

# Formation Android

Ihab ABADI / UTOPIOS

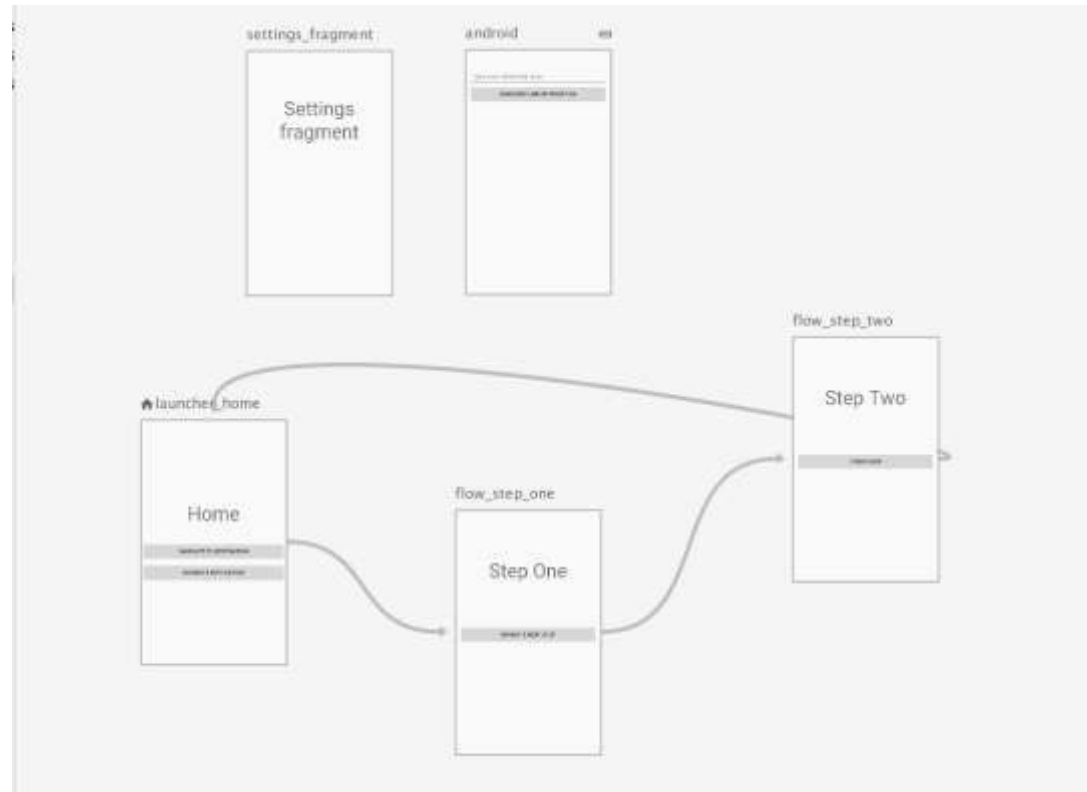
# SOMMAIRE – Partie 1

1. Navigation
2. Gestion de stockages

# Principes de navigation

- Point de départ - visible lorsque l'application est lancée à partir d'un lanceur, c'est le dernier écran après retour en appuyant sur le bouton retour
- L'état de navigation est représenté par une pile (destination de départ en bas, destination actuelle en haut de la pile)
- Le bouton Haut amène l'utilisateur à la destination parent hiérarchique, ne quitte jamais l'application
- Le bouton Haut fonctionne de la même manière que le bouton Retour du système dans la tâche de votre propre application (lorsque le bouton Retour ne quitte pas votre application)
- Le lien profond vers une destination a la même pile qu'en naviguant à partir de la destination de départ (l'utilisateur peut utiliser les boutons Précédent ou Haut pour naviguer
- à travers les destinations jusqu'à la destination de départ). Toute pile de navigation existante est supprimée et remplacée par la pile de navigation du lien profond.

# Représentation visuelle du navgraph



# Configuration de la navigation dans le projet

- Android Studio 3.2
- **File > Settings (Android Studio > Preferences on Mac)**, select the **Experimental** category in the left pane, check **Enable Navigation Editor**

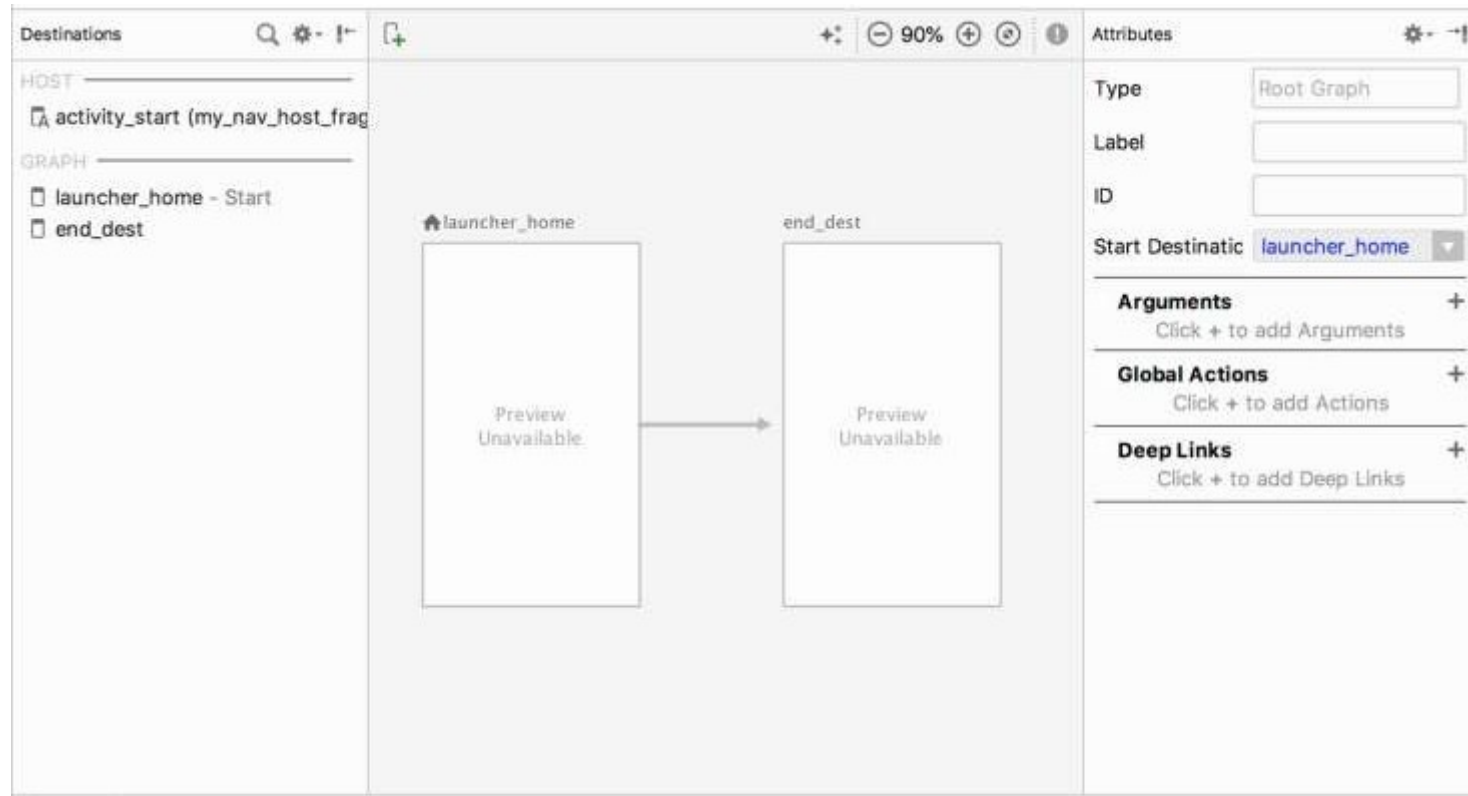
- **Gradle:**

```
implementation "android.arch.navigation:navigation-fragment:$versions.navigation"  
implementation "android.arch.navigation:navigation-ui:$versions.navigation"
```

- new resource file of type Navigation in res folder:
  - navigation resource directory is created
  - nav\_graph.xml is created which contains navigation graph



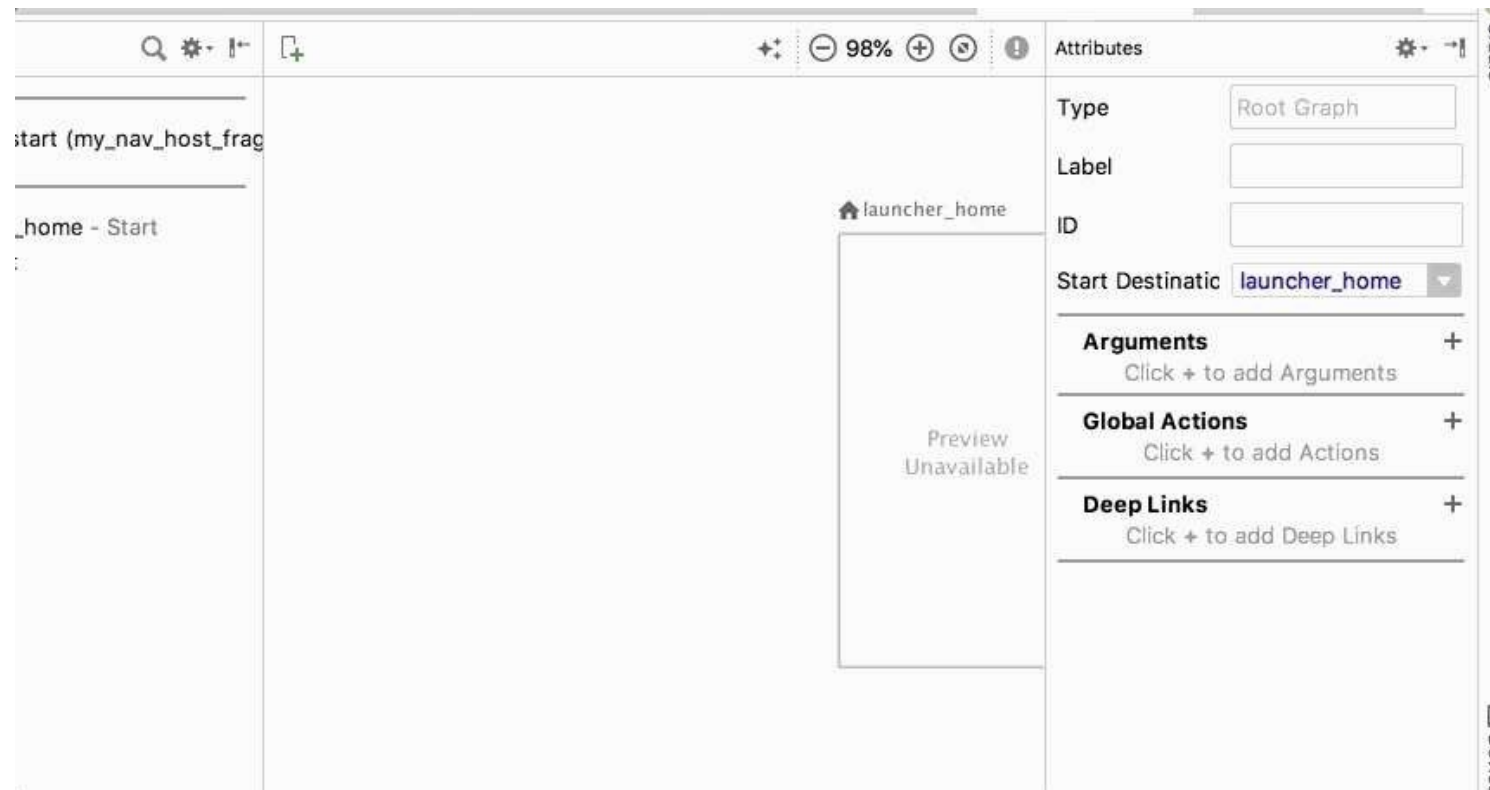
# Editeur des navgraphs



# Destinations

- Destinations:
  - Activities
  - Fragments
- Can be created in Editor as blank destination or from existing Fragments and Activities in our project.

# Destinations : Création d'une nouvelle





# Représentation XML

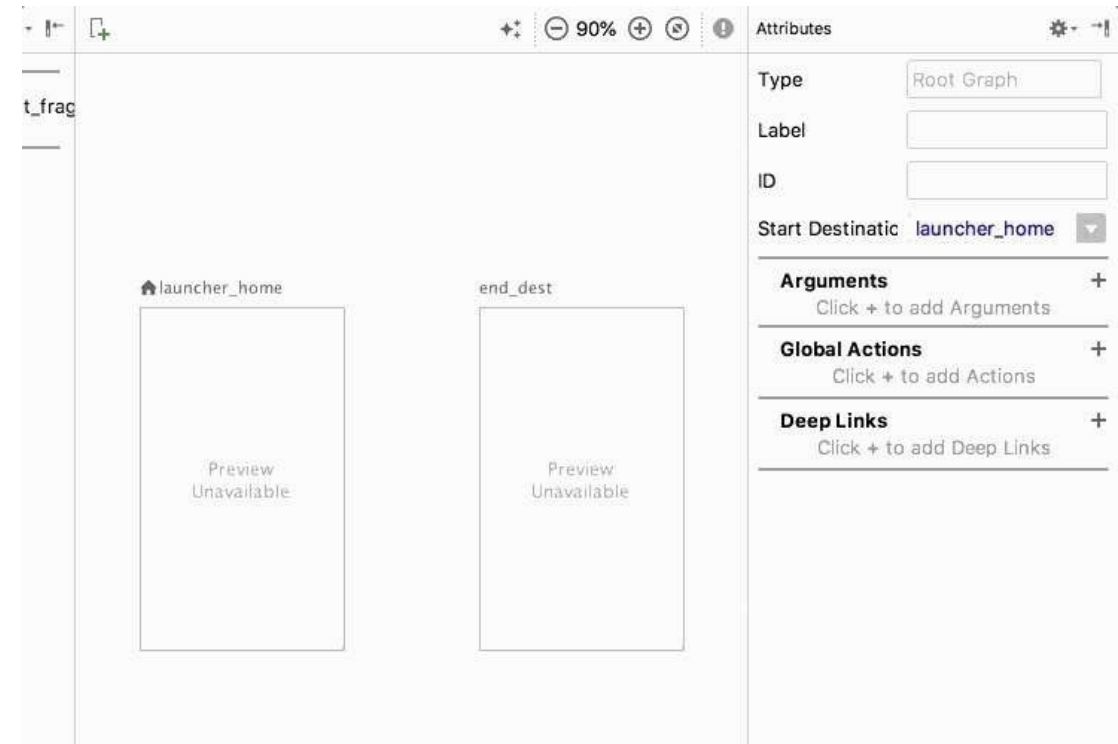
```
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
            xmlns:app="http://schemas.android.com/apk/res-auto"
            xmlns:tools="http://schemas.android.com/tools"
            app:startDestination="@+id/startFragment">
    <fragment
        android:id="@+id/startFragment"
        android:name="com.android.samples.arch.componentsbasicsample.StartFragment"
        android:label="fragment_start"
        tools:layout="@layout/fragment_start" />
</navigation>
```

# Destination initial

```
💡 app:startDestination="@+id/startFragment">  
<fragment  
    android:id="@+id/startFragment"
```

# Connexion entre destinations - Actions

Les destinations sont connectées à l'aide d'actions



# Action en XML

```
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    app:startDestination="@+id/launcher_home">

    <fragment android:id="@+id/launcher_home"
        android:label="Home"
        android:name="com.android.samples.arch.componentsbasicsample.StartFragment" >

        <action android:id="@+id/end_action" app:destination="@id/end_dest"/>

    </fragment>

    <fragment android:id="@+id/end_dest"
        android:label="End"
        android:name="com.android.samples.arch.componentsbasicsample.EndFragment" >

    </fragment>
</navigation>
```

# Hosting Navigation

Une activity accueille la navigation à l'aide d'une **NavHost** interface.  
Le NavHost par défaut est NavHostFragment.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <fragment
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/my_nav_host_fragment"
        android:name="androidx.navigation.fragment.NavHostFragment"
        app:navGraph="@navigation/nav_graph"
        app:defaultNavHost="true"
        />

</android.support.constraint.ConstraintLayout>
```

# Hosting Navigation

- NavHostFragment permute entre différentes destinations de fragment lorsque l'utilisateur navigue dans le graphique de navigation.
- Le graphique de navigation est associé par l'attribut navGraph :

```
app:navGraph="@navigation/nav_graph"
```

- Pour vous assurer que NavHostFragment intercepte le bouton Retour du système :

```
app:defaultNavHost="true"
```

- Pour prendre en charge le bouton Haut, il est nécessaire d'écraser

```
override fun onSupportNavigateUp()  
    = findNavController(R.id.nav_host_fragment).navigateUp()
```

# Navigation Controller

- La navigation vers une destination est effectuée par la classe NavController.
- Un NavController peut être récupéré à l'aide de l'une des méthodes statiques suivantes :
  - NavHostFragment.findNavController(Fragment)
  - Navigation.findNavController(Activity, @IdRes int viewId)
  - Navigation.findNavController(View)
  - View.findNavController()
- La méthode navigate() est utilisée pour naviguer vers une destination. Il accepte l'ID d'une destination ou d'une action.

```
view?.findViewById<Button>(R.id.navigate_bt)?.setOnClickListener { it: View!  
    // Navigate to the login destination  
    view?.let { Navigation.findNavController(it).navigate(R.id.end_action) }  
}
```

# Navigation Controller

Méthodes à utiliser:

- `NavController.navigateUp()`
- `NavController.popBackStack()`



# Transitions entre destinations

- Les transitions peuvent être facilement ajoutées via XML ou le panneau Attributs

Attributes

Type: Action

ID: next\_action

Destination: flow\_step\_one

**Transitions**

Enter: slide\_in\_right

Exit: slide\_out\_left

Pop Enter: slide\_in\_left

Pop Exit: slide\_out\_right

```
<fragment
    android:id="@+id/launcher_home"
    android:name="com.example.android.codelabs.navigation.MainFragment"
    android:label="Home"
    tools:layout="@layout/main_fragment">

    <action
        android:id="@+id/next_action"
        app:destination="@+id/flow_step_one"
        app:enterAnim="@anim/slide_in_right"
        app:exitAnim="@anim/slide_out_left"
        app:popEnterAnim="@anim/slide_in_left"
        app:popExitAnim="@anim/slide_out_right" />

</fragment>
```

# Transitions entre destinations

```
val options = NavOptions.Builder()
    .setEnterAnim(R.anim.slide_in_right)
    .setExitAnim(R.anim.slide_out_left)
    .setPopEnterAnim(R.anim.slide_in_left)
    .setPopExitAnim(R.anim.slide_out_right)
    .build()



view.findViewById<Button>(R.id.navigate_dest_bt)?.setOnClickListener { it: View!
    findNavController(it).navigate(R.id.flow_step_one, args: null, options)
}
```

# Passage de données entre destinations

- Chaque destination peut définir les arguments qu'elle peut recevoir.
- Deux manières possibles de transmettre des données entre les destinations :
  - en utilisant Bundle
  - manière sécurisée en utilisant le plugin safeargs Gradle
- Cela peut être fait facilement dans le panneau Attributs de la destination dans le
- Rubrique Arguments.

# Passage de données entre destinations

## Destination's Attributes:


Attributes  


Type


Label

ID



Class

▼ Arguments   
Click + to add Arguments

▼ Actions   
Click + to add Actions

▼ Deep Links   
Click + to add Deep Links

## Action's Attributes:

Attributes  

Type

ID

Destination

▼ Transitions

Enter

Exit

Pop Enter

Pop Exit

▼ Argument Default Values

param1	string	default value
--------	--------	---------------

▼ Pop Behavior

Pop To

Inclusive ☒

▼ Launch Options

Single Top ☒

# Passage de données entre destinations

```
<fragment
  android:id="@+id/settings_fragment"
  android:name="com.example.android.code labs .navigation.SettingsFragment"
  android:label="fragment_settings"
  tools:layout="@layout/fragment_settings">
  <argument
    android:name="param1"
    android:defaultValue="testValue"
    app:type="string" />
</fragment>
```

# Envoie et réception d'arguments

Créer un Bundle et ajouter les arguments à l'aide navigate():

```
val bundle = Bundle()  
bundle.putString("param1", "Hello!")  
Navigation.findNavController(view).navigate(R.id.next_action, bundle)
```

Pour recevoir les arguments, on peut utiliser la méthode getArguments:

```
val param1 = arguments?.getString(key: "param1")
```

# Les options de stockages

- Shared Preferences
  - Stockage privé clé valeur.
- Internal Storage
  - Stockage privé local.
- External Storage
  - Stockage privé externe.
- SQLite Databases
  - Base de données locale.
- Network Connection

# SharedPreferences

- La classe SharedPreferences fournit un cadre général qui vous permet d'enregistrer et de récupérer des paires clé-valeur persistantes de types de données primitifs.
- Ces données persisteront d'une session utilisateur à l'autre.



# SharedPreferences

## **Pour utiliser les sharedPreferences :**

- Initialisation.
- Stockage des données.
- Commit du changement.
- Récupération des données.
- Nettoyage ou suppression des données.

# Initialisation

Les préférences partagées de l'application peuvent être récupérées en utilisant la méthode `getSharedPreferences()`.

Vous avez également besoin d'un éditeur pour modifier et enregistrer les modifications dans les préférences partagées.

```
SharedPreferences pref =  
getApplicationContext().getSharedPreferences("FileName", 0); // 0 - for private mode  
Editor editor = pref.edit();
```

# Shared preferences

- `getSharedPreferences()` – à utiliser si besoin de plusieurs fichiers de préférences identifiés par leur nom, que vous spécifiez avec le premier paramètre.
- `getPreferences()` – à utiliser si besoin que d'un seul fichier de préférences pour votre activité. Comme il s'agira du seul fichier de préférences pour votre activité, vous ne fournissez pas de nom.

# Sauvegarde des données

- Vous pouvez enregistrer des données dans des préférences partagées à l'aide de l'éditeur. Tous les types de données primitifs tels que les booléens, les flottants, les entiers, les longs et les chaînes sont pris en charge.
- Vous devez mettre des données en tant que clé avec valeur.

```
editor.putBoolean("key_name", true); // Storing boolean - true/false
editor.putString("key_name", "string value"); // Storing string
editor.putInt("key_name", "int value"); // Storing integer editor.putFloat("key_name", "float
value"); // Storing float editor.putLong("key_name", "long value"); // Storing long
```

# Commit des changements

- Pour valider les changement :
- On appelle la méthode commit.

# Récupération des données

- Les données peuvent être récupérées à partir des préférences enregistrées en appelant Méthode getString() (pour chaîne). N'oubliez pas que cette méthode doit être appelée sur les préférences partagées et non sur l'éditeur.

```
// returns stored preference value
// If value is not present return default value - In this case null
pref.getString("key_name", null); // getting String
pref.getInt("key_name", null); // getting Integer
pref.getFloat("key_name", null); // getting Float
pref.getLong("key_name", null); // getting Long
pref.getBoolean("key_name", null); // getting boolean
```

# Suppression des données

Pour supprimer des préférences partagées, vous pouvez appeler `remove("key_name")` pour supprimer une valeur particulière. Si vous voulez supprimer toutes les données, appelez `clear()`.

```
editor.remove("name"); // will delete key name  editor.remove("email"); // will delete key email  
editor.commit(); // commit changes;
```

```
editor.clear(); //clear all the data from shared preferences editor.commit(); // commit changes
```

# Utilisation des sharedPreferences

- Paramètres généraux
- Sélection de la langue
- Taille du texte
- E-mail
- Couleur du texte
- Paramètres système
- Activer les notifications
- utiliser le Wi-Fi



# Stockage interne

- Vous pouvez enregistrer des fichiers directement sur le stockage interne de l'appareil.
- Par défaut, les fichiers enregistrés dans le stockage interne sont privés pour votre application et les autres applications ne peuvent pas y accéder.
- Lorsque l'utilisateur désinstalle votre application, ces fichiers sont supprimés.

# Création et écriture stockage interne

- Appelez `openFileOutput()` avec le nom du fichier et le mode de fonctionnement. Cela renvoie un `FileOutputStream`.
- Écrivez dans le fichier avec `write()`.
- Fermez le flux avec `close()`.

# Création et écriture stockage interne

```
String FILENAME = "hello_file";
```

```
String string = "hello world!";
```

```
FileOutputStream fos =
```

```
openFileOutput(FILENAME,Context.MODE_PRIVATE);
```

```
fos.write(string.getBytes());
```

```
fos.close();
```

# Lecture stockage interne

- Appelez `openFileInput()` et passez-lui le nom du fichier à lire. Cela renvoie un `FileInputStream`.
- Lire les octets du fichier avec `read()`.
- Fermez ensuite le flux avec `close()`.

# SQLite

- “Bibliothèque [...] qui propose un **moteur de base de données relationnelle** accessible par le langage **SQL**”

# SQLite - Avantages

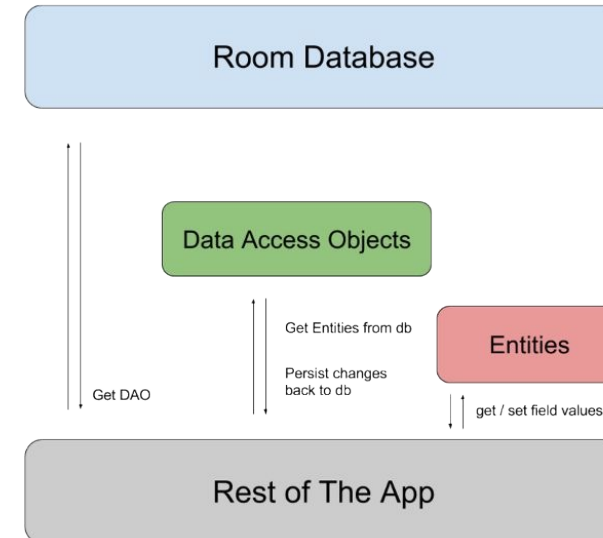
- SQL
- Base de données embarquée
- SGBD complet :
  - gestion des transactions
  - gestion des *triggers*
  - gestion des vues
  - etc.

# SQLite - Inconvénients

- SQL
- Comment faire de l'objet ?
  - *pattern* DAO
  - *pattern* repository
  - etc

# Room

- Fonctionne par-dessus SQLite
- Fonctionne avec 3 éléments :
  - Database
  - Entity
  - DAO
- Copie les éléments de la base de données vers des objets





# Room

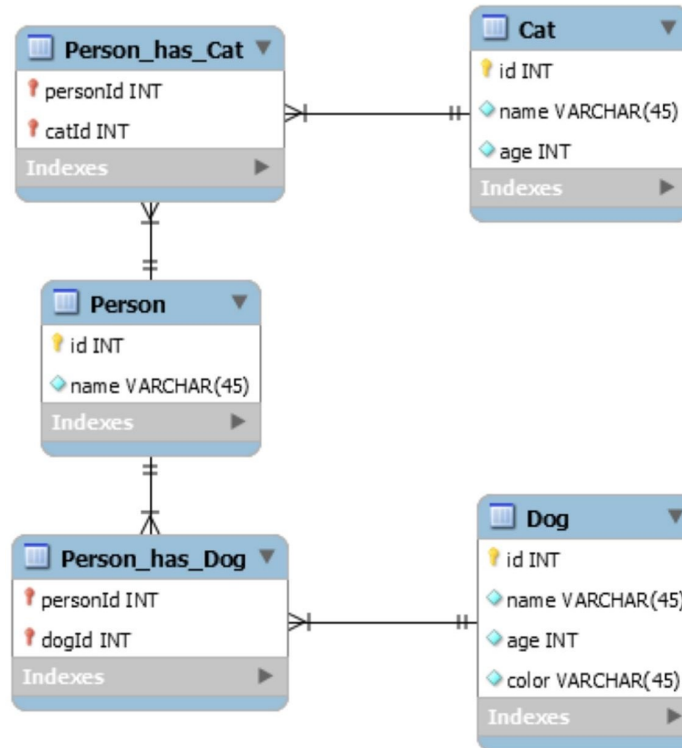
1. Ajout d'une nouvelle source dans le fichier **build.gradle** du projet

```
maven { url "https://maven.google.com" } // google()
```

2. Ajout des dépendances dans le fichier **build.gradle** du module

```
compile("android.arch.persistence.room:runtime:1.0.0")  
annotationProcessor("android.arch.persistence.room:compiler:1.0.0")
```

# Room



# Room

```
public abstract class RoomAnimal
{

    @PrimaryKey
    public int id;

    public int age;

    public String name;

}
```

```
@Entity(tableName = "cat")
public final class RoomCat
    extends RoomAnimal
{

}
```

# Room

```
@Entity(tableName = "dog")
public final class RoomDog
    extends RoomAnimal
{

    public enum RoomColor
    {
        Black, White
    }

    @TypeConverters(RoomColorConverter.class)
    public RoomColor color;
}
```

```
public final class RoomColorConverter
{

    @TypeConverter
    public RoomColor fromString(String color)
    {
        return color != null ?
            RoomColor.valueOf(color) : null;
    }

    @TypeConverter
    public String fromRealmColor(RoomColor color)
    {
        return color.toString();
    }

}
```

# Room

```
@Entity(tableName = "person")
public final class RoomPerson
{
    @PrimaryKey
    public int id;

    public String name;
}
```

```
public final class RoomPersonWithAnimals
{
    @Embedded
    public RoomPerson person;

    @Relation(parentColumn = "id",
              entityColumn = "id",
              entity = RoomDog.class)
    public List<RoomDog> dogs;

    @Relation(parentColumn = "id",
              entityColumn = "id",
              entity = RoomCat.class)
    public List<RoomCat> cats;
}
```

# Room

```
@Entity(  
    tableName = "Person_has_Cat",  
    primaryKeys = { "personId", "catId" },  
    foreignKeys = {  
        @ForeignKey(entity = RoomPerson.class,  
            parentColumns = "id",  
            childColumns = "personId",  
            onDelete = ForeignKey.CASCADE),  
        @ForeignKey(entity = RoomCat.class,  
            parentColumns = "id",  
            childColumns = "catId",  
            onDelete = ForeignKey.CASCADE) })  
public final class RoomPersonCat  
{  
  
    public int personId;  
  
    public int catId;  
  
}
```

```
@Entity(  
    tableName = "Person_has_Dog",  
    primaryKeys = { "personId", "dogId" },  
    foreignKeys = {  
        @ForeignKey(entity = RoomPerson.class,  
            parentColumns = "id",  
            childColumns = "personId",  
            onDelete = ForeignKey.CASCADE),  
        @ForeignKey(entity = RoomDog.class,  
            parentColumns = "id",  
            childColumns = "dogId",  
            onDelete = ForeignKey.CASCADE) })  
public final class RoomPersonDog  
{  
  
    public int personId;  
  
    public int dogId;  
  
}
```

# Room

```
@Dao
public interface RoomGenericDao<T>
{

    @Insert(
        onConflict = OnConflictStrategy.REPLACE)
    void insert(T bo);

    @Insert(
        onConflict = OnConflictStrategy.REPLACE)
    void insertAll(T... bo);

    @Delete
    void delete(T bo);

    @Update
    void update(T bo);

}
```

```
@Dao
public interface RoomCatDao
    extends RoomGenericDao<RoomCat>
{

    @Query("SELECT * FROM cat")
    List<RoomCat> getAll();

    @Query("SELECT * FROM cat WHERE id = :id")
    RoomCat findById(int id);

}
```

# Room

```
@Dao
public interface RoomDogDao
    extends RoomGenericDao<RoomDog>
{

    @Query("SELECT * FROM dog")
    List<RoomDog> getAll();

    @Query("SELECT * FROM dog WHERE id = :id")
    RoomDog findById(int id);

}
```

```
@Dao
public interface RoomPersonDao
    extends RoomGenericDao<RoomPerson>
{

    @Query("SELECT * FROM person")
    List<RoomPersonWithAnimals> getAll();

    @Query("SELECT * FROM person WHERE id = :id")
    RoomPersonWithAnimals findById(int id);

}
```



# Room

## 1. Création d'une classe abstraite pour exposer les DAO et les entités managés

```
@Database(entities = { RoomCat.class, RoomDog.class, RoomPerson.class, RoomPersonCat.class,
RoomPersonDog.class }, version = 1, exportSchema = false)
public abstract class MyRoomDatabase extends RoomDatabase {

    public abstract RoomCatDao roomCatDao();

    public abstract RoomDogDao roomDogDao();

    public abstract RoomPersonDao roomPersonDao();
}
```

## 2. Instanciation de la classe

```
myRoomDatabase = Room.databaseBuilder(this, MyRoomDatabase.class, "myRoomDatabase").build();
```

# Room

```
//1. Création d'une transaction (facultatif) → Interdit dans le main thread
myRoomDatabase.beginTransaction();

//2. Exécution de la requête
myRoomDatabase.roomPersonDao().insertAll(person);

//3. On indique que tout s'est bien passé
myRoomDatabase.setTransactionSuccessful();

//4. On termine la transaction
myRoomDatabase.endTransaction();
```

# Room

## Requête de sélection

```
final RoomPersonWithAnimals personWithAnimals = myRoomDatabase.roomPersonDao().findById(299);
```

## Requête de mise à jour

```
final RoomPersonWithAnimals personWithAnimals = myRoomDatabase.roomPersonDao().findById(299);  
personWithAnimals.person.name = UUID.randomUUID().toString();  
  
myRoomDatabase.roomPersonDao().update(personWithAnimals.person);
```

## Requête de suppression

```
final RoomPersonWithAnimals persons = myRoomDatabase.roomPersonDao().findById(3);  
myRoomDatabase.roomPersonDao().delete(persons.person);
```

# Room

```
final List<RoomPersonWithAnimals> personsWithAnimals = new ArrayList<>();

for (int index = 0; index < NUMBER_OF_ITEMS; index++) { /*...*/ }
//1. Création d'une transaction (facultatif)
myRoomDatabase.beginTransaction();

//2. Insertion des éléments
for (RoomPersonWithAnimals personAnimals : personsWithAnimals) {
    myRoomDatabase.roomPersonDao().insert(personAnimals.person);
    myRoomDatabase.roomCatDao().insertAll(personWithAnimals.cats.toArray(newRoomCat[personWithAnimals.cats.size()]));
    myRoomDatabase.roomDogDao().insertAll(personWithAnimals.dogs.toArray(newRoomDog[personWithAnimals.dogs.size()]));

    for (RoomCat cat : personWithAnimals.cats) {
        myRoomDatabase.roomPersonCatDao().insert(new RoomPersonCat(personWithAnimals.person.id, cat.id));
    }

    for (RoomDog dog : personWithAnimals.dogs){
        myRoomDatabase.roomPersonDogDao().insert(new RoomPersonDog(personWithAnimals.person.id, dog.id));
    }
}

//3. On termine la transaction
myRoomDatabase.setTransactionSuccessful();
myRoomDatabase.endTransaction();
```

# Retrofit

- ***Un client REST de type sécurisé pour Android et Java.***
- Retrofit transforme votre API REST en une interface Java. Il utilise des annotations pour décrire les requêtes HTTP, le remplacement des paramètres URL et la prise en charge des paramètres de requête est intégrée par défaut. En outre, il fournit des fonctionnalités pour le téléchargement de requêtes et de fichiers en plusieurs parties.

# Retrofit

- ***Installation***

- dependencies {

...

compile 'com.squareup.retrofit2:converter-gson:2.3.0'

compile 'com.squareup.retrofit2:retrofit:2.3.0'

... }

# Retrofit - Gson

- **Les annotations `@SerializedName`** proviennent de la bibliothèque GSON et nous permettent de serialize et deserialize cette classe en JSON utilisant le nom sérialisé comme clé..

```
public class Device
{
    @SerializedName("deviceId")
    public String id;

    @SerializedName("name")
    public String name;

    @SerializedName("eventCount")
    public int eventCount;
}
```

# Retrofit – Interface API

- L'annotation `@GET` provient de Retrofit et indique à la bibliothèque que nous définissons une requête GET.
- Le chemin entre parenthèses est le point final que notre requête GET doit atteindre (nous définirons l'URL de base un peu plus tard).
- Les accolades nous permettent de remplacer des parties du chemin au moment de l'exécution pour que nous puissions passer des arguments.
- La fonction que nous définissons s'appelle `getDevice` et prend l'identifiant de périphérique que nous voulons comme argument.
- L'annotation `@PATH` indique à Retrofit que cet argument doit remplacer l'espace réservé "deviceId" dans le chemin.
- La fonction renvoie un objet `Call` de type `Device`.

```
public interface DeviceAPI
{
    @GET("device/{deviceId}")
    Call<Device> getDevice (@Path("deviceId") String deviceId);

    @GET("devices")
    Call<List<Device>> getDevices();
}
```



# Retrofit – Classe Client (Wrapper)

- Cette classe crée une instance GSON pour pouvoir analyser la réponse JSON, crée une instance Retrofit avec notre URL de base et un GsonConverter, puis crée une instance de notre API.

```
public class DeviceAPIHelper
{
    public final DeviceAPI api;

    private DeviceAPIHelper ()
    {
        Retrofit retrofit = new Retrofit.Builder()
            .baseUrl("http://example.com/")
            .addConverterFactory(GsonConverterFactory.create())
            .build();

        api = retrofit.create(DeviceAPI.class);
    }
}
```

# Retrofit – Appel de l'API

```
// Getting a JSON object
Call<Device> callObject = api.getDevice(deviceID);
callObject.enqueue(new Callback<Response<Device>>()
{
    @Override
    public void onResponse (Call<Device> call, Response<Device> response)
    {
        if (response.isSuccessful())
        {
            Device device = response.body();
        }
    }

    @Override
    public void onFailure (Call<Device> call, Throwable t)
    {
        Log.e(TAG, t.getLocalizedMessage());
    }
});
```

# Retrofit – Appel de l'API

```
// Getting a JSON array
Call<List<Device>> callArray = api.getDevices();
callArray.enqueue(new Callback<Response<List<Device>>>()
{
    @Override
    public void onResponse (Call<List<Device>> call, Response<List<Device>> response)
    {
        if (response.isSuccessful())
        {
            List<Device> devices = response.body();
        }
    }

    @Override
    public void onFailure (Call<List<Device>> call, Throwable t)
    {
        Log.e(TAG, t.getLocalizedMessage());
    }
});
```

# Retrofit – Appel de l'API

- interface API pour créer un objet `Call<Device>` et pour créer un `Call<List<Device>>` respectivement.
- L'appel de enqueue indique à Retrofit de faire cet appel sur un thread d'arrière-plan et de renvoyer le résultat au rappel que nous créons ici.