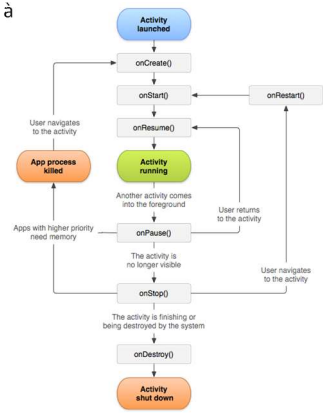
Android – Passage de paramètres et stockage de données

Emmanuel Conchon
(emmanuel.conchon@unilim.fr)

1

Rappels sur les activités (Activity)

- Une activité fonctionne suivant une machine à états
  - 5 états
    - Starting: en train de démarrer
    - Running: en fonctionnement
    - Paused: Obscurcie ou hors champ mais pas arrêtée
    - Stopped: Arrêtée mais toujours en mémoire
    - Destroyed: Eteinte et plus en mémoire
  - On passe d'un état à un autre à l'aide de transitions représentées par des évènements
    - On peut réagir à ces évènements dans le code dans des méthodes dédiée
      - ★ *onCreate(), on Pause(), onResume(), onStop(), onDestroy()…*



2

2

Rappels sur les Intents

- Une activité est atteignable à l'aide d'un point d'entrée (Intent)
  - Elle peut en avoir plusieurs
- Un Intent est un objet message utilisé pour demander une action à un composant Android comme une activité par exemple
  - Une activité peut en déclencher une autre à l'aide d'un Intent explicite en fournissant le nom de l'activité à lancer (le nom de la classe)

```
Intent intent = new Intent(this, ActivityName.class);
startActivity(intent);
```

  - Elle peut également lui fournir des informations par le biais des Intent grâce aux « extra data »

```
Intent intent = new Intent(this, ActivityName.class);
intent.putExtra("name", value);
intent.putExtra("name", value);
startActivity(intent);
```

3

3

Le passage de paramètres

- Les « extra data » sont des associations entre une clé et une valeur un peu à la manière des dictionnaires
- Elles servent à la fois pour envoyer des données de l'activité A vers l'activité B mais également à récupérer les données coté B

```
public class ActiviteB extends Activity {
...
    public void onCreate(Bundle savedState) {
        super.onCreate(savedState);
        setContentView(R.layout.activity_b);
        Intent intent = getIntent();
        String extra = intent.getStringExtra("name");
        ...
    }
}
```

4

4

Le retour d'information

- Si une activité a besoin de récupérer un retour de l'activité qu'elle appelle on va plutôt utiliser la méthode
  - startActivityForResult(…) : Prend en paramètre l'Intent et une constante entière qui permet ensuite d'identifier le résultat retourné
- Le retour se gère à l'aide d'une méthode dédiée qui est appelée automatiquement onActivityResult(…) qui prend trois arguments
  - Le code spécifié lors de l'appel
  - Un code de résultat qui peut prendre soit la valeur RESULT_OK soit RESULT_CANCELED
  - L'Intent qui va contenir les données retournées

Le retour d'information

- Dans l'activité B il est nécessaire de spécifier qu'un retour aura lieu
  - On doit créer un nouvel Intent pour contenir les données à retourner
  - On fait ensuite appel à la méthode setResult() pour spécifier le code d'erreur et l'intent à renvoyer
  - Enfin on fait appel à la méthode finish() pour déclencher le retour proprement dit

```
public class ActiviteB extends Activity {
...
    public void myReturnMethod() {
        Intent intent = new Intent();
        intent.putExtra("name", value);
        setResult(RESULT_OK, intent);
        finish();
    }
}
```


Le retour d'information - Exemple

- Dans la classe appelante:
  - Appel: `startActivityForResult(myIntent,1);`
  - Traitement du retour:

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

    if (requestCode == 1) {
        if(resultCode == RESULT_OK){
            String result=data.getStringExtra("MyResult");
            Toast toast = Toast.makeText(this, result, Toast.LENGTH_SHORT);
            toast.show();
        }
        if (resultCode == RESULT_CANCELED) {
            Log.e(TAG, "Impossible de recuperer les donnees de B");
        }
    }
}
```

- Dans la classe appelée:

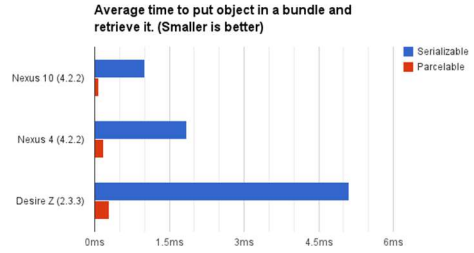
```
public void sendDataBack(View view){
    Intent returnIntent = new Intent();
    returnIntent.putExtra("MyResult","Retour de B");
    setResult(RESULT_OK,returnIntent);
    finish();
}
```


Le passage d'Objets

- Le mécanisme pour passer un objet d'une activité A à une activité B repose sur le même principe mais quelques règles doivent être respectées
- Les applications fonctionnent dans des processus séparés
  - La mémoire n'est donc pas partagée
  - Impossible de simplement fournir une référence vers l'objet
  - Il faut donc transformer l'objet en mémoire en un format transportable/stockable
    - Marshalling / Unmarshalling
- Pour effectuer l'opération de transformation, la solution classique en Java consiste à sérialiser l'objet
  - Solution toujours disponible en Android même si on lui préférera l'opération Parcelable qui est plus performante

Le passage d'Objets

- Comparaison de Serializable & Parcelable



Source : http://www.developerphil.com/parcelable-vs-serializable/

Le passage d'Objets

- Un objet parcelable doit implémenter l'interface **Parcelable** et fournir quelques méthodes
  - Une méthode **describeContents()** dont vous n'avez la plupart du temps pas besoin
    - Laisser la méthode auto-générée

```
@Override
public int describeContents() {
    return 0;
}
```

  - Une méthode **writeToParcel(Parcel out, int flags)** qui permet d' « écrire » l'objet à l'aide du **Parcel** fourni
  - Une constante **CREATOR** permettant de définir la façon dont on charge un objet pendant l'opération de dé-sérialisation
    - Cette constante est obligatoire
    - Elle signalise avec un objet **Parcelable.Creator<MyParcelable>()** dont on redéfinit en ligne deux méthodes
      - ★ *createFromParcel() qui prend en paramètre un Parcel pour le chargement*
      - ★ *newArray(int size) qui renvoie un tableau contenant les objets sérialisés*
  - Enfin, il faut définir un constructeur privé prenant en paramètre un **Parcel**

Le passage d'Objets

- Prenons l'exemple d'un objet simple composé
  - D'une valeur
  - D'un nom
- Cet objet va devoir transiter entre deux activités
  - L'activité principale A qui construit l'objet et l'envoie à la B
  - La B qui affiche le contenu

```
public class SimpleObject {
    protected int value;
    protected String name;

    public SimpleObject(int value,String name){
        this.value = value;
        this.name = name;
    }

    public int getValue(){
        return value;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setValue(int value) {
        this.value = value;
    }
}
```


Le passage d'Objets

- La version Parcelable

```
public class SimpleObjectParcelable extends SimpleObject implements Parcelable{
    public SimpleObjectParcelable(int value,String name){
        super(value, name);
    }

    private SimpleObjectParcelable(Parcel in) {
        super(in.readInt(), in.readString());
    }

    @Override
    public int describeContents() { return 0; }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeInt(this.value);
        dest.writeString(this.name);
    }

    public static final Parcelable.Creator<SimpleObjectParcelable> CREATOR =
            new Parcelable.Creator<SimpleObjectParcelable>() {
        public SimpleObjectParcelable createFromParcel(Parcel in) {
            return new SimpleObjectParcelable(in);
        }

        public SimpleObjectParcelable[] newArray(int size) {
            return new SimpleObjectParcelable[size];
        }
    };
}
```

Le passage d'Objets

- Les deux activités s'échangeant l'objet Parcelable
  - Activité principale dotée d'un bouton déclenchant sendObject()

```
public class ActiviteA extends AppCompatActivity {

    //private SimpleObjectParcelable obj;
    private SimpleObjectParcelable obj;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_activite);
        this.obj = new SimpleObjectParcelable(1,"obj1");
    }

    public void sendObject(View view) {
        Intent myIntent = new Intent(ActiviteA.this, ActiviteB.class);
        myIntent.putExtra("MonObjet", this.obj);
        startActivity(myIntent);
    }
}
```


Le passage d'Objets

- Activité B affichant dans un TextView le contenu de l'objet

```
public class ActiviteB extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_activite_b);
        Intent myIntent = getIntent();
        SimpleObjectParcelable obj =
                (SimpleObjectParcelable) myIntent.getParcelableExtra("MonObjet");
        TextView textView = (TextView)findViewById(R.id.textView);
        textView.setText("id: "+obj.getName()+" value: "+obj.getValue());
    }
}
```


Le passage d'Objets

- Un autre solution consiste à passer par un format JSON
  - Consiste à transformer votre objet en chaine de caractère
    - La librairie GSON développée par Google simplifie considérablement cette tâche
      - ★ *toJson(…) qui prend en paramètre un objet et renvoie une String*
      - ★ *fromJson(…) qui effectue l'opération inverse*
  - La chaine de caractère est ensuite passée dans un Intent
  - Solution rapide mais à limiter à des objets simples
    - Privilégier la solution précédente pour les objets complexes

Le passage d'Objets

- Même exemple avec le format JSON

```
public class ActiviteA extends AppCompatActivity {

    //private SimpleObjectParcelable obj;
    private SimpleObject obj;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_activite);
        //this.obj = new SimpleObjectParcelable(1,"obj1");
        this.obj = new SimpleObject(1,"obj1");
    }

    public void sendObject(View view) {
        Intent myIntent = new Intent(ActiviteA.this, ActiviteB.class);
        myIntent.putExtra("MonObjet", (new Gson()).toJson(obj));
        startActivity(myIntent);
    }
}
```

- La chaine de caractère contient: {"name":"obj1","value":1}

Le passage d'Objets

- Coté récepteur

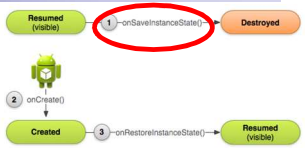
```
public class ActiviteB extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_activite_b);
        Intent myIntent = getIntent();
        //SimpleObjectParcelable obj =
        //        (SimpleObjectParcelable) myIntent.getParcelableExtra("MonObjet");
        SimpleObject obj =
                (new Gson()).fromJson(myIntent.getStringExtra("MonObjet"), SimpleObject.class);
        TextView textView = (TextView)findViewById(R.id.textView);
        textView.setText("id: "+obj.getName()+" value: "+obj.getValue());
    }
}
```


La persistance des données

- Les solutions précédentes nous permettent d'échanger des objets et des données entre activités pendant l'exécution d'une application
- Cependant, une activité peut être détruite
  - En cas de changement de géométrie de l'écran
  - Par le ramasse miette en cas de pénurie de mémoire sur le terminal
- Il est donc souvent nécessaire d'assurer la sauvegarde temporaire des données avant la fin de l'activité pour pouvoir les retrouver lors de la nouvelle instanciation
  - Pour la sauvegarde, on implémente la méthode onSaveInstanceState() qui prend en paramètre un Bundle
  - Pour la restauration, on utilise le bundle fournit en paramètre de onCreate()
- Cette solution n'est cependant qu'un procédé temporaire qui ne survit pas au redémarrage complet du système

La persistance des données

- onSaveInstanceState() est appelée avant la destruction de l'activité
  - Prend un Bundle en paramètre pour stocker les couples clés - valeurs




```
static final String STATE_SCORE = "playerScore";
static final String STATE_LEVEL = "playerLevel";
...

@Override
public void onSaveInstanceState(Bundle savedInstanceState) {
    // Save the user's current game state
    savedInstanceState.putInt(STATE_SCORE, mCurrentScore);
    savedInstanceState.putInt(STATE_LEVEL, mCurrentLevel);

    // Always call the superclass so it can save the view hierarchy state
    super.onSaveInstanceState(savedInstanceState);
}
```

Source: http://developer.android.com/training/basics/activity-lifecycle/recreating.html

La persistance des données

- La méthode onCreate() est modifiée pour pouvoir récupérer les données depuis le Bundle


```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState); // Always call the superclass first

    // Check whether we're recreating a previously destroyed instance
    if (savedInstanceState != null) {
        // Restore value of members from saved state
        mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
        mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
    } else {
        // Probably initialize members with default values for a new instance
    }
    ...
}
```

Source: http://developer.android.com/training/basics/activity-lifecycle/recreating.html

La persistance des données

- La méthode onRestoreInstanceState() est appelée lorsqu'une activité redevient visible
  - Fonctionnement très similaire à onCreate()


```
public void onRestoreInstanceState(Bundle savedInstanceState) {
    // Always call the superclass so it can restore the view hierarchy
    super.onRestoreInstanceState(savedInstanceState);

    // Restore state members from saved instance
    mCurrentScore = savedInstanceState.getInt(STATE_SCORE);
    mCurrentLevel = savedInstanceState.getInt(STATE_LEVEL);
}
```

Source: http://developer.android.com/training/basics/activity-lifecycle/recreating.html

La persistance des données

- Pour assurer une véritable persistance des données les solutions sont similaires à celle que l'on peut retrouver dans d'autres environnements
  - Stockage de fichiers
    - Sur la mémoire flash interne où se trouve le système de fichier principal
    - Sur la carte mémoire SD lorsqu'elle est présente
    - Sur le cloud à travers une connexion réseau
  - Stockage de données
    - Dans un environnement structuré comme une base de données relationnelle
      - ★ *SQLite sur Android*
- Android propose également une autre solution qui est le stockage de préférences avec SharedPreferences
  - Stockage de couple clés – valeurs
    - Attention: les valeurs ne peuvent pas être de type Objet

La persistance des données

- Une instance de SharedPreferences peut s'obtenir de deux manières
  - *getSharedPreferences()* qui permet de stocker les préférences dans plusieurs fichiers, chacun étant identifié par un nom
  - *getPreferences()* qui permet de tout stocker dans le même fichier
- L'instance permet ensuite d'obtenir un éditeur pour pouvoir stocker les couples clés – valeurs
  - *edit()*
- L'éditeur stocke les données à l'aide des méthodes correspondantes
  - *putBoolean(), putInt(), putString()…*
- A la fin du stockage, il faut faire appel à la méthode commit() pour les rendre effectifs

La persistance des données - Exemple

```
public void storePreferences(View view) {
    SharedPreferences mPrefs = getSharedPreferences("Fichier",0);
    SharedPreferences.Editor prefsEditor = mPrefs.edit();
    this.obj.setValue(5);
    Gson gs = new Gson();
    String json = gs.toJson(obj);
    prefsEditor.putString("MyObject", json);
    prefsEditor.commit();
}
```

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_activite);

    SharedPreferences mPrefs = getSharedPreferences("Fichier",0);
    Gson gs = new Gson();
    String json = mPrefs.getString("MyObject", "");
    if (json.compareTo("")==0)
       this.obj = new SimpleObject(1,"obj1");
    else
        this.obj = gs.fromJson(json, SimpleObject.class);

}
```

La persistance des données – Stockage de fichiers

- Chaque application dispose d'un espace réservé pour stocker ses fichiers
  - Récupérable à l'aide de FileContext.getFilesDir()
  - Les noms de fichiers sont stockés au format UTF-8
  - Le système de fichier peut être chiffré
- Les opérations réalisables depuis l'application sont similaires à ce que l'on peut retrouver en Java
- On les retrouve dans l'interface Context et donc dans l'Activité
  - *FileInputStream openFileInput(String name)*
  - *FileOutputStream openFileOutput(String name, int mode)*
  - *File getDir(String name, int mode)*
  - *File deleteFile(String name)*
  - *String[] fileList()*
  - …

La persistance des données – Stockage de fichiers

- Le mode définit la visibilité et les droits associés au fichier
  - MODE_APPEND permet d'ajouter en fin de fichier
  - MODE_PRIVATE permet d'écrire un fichier uniquement accessible par l'application
  - Deux autres modes existaient avant l'API 17: INTERDICTION DE LES UTILISER !!!
    - MODE_WORLD_READABLE
    - MODE_WORLD_WRITABLE
- Les fichiers externes à l'application sont toujours publiques et non chiffrés
- On peut y accéder à travers l'interface Context et surtout la classe Environment
  - Elle fournit de nombreuses méthodes statique pour obtenir des fichiers
    - getExternalStorageDirectory(), getRootDirectory(), getExternalStoragePublicDirectory(String type), etc

La persistance des données – Stockage de fichiers

- Le stockage de fichier sous Android doit respecter des conventions pour permettre aux applications de les manipuler plus simplement
- Plusieurs catégories ont été définies
  - *DIRECTORY_DOCUMENTS*
  - *DIRECTORY_MUSIC*
  - *DIRECTORY_PODCASTS*
  - *DIRECTORY_RINGTONES*
  - *DIRECTORY_ALARMS*
  - *DIRECTORY_NOTIFICATIONS*
  - *DIRECTORY_PICTURES*
  - *DIRECTORY_MOVIES*
  - *DIRECTORY_DOWNLOADS*
  - *DIRECTORY_DCIM*

La persistance des données – Stockage de fichiers

- La majorité des méthodes renvoient des objets Java bien connus
  - java.io.File
  - java.io.InputStream et java.io.OutputSteam
- Leur manipulation est donc identique à celle que vous avez pu voir en Java
  - Scanner
  - BufferedReader
  - PrintStream
  - …
- Remarque: Il est possible d'embarquer un texte directement dans l'application en le sauvant dans res/raw/

La persistance des données – Stockage de fichiers

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        TextView tv = (TextView)findViewById(R.id.textView);
        String texte = readFile(this, R.raw.test );
        tv.setText(texte);
    }

    public String readFile(Context ctx, int resId) {
        InputStream inputStream =
                ctx.getResources().openRawResource(resId);
        Scanner sc = new Scanner(inputStream);
        String line="";
        while (sc.hasNextLine()){
            line = line+sc.nextLine()+"\n";
        }

        return line;
    }
}
```

La persistance des données – Stockage de fichiers

- Par défaut les données sont lues ou écrites sur le stockage système
  - Directement lié à l'appareil
  - Les données peuvent être privées
  - Taille assez faible
- Utiliser l'espace de stockage externe présente plusieurs avantages
  - Le support est en général amovible
  - La taille est généralement beaucoup plus grande (8-64go)
  - Les données ne sont pas supprimées lors de la désinstallation de l'application
- Mais aussi des inconvénients…
  - Il peut tout simplement ne pas y en avoir dans le terminal
  - Les données stockées ne sont pas privées et sont donc accessible par n'importe quelle application
  - Les données ne sont pas supprimées lors de la désinstallation de l'application

La persistance des données – Stockage de fichiers

- L'utilisation du stockage externe nécessite de bénéficier de certaines permissions de la part de l'utilisateur

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

- Il est recommandé de toujours tester dans votre application si le support externe est disponible/utilisable

```
public boolean isExternalStorageWritable() {
    return Environment.MEDIA_MOUNTED.equals(
            Environment.getExternalStorageState());
}

public boolean isExternalStorageReadable() {
    return isExternalStorageWritable() ||
            Environment.MEDIA_MOUNTED_READ_ONLY.equals(
                    Environment.getExternalStorageState());
}
```


La persistance des données – SQLite

- Android propose nativement une base de donnée: SQLite 3
  - Stockage dans une zone privée et sécurisée
- Permet la persistance de données structurées sous forme de tables
  - Possibilité d'avoir des relations entre tables (jointures)
- Propose le maintient d'un index de tri sur les enregistrements de la table pour faire des requêtes de sélection rapides
- Fourniture d'une API permettant de s'abstraire partiellement du langage SQL à travers la classe SQLiteOpenHelper
  - On spécialise la classe avec sa configuration en redéfinissant les méthodes onCreate(), onUpgrade(), onOpen(), …

La persistance des données – SQLite

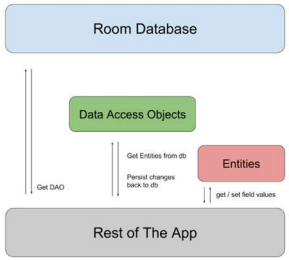
- Un objet de type SQLiteOpenHelper permet d'obtenir la base de données proprement dite de type SQLiteDatabase
  - getWritableDatabase()
  - getReadableDatabase()
- Un objet SQLiteDatabase se manipule à l'aide de la méthode query()
  - C'est une méthode polymorphe permettant de composer des requêtes très variées
  - La méthode query() renvoie un Cursor qui permet de parcourir les résultats renvoyés par la requête
    - getCount() donne le nombre de résultats
    - getInt(int i) renvoie la valeur entière de la ième colonne de l'enregistrement courant
      - ★ *Marche avec Int, Short, Float, String…*
    - moveToNext() pour passer au résultat suivant

La persistance des données – SQLite

- La méthode insert() permet de stocker des valeurs de type ContentValues dans une table
  - ContentValues est une sorte de dictionnaire où les valeurs des colonnes sont insérées avec put()
- La méthode update() permet de mettre à jour un ou plusieurs enregistrements
- La méthode delete() permet de supprimer un ou plusieurs enregistrements

La persistance des données – Room Persistance API

- Dans les versions récentes d'Android, il est recommandé de passer par l'API Room pour dialoguer avec la base de données SQLite
  - Disponible dans la librairie Jetpack fournie par Google
  - C 'est une couche d'abstraction pour SQLite
- Concrètement, Room se compose de trois composants
  - Database: qui défini le point d'accès vers la base de données
  - Entity: qui désigne une table dans la base de données
  - DAO: qui contient les méthodes pour accéder à la base de données



La persistance des données – Room Persistance API

- Exemple d'utilisation
  - Prenons une classe User à persister dans la base:

```
@Entity
public class User {
    @PrimaryKey
    public int uid;

    @ColumnInfo(name = "first_name")
    public String firstName;

    @ColumnInfo(name = "last_name")
    public String lastName;
}
```

  - Cette classe va être manipulée à l'aide d'un DAO

```
@Dao
public interface UserDao {
    @Query("SELECT * FROM user")
    List<User> getAll();

    @Query("SELECT * FROM user WHERE uid IN (:userIds)")
    List<User> loadAllByIds(int[] userIds);

    @Query("SELECT * FROM user WHERE first_name LIKE :first AND " +
           "last_name LIKE :last LIMIT 1")
    User findByName(String first, String last);

    @Insert
    void insertAll(User... users);

    @Delete
    void delete(User user);
}
```

La persistance des données – Room Persistance API

- Une fois le DAO défini, on construit une représentation de la base en s'appuyant sur la classe RoomDatabase

```
@Database(entities = {User.class}, version = 1)
public abstract class AppDatabase extends RoomDatabase {
    public abstract UserDao userDao();
}
```

- Cette représentation est ensuite instanciée pour obtenir une référence vers la base de donnée

```
AppDatabase db = Room.databaseBuilder(getApplicationContext(),
        AppDatabase.class, "database-name").build();
```