResult(int requestCode, int resultCode, Intent data)
{
    super.onActivityResult(requestCode, resultCode, data);
    if(requestCode == 42 && resultCode == Activity.RESULT_OK)
        int val = data.getExtras().getInt("somethingPlusOne");
}
```

ActivityB

```
Intent request = getIntent();
if(request.hasExtra("something"))
{
    int val = request.getIntExtra("something");
    Intent response = new Intent();
    response.putExtra("somethingPlusOne", val + 1);
    setResult(Activity.RESULT_OK, response);
}
else
    setResult(1); // error

finish();
```



Intent Resolution

- Les champs action, data (URI, type MIME) et category sont utilisés pour le matching des composants avec l'intent implicite.
  - De plus, il est possible de définir le package, comme pour un intent explicite.
- Chaque composant déclare des filtres dans son manifest (intent-filters).
- Chaque filtre est évalué indépendamment au moyen de trois tests.
- Un composant est sélectionné si l'un de ses filtres passe tous les tests.

Action test

Component

Intent

	No action	Set of actions A
No action	Failure	Success
Set of actions B	Failure	Success if A and B shares at least one element : $\|A \cap B\| \geq 1$

Category test

Component

Intent

	No category	Set of categories A
No category	Success	Success
Set of categories B	Failure	Success if every category of A is defined in B : $A \subseteq B$

- Android considère que tous les Intents implicites passés à startActivity() appartiennent à une catégorie “default”…
- … il faut donc l’ajouter dans le manifest si l’on désire que l’activité puisse recevoir les intents…
- … sauf pour les activités qui définissent l’action “main” ou la catégorie “launcher”.

DATA TEST

Component

Intent	No URI, No MIME	URI, No MIME	No URI, MIME	URI, MIME
No URI, No MIME	Success	Failure	Failure	Failure
URI, No MIME	Failure	URI match	Failure	Failure
No URI, MIME	Failure	Failure	MIME match	Failure
URI, MIME	Failure	Failure	Success si l'URI de l'Intent est content: ou file:	URI match, MIME match

- Les URI sont de la forme scheme://host:port/path.
- Le matching d'URI se fait sur la base des URIs définies par le composant (e.g., un composant avec URI="http" matche un Intent avec URI="http://inria.fr").
- Le matching de type MIME est exact, mais "*" peut être utilisé dans le sous-type (e.g., audio/*).
- Le matching réussit si le composant possède au moins une correspondance.

Les intent-filters

```
<intent-filter ... >
    <action android:name="com.example.project.SHOW_CURRENT" />
    <action android:name="com.example.project.SHOW_RECENT" />
    <action android:name="com.example.project.SHOW_PENDING" />
    ...
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    ...
    <data android:mimeType="video/mpeg" android:scheme="http" ... />
    <data android:mimeType="audio/mpeg" android:scheme="http" ... />
    ...
</intent-filter>
<intent-filter ... >
    ...
</intent-filter>
<intent-filter ... >
    ...
</intent-filter>
```

- Un Intent doit matcher au moins un filtre.
- Les étiquettes de catégorie/action prédéfinies incluent le nom de package complet :

Intent.CATEGORY_DEFAULT = android.intent.category.DEFAULT

Quelques Intent implicites

```
// exemple 1
Intent i = new Intent(Intent.ACTION_VIEW, Uri.parse("http://inria.fr"));
startActivity(i);

// exemple 2
Intent i = new Intent();
i.setAction(Intent.ACTION_CALL);
i.setData(Uri.parse("tel:0102030405"));
startActivity(i);

// exemple 3
Intent i = new Intent(Intent.ACTION_PICK, ContactsContract.Contacts.CONTENT_URI);
startActivityForResult(i, 42);

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
    super.onActivityResult(requestCode, resultCode, data);
    if (requestCode == 42 && resultCode == Activity.RESULT_OK)
    {
        Uri contactData = data.getData();
        ...
    }
}
```

SERVICES

Le composant Service

- Un service effectue des opérations en arrière plan (pas d'IHM).
- Les services sont partagés entre les applications, qui peuvent les invoquer à tout moment avec un Intent.
- Un service de type "started" possède son propre cycle de vie, indépendamment de toute application. Il doit être stoppé explicitement.
- Un service de type "bound" qui est détruit lorsque plus aucune application ne l'utilise.
- Un service peut être à la fois "started" et "bound".

Le composant Service

```
<manifest ... >
    <application ... >
        <service android:name=".MyPublicService" />
        <service android:name=".MyPrivateService" android:exported="false" />
        ...
    </application ... >
    ...
</manifest >
```

- Un service doit être déclaré dans le manifest.
- Un service est public par défaut.
- Tous les services étendent la classe Service.
- Sauf implémentation contraire, les services s'exécutent dans le thread courant.

Un simple started service

- La classe IntentService fournit une implémentation basique pour la classe Service, avec comme particularités :
  - Un nouveau thread est crée pour l'exécution.
  - Une file d'attente est mise en place pour traiter les requêtes simultanées.
  - Ferme le service lorsque les requêtes ont été traitées.
- Une seule méthode à implémenter : onHandleIntent()

Un simple started service

```
public class MyService extends IntentService
{
    public MyService()
    {
        super("MyService");
    }

    // The IntentService calls this method from the default worker thread with
    // the intent that started the service
    @Override
    protected void onHandleIntent(Intent intent)
    {
        ...
    }
}

// Invoke the service using an Intent
Intent intent = new Intent(this, MyService.class);
startService(intent);
```

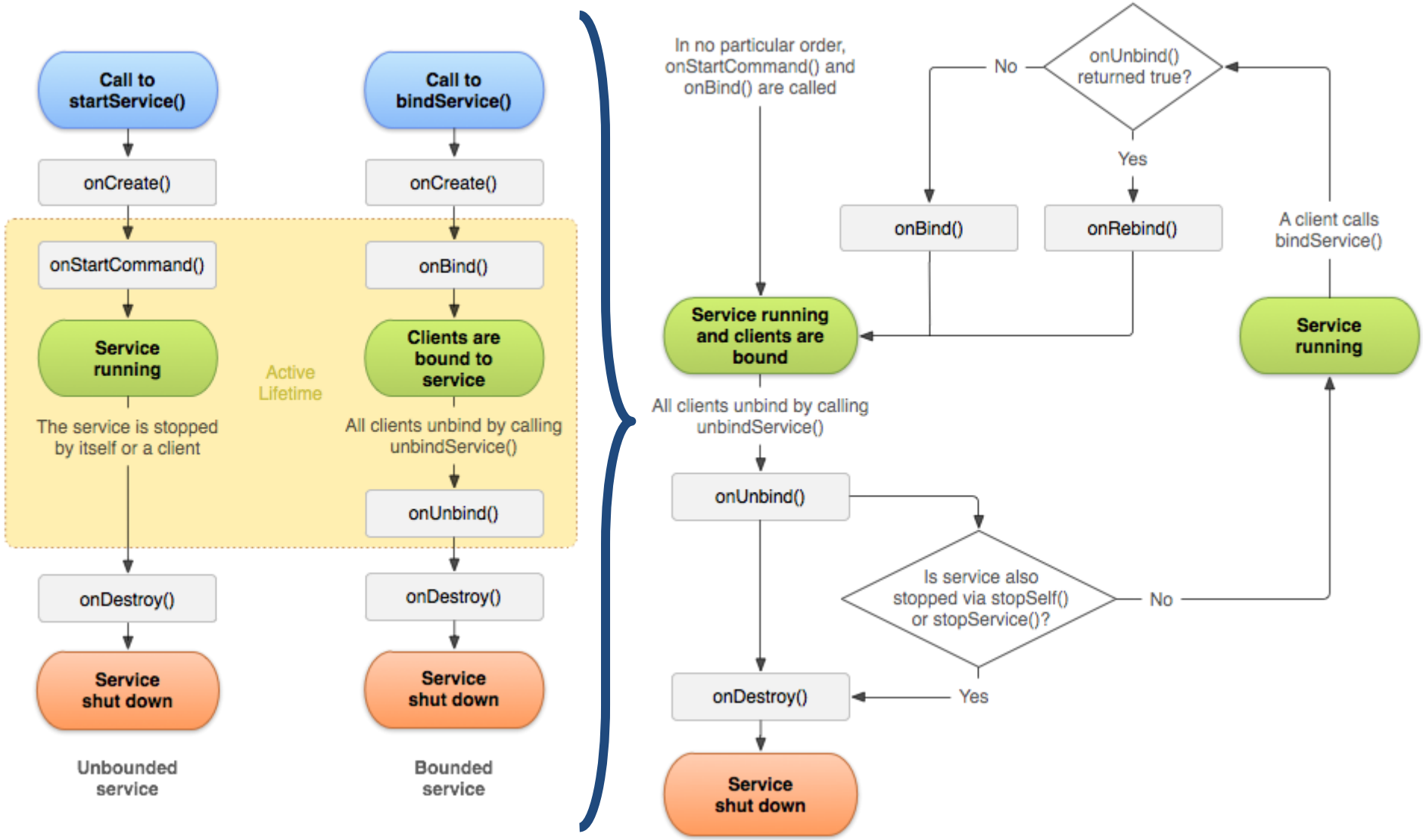
Le composant Service

- Deux méthodes importantes :
  - onStartCommand() : invoquée quand le service est démarré avec startService()
  - onBind() : invoquée quand un composant se lie au service avec bindService

	onBind() returns null	onBind() returns a binder
onStartCommand() is not implemented	Nonsense	Pure bound service
onStartCommand() is implemented	Pure started service	Both

- onBind() retourne un objet de type Binder

Cycle de vie d'un service



La classe IntentService, décortiquée

```
public class IntentService extends Service
{
   private Looper looper;
   private ServiceHandler handler;

   private class ServiceHandler extends Handler
   {
      public ServiceHandler(Looper looper)
      {
         super(looper);
      }

      @Override
      public void handleMessage(Message msg)
      {
         onHandleIntent((Intent)msg.obj);
         stopSelf(msg.arg1);
      }
   }

   @Override
   public void onCreate()
   {
      super.onCreate();
      HandlerThread thread = new HandlerThread("thread");
      thread.start();

      looper = thread.getLooper();
      handler = new ServiceHandler(looper);
   }

   @Override
   public int onStartCommand(Intent i, int flg, int id)
   {
      Message msg = handler.obtainMessage();
      msg.arg1 = id;
      msg.obj = i;
      handler.sendMessage(msg);
      return START_STICKY;
   }

   @Override
   public void onDestroy()
   {
      looper.quit();
   }

   @Override
   public IBinder onBind(Intent i)
   {
      return null;
   }

   protected abstract void onHandleIntent(Intent i);
}
```

La classe IntentService, décortiquée

```
public class IntentService extends Service
{
   private Looper looper;
   private ServiceHandler handler;

   private class ServiceHandler extends Handler
   {
      public ServiceHandler(Looper looper)
      {
         super(looper);
      }

      @Override
      public void handleMessage(Message msg)
      {
         onHandleIntent((Intent)msg.obj);
         stopSelf(msg.arg1);
      }
   }

   @Override
   public void onCreate()
   {
      super.onCreate();
      HandlerThread thread = new HandlerThread("t");
      thread.start();

      looper = thread.getLooper();
      handler = new ServiceHandler(looper);
   }
```

- Un premier aperçu du multithreading : le looper s'exécute dans le thread dédié à la réception de message.
- Chaque message est échangé entre le thread maintenant la queue et le thread courant exécutant le service.
- Comme promis, le service doit être stoppé explicitement, ici grâce à stopSelf.

La classe IntentService, décortiquée

Si le système tue le service pour des raisons de mémoire, alors :

- START_NOT_STICKY : le service n'est pas redémarré automatiquement sauf s'il reste des Intent en attente.
- START_STICKY : le service est redémarré automatiquement, onStartCommand() est réinvoqué, mais avec un Intent null.
- START_REDELIVER_INTENT : le service est redémarré automatiquement et la dernière instance qui n'a pas été explicitement terminée par stopSelf() ou stopService() est relancée.

```
    @Override
    public int onStartCommand(Intent i, int flg, int id)
    {
       Message msg = handler.obtainMessage();
       msg.arg1 = id;
       msg.obj = i;
       handler.sendMessage(msg);
       return START_NOT_STICKY;
    }

    @Override
    public void onDestroy()
    {
       looper.quit();
    }

    @Override
    public IBinder onBind(Intent i)
    {
       return null;
    }

    protected abstract void onHandleIntent(Intent i);
}
```

Un simple bound service

```
public class LocalService extends Service
{
    private final IBinder binder = new LocalBinder();
    public class LocalBinder extends Binder
    {
        LocalService getService()
        {
            return LocalService.this;
        }
    }

    @Override
    public IBinder onBind(Intent intent)
    {
        return binder;
    }
}
```

- Client et service doivent appartenir au même processus (pas de sérialisation automatique pour LocalBinder).
- Les services partagés entre plusieurs processus doivent utiliser des mécanismes d'échange de message.
- Attention à l'état partagé !

INVOCATION DU BOUND SERVICE

```
private LocalService service;
private boolean isBound = false;
private ServiceConnection conn = new ServiceConnection()
{
     public void onServiceConnected(ComponentName name, IBinder binder)
     {
          service = ((LocalBinder)binder).getService();
          isBound = true;
     }
     public void onServiceDisconnected(ComponentName name)
     {
          isBound = false;
     }
};


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

protected void onStop()
{
     super.onStop();
     if(isBound)
     {
          unbindService(conn);
          isBound = false;
     }
}
```

BROADCAST RECEIVER

Le composant BroadcastReceiver

- Réagis aux notifications qui sont envoyées par le système ou les autres applications (mise en veille, batterie faible, écouteurs connectés, etc.).
- Ne possèdent pas d'interface graphique, comme les services.
- Ne sont pas dédiés à de longues tâches et sont détruit une fois le traitement effectué.

Le composant BroadcastReceiver

- Deux types de broadcasts :
  - Normal broadcasts : l'ensemble des BroadcastReceiver compatibles est exécuté, sans ordre précis.
  - Ordered broadcasts : chaque receiver est invoqué l'un après l'autre et peut (i) propager un résultat au suivant ou (ii) stopper la propagation du broadcast.

Un simple receiver

```
public class MyReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        ...
    }
}
```

```
<receiver android:name="MyReceiver" >
    <intent-filter>
        <action android:name="android.intent.action.HEADSET_PLUG" />
    </intent-filter>
</receiver>
```

```
<receiver android:name="MyReceiver" android:exported="false" >
    <intent-filter>
        <action android:name="android.intent.action.HEADSET_PLUG" />
    </intent-filter>
</receiver>
```

Le cas asynchrone

- Un receiver est détruit à la fin de onReceive()
  - Pas d'asynchrone !
  - Pas de bound service !
- L'API 11 (Android 3.0) introduit la méthode goAsync(), qui retourne un objet PendingResult.
- Le receiver est alors maintenu jusqu'à l'appel de PendingResult.finish().

Créer son propre receiver

```
public class MyReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        ...
    }
}
```

```
<receiver android:name="MyReceiver" >
    <intent-filter>
        <action android:name="fr.inria.action.A_WONDERFUL_EVENT" />
    </intent-filter>
</receiver>
```

```
Intent intent = new Intent();
intent.setAction("fr.inria.action.A_WONDERFUL_EVENT");
sendBroadcast(intent);
```

Créer son propre receiver

```
<manifest>
<permission android:name="fr.inria.INRIA_PERMISSION"  />
...
<application>
    <receiver android:name="MyReceiver" android:permission="fr.inria.INRIA_PERMISSION">
        <intent-filter>
            <action android:name="fr.inria.action.A_WONDERFUL_EVENT" />
        </intent-filter>
    </receiver>
</application>
...
</manifest>
```

Le composant LocalBroadcastManager

- Le broadcast local n'est pas transmis au delà de l'application.
- Le broadcast local ne peut pas être émis par d'autres applications.
- Plus efficace qu'un broadcast global.
- Pas besoin d'être défini dans le manifest.

Le composant LocalBroadcastManager

```
public class ActivityA extends Activity
{
  @Override
  protected void onResume()
  {
    super.onResume();
    LocalBroadcastManager.getInstance(this).registerReceiver(receiver, new IntentFilter("my-event"));
  }

  private BroadcastReceiver receiver = new BroadcastReceiver()
  {
    @Override
    public void onReceive(Context context, Intent intent)
    {
      String message = intent.getStringExtra("message");
    }
  };

  @Override
  protected void onPause ()
  {
    LocalBroadcastManager.getInstance(this).unregisterReceiver(mMessageReceiver);
    super.onPause();
  }
}
```

Le composant LocalBroadcastManager

```
public class ActivityB extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        Intent intent = new Intent("my-event");
        intent.putExtra("message", "This is my message!");
        LocalBroadcastManager.getInstance(this).sendBroadcast(intent);
    }
}
```

Le cas ordered

- Chaque receiver reçoit le broadcast dans un ordre défini (priorité), ou dans un ordre aléatoire si priorité identique.
- Chaque receiver peut modifier les informations contenues dans l'Intent avant de le retransmettre, ou mettre fin à la diffusion en invoquant abortBroadcast().

Le cas ordered

```
public class CallReceiver extends BroadcastReceiver
{
    @Override
    public void onReceive(Context context, Intent intent)
    {
        super.onCreate(savedInstanceState);
        String number = getResultData();
        if (number == null)
            number = intent.getStringExtra(Intent.EXTRA_PHONE_NUMBER);

        if (shouldCancel(number))
            setResultData(null);
        else
            setResultData(doSomething(number));
    }
}
```

```
<intent-filter android:priority="100" >
    <action android:name="android.intent.action.NEW_OUTGOING_CALL" />
</intent-filter>
```

- Les priorités négatives sont réservées au système.
- Une bonne pratique consiste à vérifier si un résultat existe, pour ne pas écraser les précédentes transformations.

Le cas ordered

```
public class ActivityA extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState)
        Intent intent = new Intent("fr.inria.INRIA_ACTION");
        sendOrderedBroadcast(intent, "fr.inria.INRIA_PERMISSION");
    }
}
```

```
void sendOrderedBroadcast(

     Intent intent,
     String receiverPermission,           // required permission
     BroadcastReceiver resultReceiver,    // final receiver of the broadcast
     Handler scheduler,                   // execution context of the result receiver callback
     int initialCode,                     // the result code, usually Activity.RESULT_OK
     String initialData,
     Bundle initialExtras

)
```

Le cas sticky

- Tout broadcast (ordered ou non-ordered) peut être sticky.
- Un broadcast sticky est mémorisé par le système.
- Le broadcast sticky peut être consulté à tout moment, sans utiliser un receiver.
- Un sticky broadcast est reçu par tout les receivers compatibles, même si ceux-ci sont enregistrés après l'émission.
- Un sticky broadcast peut être écrasé en émettant un broadcast équivalent (même action/data), peut importe son type (ordered ou non). Par exemple, le niveau de batterie.

Le cas sticky

```
public class ActivityA extends Activity
{
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState)
        Intent intent = new Intent("fr.inria.INRIA_ACTION");

        sendStickyBroadcast(intent);

        sendStickyOrderedBroadcast(intent, null, null, Activity.RESULT_OK, null, null);

        // get the sticky broadcast without receiver
        // if more than one sticky broadcast match the filter, the first is returned
        IntentFilter f = new IntentFilter("fr.inria.INRIA_ACTION");
        Intent i = registerReceiver(null, f);

         // remove
        removeStickyBroadcast(intent);
    }
}
```

Le cas sticky

```
// register for the battery changed event
IntentFilter filter = new IntentFilter(Intent.ACTION_BATTERY_CHANGED);

// sticky so using null as receiver works fine
// return value contains the status
Intent batteryStatus = this.registerReceiver(null, filter);

int status = batteryStatus.getIntExtra(BatteryManager.EXTRA_STATUS, -1);
boolean isCharging = (status == BatteryManager.BATTERY_STATUS_CHARGING
  || status == BatteryManager.BATTERY_STATUS_FULL);

boolean isFull = (status == BatteryManager.BATTERY_STATUS_FULL);

int chargePlug = batteryStatus.getIntExtra(BatteryManager.EXTRA_PLUGGED, -1);
boolean usbCharge = (chargePlug == BatteryManager.BATTERY_PLUGGED_USB);
boolean acCharge = (chargePlug == BatteryManager.BATTERY_PLUGGED_AC);
```

RUNTIME CHANGES

Runtime changes

- Les propriétés d'un device peuvent changer pendant l'exécution (changement d'orientation, changement de langue, etc.).
- Lorsqu'un de ces changements se produit, Android reconstruit l'activité en cours en invoquant onDestroy() puis onCreate().
- L'utilisation de onSaveInstanceState() et onRestoreInstanceState() est recommandée, mais peut conduire à des ralentissements (sérialisation / désérialisation, réouverture de connexions, etc.).

Sauvegarde d'un etat en mémo

```
public class MyActivity extends Activity
{
    @Override
    public Object onRetainNonConfigurationInsta
    {
        final MyDataObject data = collectM
        return data;
    }

    @Override
    protected void onCreate(                    anceState)
    {
        super.onCreate(                         e);
        final MyData                 DataObject)getLastNonConfigurationInstance();
        if(data ==
            data                    ata();
    }
}
```

- sauvegarder d'objet possédant un te (activité, service, view, etc.) car ceux-ci eraient maintenu en mémoire (memory leak).

DEPRECATED, TO CHANGE

PRISE EN CHARGE MANUELLE

```
<activity android:name=".MyActivity"
          android:configChanges="orientation|keyboardHidden|screenSize|...">
</activity>
```

```
public class MyActivity extends Activity
{
    @Override
    public void onConfigurationChanged (Configuration config)
    {
        super.onConfigurationChanged(config);
        if(config.orientation == Configuration.ORIENTATION_LANDSCAPE)
            ...
        else if(config.orientation == Configuration.ORIENTATION_PORTRAIT)
            ...
    }
}
```

- Android ne reconstruit pas l'application et laisse le développeur gérer le problème.
- Fastidieux en pratique.

PRISE EN CHARGE MANUELLE

```
<activity android:name=".MyActivity"
          android:configChanges="orientation|keyboardHidden|screenSize|...">
</activity>
```

```
public class MyActivity extends Activity
{
    @Override
    public void onConfigurationChanged (Configuration config)
    {
        super.onConfigurationChanged(config);
        if(config.orientation == Configuration.ORIENTATION_LANDSCAPE)
            ...
        else if(config.orientation == Configuration.ORIENTATION_PORTRAIT)
            ...
    }
}
```

- Android ne reconstruit pas l'application et laisse le développeur gérer le problème.
- Fastidieux en pratique.