# TRAVAUX PRATIQUES – FILIÈRE DÉVELOPPEMENT DIGITAL

## Option Applications mobiles

## M212 – S'initier aux composants et modèle d'une application Android

65 heures

# SOMMAIRE



# MODALITÉS PÉDAGOGIQUES

1

**LE GUIDE DE SOUTIEN**

Il contient le résumé théorique et le manuel des travaux pratiques


2

**LA VERSION PDF**

Une version PDF est mise en ligne sur l'espace apprenant et formateur de la plateforme WebForce Life

3

**DES CONTENUS TÉLÉCHARGEABLES**

Les fiches de résumés ou des exercices sont téléchargeables sur WebForce Life

4

**DU CONTENU INTERACTIF**

Vous disposez de contenus interactifs sous forme d'exercices et de cours à utiliser sur WebForce Life

5

**DES RESSOURCES EN LIGNES**

Les ressources sont consultables en synchrone et en asynchrone pour s'adapter au rythme de l'apprentissage

# PARTIE 1

## Maîtriser l’architecture d’une application Android

**Dans ce module, vous allez :**

- Structurer correctement une application Android
- Gérer correctement le cycle de vie d’une application Android
- Maîtriser la communication entre composants applicatifs

**16 heures**


# ACTIVITÉ n° 1

## Mise en œuvre d’une application avec plusieurs fragments et une seule activité

### Compétences visées :

- Utilisation des fragments
- Communication entre les fragments

### Recommandations clés :

- Révision générale du résumé théorique
- Suivre les instructions du TP et organiser le dossier de travail

**6 heures**

# CONSIGNES

## Pour le formateur :

- Faire un rappel de cours
- Demander aux stagiaires de suivre les étapes de l'activité

## Pour l'apprenant :

- Suivre les étapes indiquées dans l'activité

## Conditions de réalisation :

- Support de résumé théorique accompagnant
- Accès Internet pour installer les dépendances

## Critères de réussite :

- Le stagiaire doit être capable de :
  - manipuler les fragments
  - communiquer entre les fragments

# Activité n° 1

## Android - Passage des données entre fragments

### Enoncé

L'objectif de cette activité est de :

- Développer une application Android qui contient un **TabLayout**, un **ViewPager** et des **fragments** ;
- Permettre le passage des données d'un fragment à l'autre.

Le flux pour envoyer des données de type String d'un fragment à un autre est présenté ci-dessous.


1. Déclarer une interface personnalisée appelée **SendMessage** qui contient une méthode **send(String)** Pour envoyer un string au **FragmentB**, appeler la méthode send sur une instance de **SendMessage**.
2. Implémenter l'interface **SendMessage** et surcharger la méthode **send(String)**
3. Créer une méthode **get(String) FragmentB.get(Chaîne)**. Cette méthode récupère la chaîne de caractères de send et la transmet à la méthode get du **FragmentB**.

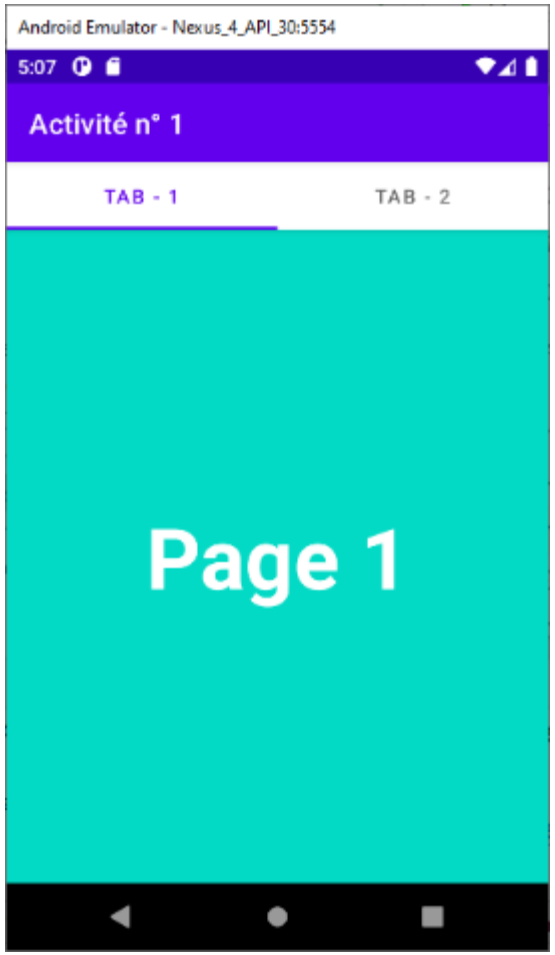
PARTIE 1

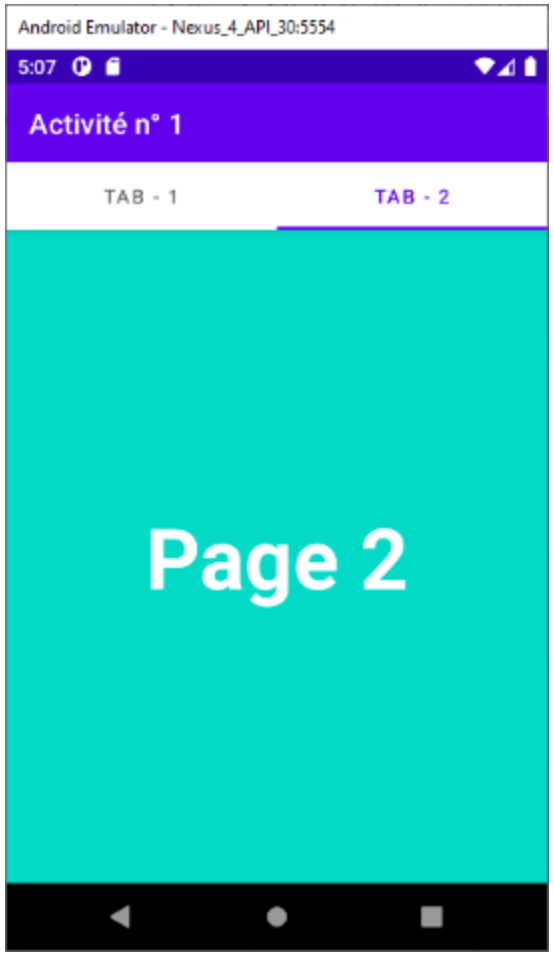
# Activité n° 1

## Android - Passage des données entre fragments

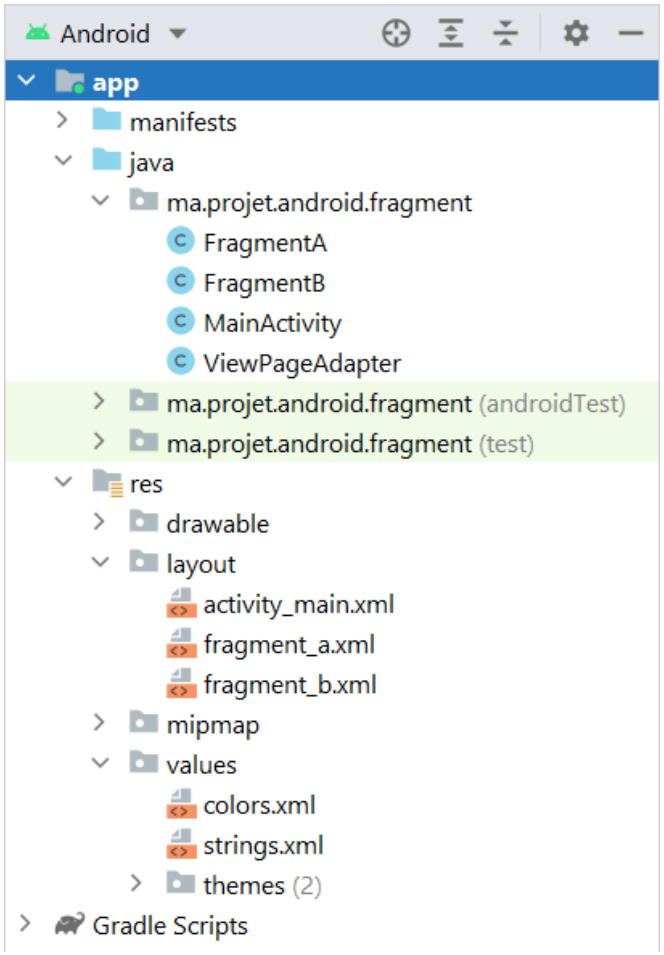
PARTIE 1

### Démo de projet (Version 1)




### Structure de projet



## ViewPager utilisant des fragments dans Android

- **ViewPager** est un layout qui permet à l'utilisateur de feuilleter de gauche à droite des pages de données. On le trouve principalement dans des applications comme Youtube ou Snapchat, où l'utilisateur se déplace de droite à gauche pour passer à un écran. Au lieu d'utiliser des activités, on utilise des fragments. Il est également utilisé pour guider l'utilisateur dans l'application lorsqu'il la lance pour la première fois.
- **Étapes de la mise en œuvre du viewpager :**
    1. Ajouter le widget **ViewPager** au **layout XML** (généralement le **main_layout**).
    2. Créer d'un adaptateur en étendant la classe **FragmentPagerAdapter** ou **FragmentStatePagerAdapter**.

### Remarques

Un adaptateur remplit les pages à l'intérieur du **Viewpager**. **PagerAdapter** est la classe de base qui est étendue par **FragmentPagerAdapter** et **FragmentStatePagerAdapter**. Voici une brève différence entre ces deux classes.

Différence entre **FragmentPagerAdapter** et **FragmentStatePagerAdapter** :

- **FragmentStatePagerAdapter** : conserve en mémoire uniquement le fragment actuel affiché à l'écran. Cette méthode est efficace en termes de mémoire et devrait être utilisée dans les applications comportant des fragments dynamiques (où le nombre de fragments n'est pas fixe).
- **FragmentPagerAdapter** : cet adaptateur doit être utilisé lorsque le nombre de fragments est fixe. Une application qui a 3 onglets qui ne changeront pas pendant l'exécution de l'application.

# Activité n° 1
## Android - Passage des données entre fragments

PARTIE 1

### Structure de la classe ViewPagerAdapter

```
public class ViewPagerAdapter extends FragmentPagerAdapter {

   private final List<Fragment> fragments = new ArrayList<>();
   private final List<String> fragmentTitle = new ArrayList<>();

   public ViewPagerAdapter(@NonNull FragmentManager fm) {
      super(fm);
   }
   public void add(Fragment fragment, String title) {
      fragments.add(fragment);
      fragmentTitle.add(title);
   }
   @NonNull
   @Override
   public Fragment getItem(int position) {
      return fragments.get(position);
   }
   @Override
   public int getCount() {
      return fragments.size();
   }
   @Nullable
   @Override
   public CharSequence getPageTitle(int position) {
      return fragmentTitle.get(position);
   }
}
```

**Description des méthodes :**

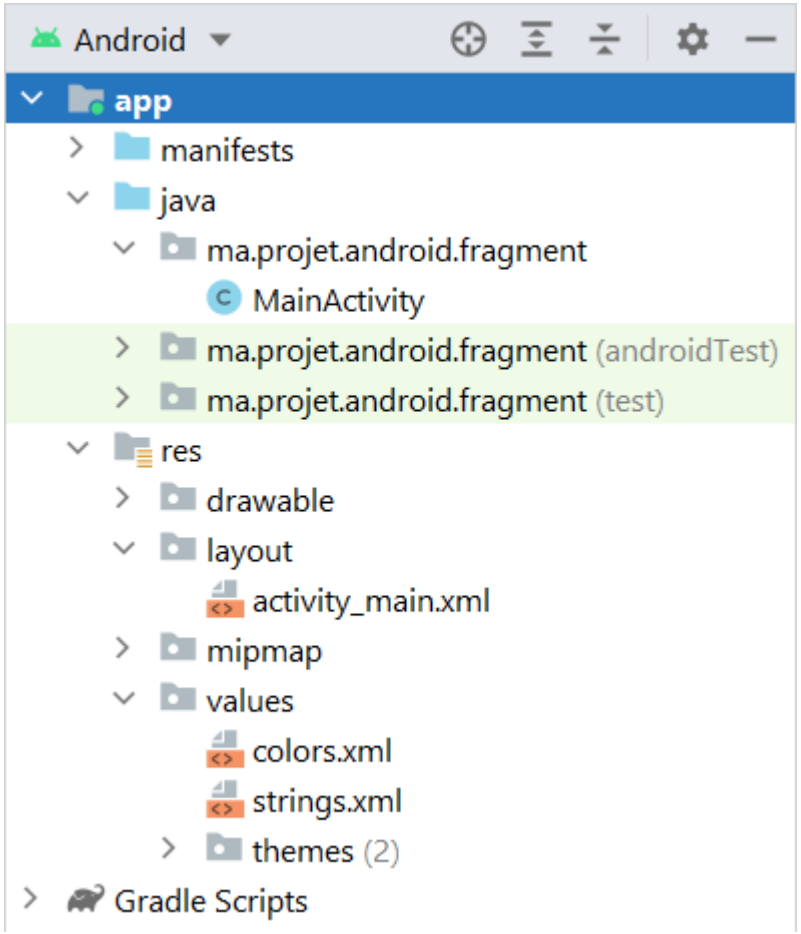
- **getCount()** : renvoie le nombre de fragments à afficher ;
- **getItem(int pos)** : renvoie le fragment à l'indice pos ;
- **ViewPagerAdapter(@NonNull FragmentManager fm)** : l'adaptateur ViewPager doit avoir un constructeur paramétré qui accepte l'instance de **FragmentManager**. Celle-ci est responsable de la gestion des fragments. Un **FragmentManager** gère les fragments dans Android, plus précisément, il gère les transactions entre les fragments. Une transaction est un moyen d'ajouter, de remplacer ou de supprimer des fragments ;
- **getPageTitle(int pos)** : (facultatif) similaire à **getItem(),** renvoie le titre de la page à l'indice pos ;
- **add(Fragment fragment, String title)** : cette méthode est responsable du remplissage des listes de fragments et de **fragmentTitle**, qui contiennent respectivement les fragments et les titres.

La **solution** de cette activé est présentée dans les diapositives qui suivent :

# Activité n° 1
## Android - Passage des données entre fragments

### Etape 1 : créer un nouveau projet

1. Cliquer sur Fichier, puis **New -> New Project**
2. Choisir l'activité **Empty**
3. Sélectionner le langage **Java**
4. Choisir le **SDK minimum** en fonction des besoins




PARTIE 1

# Activité n° 1
## Android - Passage des données entre fragments

PARTIE 1

### Etape 2 : définir le fichier activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.coordinatorlayout.widget.CoordinatorLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   android:layout_width="match_parent"
   android:layout_height="match_parent">

   <com.google.android.material.appbar.AppBarLayout
      android:id="@+id/appbar"
      android:layout_width="match_parent"
      android:layout_height="wrap_content">

      <com.google.android.material.tabs.TabLayout
         android:id="@+id/tab_layout"
         android:layout_width="match_parent"
         android:layout_height="match_parent"
         app:tabGravity="fill"
         app:tabMode="fixed" />

   </com.google.android.material.appbar.AppBarLayout>

   <androidx.viewpager.widget.ViewPager
      android:id="@+id/viewpager"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      app:layout_behavior="@string/appbar_scrolling_view_behavior" />

</androidx.coordinatorlayout.widget.CoordinatorLayout>
```

- Les trois widgets : **AppBarLayout** utilisé pour héberger le **TabLayout** qui est responsable de l'affichage des titres des pages et **ViewPager** layout qui hébergera les différents fragments
- Le code à gauche explique les paramètres importants à définir pour que l'application fonctionne comme prévu
- Dans le **TabLayout**, il faut ajouter le paramètre **tabmode = "fixed"** qui indique au système Android que les onglets de l'application seront en nombre fixe

PARTIE 1

## Etape 3 : créer les fragments

A présent, il faut créer des pages qui sont fragmentées. Pour cette activité, on utilisera deux pages (fragments).

Ajoutez deux fragments vierges au projet, comme illustré dans la structure de projet.

Voici le code des fichiers **FragmentA.java** et **FragmentB.java** respectivement.

```java
public class FragmentA extends Fragment {

    public FragmentA() {
        // required empty public constructor.
    }

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,
                             @Nullable ViewGroup container,
                             @Nullable Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_a, container, false);
    }
}
```

```java
public class FragmentB extends Fragment {

    public FragmentB() {
        // required empty public constructor.
    }

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,
                             @Nullable ViewGroup container,
                             @Nullable Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_b, container, false);
    }
}
```

## Description des méthodes

- **FragmentA()** : constructeur par défaut
- **onCreateView( onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)** : cette méthode est responsable d'inflater (analyser) le fichier XML respectif et de retourner la vue qui est ajoutée à l'adaptateur **ViewPager**
- **onCreate(Bundle savedInstanceState)** : cette méthode est similaire à la méthode **OnCreate()** des activités

**Conception des fichiers XML des pages (Onglet)** : Tous les layouts des fragments ont la même conception. Un **TextView** au centre affiche le nom de la page concernée, le conteneur racine utilisé ici est **FrameLayout** dont l'arrière-plan est fixé sur **@color/teal_200**.

# Activité n° 1
## Android - Passage des données entre fragments

PARTIE 1

### Le fichier fragment_a.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/teal_200"
    tools:context=".FragmentA">

    <!-- TODO: Update blank fragment layout -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:text="Page 1"
        android:textColor="@color/white"
        android:textSize="60sp"
        android:textStyle="bold" />
</FrameLayout>
```

### Le fichier fragment_b.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/teal_200"
    tools:context=".FragmentB">

    <!-- TODO: Update blank fragment layout -->
    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:text="Page 2"
        android:textColor="@color/white"
        android:textSize="60sp"
        android:textStyle="bold" />
</FrameLayout>
```

# Activité n° 1
## Android - Passage des données entre fragments

PARTIE 1

### Etape 4 : créer l'adaptateur ViewPager

```
package ma.projet.android.fragment;


import androidx.annotation.NonNull;
import androidx.annotation.Nullable;
import androidx.fragment.app.Fragment;
import androidx.fragment.app.FragmentManager;
import androidx.fragment.app.FragmentPagerAdapter;

import java.util.ArrayList;
import java.util.List;


public class ViewPagerAdapter extends FragmentPagerAdapter {

    private final List<Fragment> fragments = new ArrayList<>();
    private final List<String> fragmentTitle = new ArrayList<>();

    public ViewPagerAdapter(@NonNull FragmentManager fm) {
        super(fm);
    }

    public void add(Fragment fragment, String title) {
        fragments.add(fragment);
        fragmentTitle.add(title);
    }

    @NonNull
    @Override
    public Fragment getItem(int position) {
        return fragments.get(position);
    }

    @Override
    public int getCount() {
        return fragments.size();
    }

    @Nullable
    @Override
    public CharSequence getPageTitle(int position) {
        return fragmentTitle.get(position);
    }
}
```

## Etape 5 : travailler avec le fichier MainActivity.java

Dans l'activité principale (**MainActivity**), il faut suivre les étapes suivantes :

1. Initialiser le **ViewPager**, le **TabLayout** et l'adaptateur.
2. Ajouter les pages (fragments) ainsi que les titres.
3. Lier le **TabLayout** au **ViewPager** à l'aide de la méthode **setupWithiewPager.**

```
public class MainActivity extends AppCompatActivity {

    private ViewPagerAdapter viewPagerAdapter;
    private ViewPager viewPager;
    private TabLayout tabLayout;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        viewPager = findViewById(R.id.viewpager);

        // configuration de l'adaptateur
        viewPagerAdapter = new ViewPagerAdapter(getSupportFragmentManager());

        // ajouter les fragments
        viewPagerAdapter.add(new FragmentA(), "Tab - 1");
        viewPagerAdapter.add(new FragmentB(), "Tab - 2");

        // définir l'adaptateur
        viewPager.setAdapter(viewPagerAdapter);

        // Les titres de la page (fragment) seront affichés dans le
        // tabLayout, il faut donc définir le visualisateur de page.
        // On utilise la fonction setupWithViewPager().
        tabLayout = findViewById(R.id.tab_layout);
        tabLayout.setupWithViewPager(viewPager);
    }
}
```

PARTIE 1

# Activité n° 1
## Android - Passage des données entre fragments

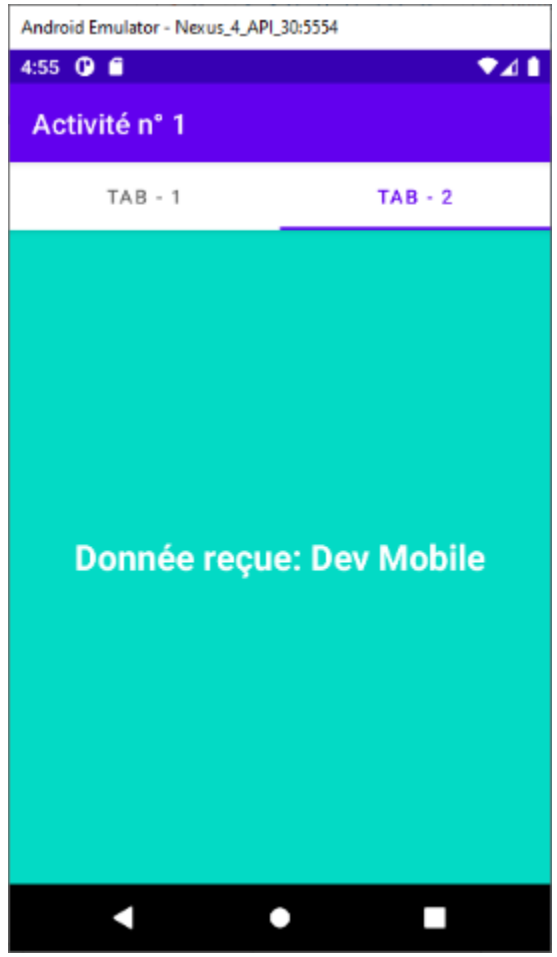
### Version 2

Maintenant, on demande de modifier la version 1, afin de transmettre les données entre les deux fragments.

NB : plusieurs solutions sont possibles.

La **solution** est présentée dans les diapositives qui suivent :

### Démo de projet (Version 2)


PARTIE 1

# Activité n° 1
## Android - Passage des données entre fragments

PARTIE 1

### Le fichier fragment_a.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/teal_200"
    tools:context=".FragmentA">

    <ScrollView
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:fillViewport="true">

        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:orientation="vertical"
            android:padding="15dp">

            <EditText
                android:id="@+id/inMessage"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_above="@+id/btnPassData"
                android:layout_margin="16dp"
                android:hint="Saisir ici" />

            <Button
                android:id="@+id/btnPassData"
                android:layout_width="match_parent"
                android:layout_height="wrap_content"
                android:layout_centerInParent="true"
                android:text="TRANSMETTRE  LES DONNÉES AU FRAGMENT 2" />

        </LinearLayout>
    </ScrollView>
</FrameLayout>
```

### Le fichier fragment_b.xml

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/teal_200"
    tools:context=".FragmentB">


    <!-- TODO: Update blank fragment layout -->
    <TextView
        android:id="@+id/txtData"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:gravity="center"
        android:text="Page 2"
        android:textColor="@color/white"
        android:textSize="25sp"
        android:textStyle="bold" />
</FrameLayout>
```

# Activité n° 1
## Android - Passage des données entre fragments

PARTIE 1

### La classe FragmentA.java

```
public class FragmentA extends Fragment {
    SendMessage sm;

    public FragmentA() {
        // required empty public constructor.
    }

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,
                             @Nullable ViewGroup container,
                             @Nullable Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_a, container, false);
    }

    @Override
    public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        Button btnPassData = (Button) view.findViewById(R.id.btnPassData);
        final EditText inData = (EditText) view.findViewById(R.id.inMessage);
        btnPassData.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View v) {
                sm.sendData(inData.getText().toString().trim());
            }
        });
    }

    @Override
    public void onAttach(Context context) {
        super.onAttach(context);

        try {
            sm = (SendMessage) getActivity();
        } catch (ClassCastException e) {
            throw new ClassCastException("Erreur dans la récupération des données. Veuillez
                                                                    réessayer");
        }
    }

    interface SendMessage {
        void sendData(String message);
    }
}
```

## La classe FragmentB.java

```
public class FragmentB extends Fragment {

    TextView txtData;

    public FragmentB() {
        // required empty public constructor.
    }

    @Override
    public void onCreate(@Nullable Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater, @Nullable ViewGroup container, @Nullable Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_b, container, false);
    }

    @Override
    public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        txtData = (TextView)view.findViewById(R.id.txtData);
    }

    protected void displayReceivedData(String message) {
        txtData.setText("Donnée reçue: "+message);
    }
}
```

# Activité n° 1
## Android - Passage des données entre fragments

PARTIE 1

### Le fichier MainActivity.java

```
public class MainActivity extends AppCompatActivity implements FragmentA.SendMessage{

    private ViewPagerAdapter viewPagerAdapter;
    private ViewPager viewPager;
    private TabLayout tabLayout;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        viewPager = findViewById(R.id.viewpager);

        // configuration de l'adaptateur
        viewPagerAdapter = new ViewPagerAdapter(getSupportFragmentManager());

        // ajouter les fragments
        viewPagerAdapter.add(new FragmentA(), "Tab - 1");
        viewPagerAdapter.add(new FragmentB(), "Tab - 2");

        // définir l'adaptateur
        viewPager.setAdapter(viewPagerAdapter);

        // Les titres de la page (fragment) seront affichés dans le
        // tabLayout, il faut donc définir le visualisateur de page.
        // On utilise la fonction setupWithViewPager().
        tabLayout = findViewById(R.id.tab_layout);
        tabLayout.setupWithViewPager(viewPager);
    }

    @Override
    public void sendData(String message) {
        String tag = "android:switcher:" + R.id.viewpager + ":" + 1;
        FragmentB f = (FragmentB)
                getSupportFragmentManager().findFragmentByTag(tag);
        f.displayReceivedData(message);
    }
}
```


# ACTIVITÉ n° 2

## Mise en œuvre d’une application MVP

### Compétences visées :

- Création d’une application mobile MVP

### Recommandations clés :

- Révision générale du résumé théorique
- Suivre les instructions du TP et organiser le dossier de travail

**5 heures**

# CONSIGNES

## Pour le formateur :

- Rappeler les composants d'un modèle MVP
- Demander aux apprenants de suivre les étapes décrites dans l'activité

## Pour l'apprenant :

- Suivre les étapes indiquées dans l'activité

## Conditions de réalisation :

- Support de résumé théorique accompagnant
- Un ordinateur sur lequel est installée la dernière version stable d'Android Studio

## Critères de réussite :

- Le stagiaire doit être capable de :
  - connaître le rôle de chaque composant dans le modèle MVP
  - réaliser une application mobile MVP

# Activité n° 2
## Mise en œuvre d'une application MVP

### Enoncé

Le développement d'une application Android entraîne un couplage étroit entre l'activité et le mécanisme de traitement des données de l'application. En outre, cela entraîne des difficultés dans la maintenance et la mise à l'échelle de ces applications mobiles. Pour éviter ces problèmes de maintenance, de lisibilité, d'évolutivité et de refactoring des applications, les développeurs préfèrent définir des couches de code bien séparées. En appliquant des modèles d'architecture logicielle, il est possible d'organiser le code de l'application pour séparer les préoccupations.

L'architecture **MVP** (**Model** - **View** - **Presenter**) est l'un des modèles d'architecture les plus populaires et il est efficace pour organiser le projet.

Pour montrer l'implémentation du patron d'architecture **MVP** sur des projets, on demande de développer une application Android à une seule activité. L'application va afficher quelques chaînes de caractères sur la vue (activité) en effectuant une sélection aléatoire dans le modèle. Le rôle de la classe **Presenter** est de garder la logique métier de l'application loin de l'activité.

La **solution** de cette activé est présentée dans les diapositives qui suivent étape par étape :

PARTIE 1

# Activité n° 2

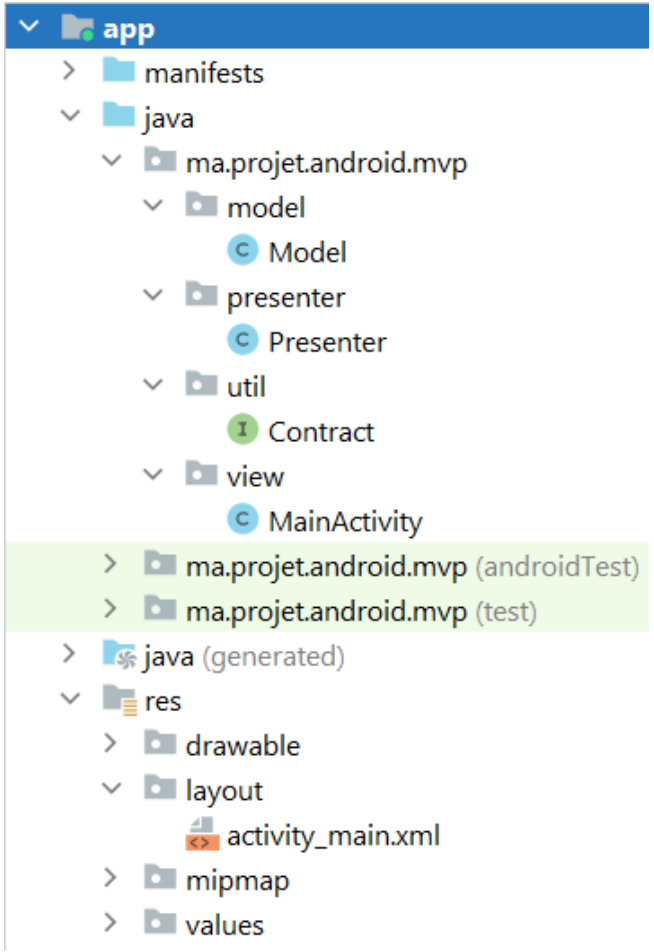
## Mise en œuvre d'une application MVP

PARTIE 1

### Démo



### Structure de projet



# Activité n° 2
## Mise en œuvre d'une application MVP

PARTIE 1

### Etape 1 : créer un projet Android

1. Cliquer sur Fichier, puis **New -> New Project**
2. Choisir l'activité **Empty**
3. Sélectionner le langage Java
4. Choisir le SDK minimum en fonction des besoins

### Etape 2 : modifier le fichier String.xml

```
<resources>
   <string name="app_name">Architecture MVP</string>
   <string name="buttonText">Afficher le cours suivant</string>
   <string name="heading">MVP Architecture Pattern</string>
   <string name="subHeading">Cours en ligne d\'informatique </string>
   <string name="description">Description du cours</string>
</resources>
```

PARTIE 1

## Etape 3 : créer le fichier activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.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"
    android:background="#168BC34A"
    android:padding="10dp"
    tools:context=".view.MainActivity">

    <!-- TextView pour afficher le titre de l'activité -->
    <TextView
        android:id="@+id/textView3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/heading"
        android:textAlignment="center"
        android:textColor="@android:color/holo_blue_light"
        android:textSize="30sp"
        android:textStyle="bold"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.498"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.060000002" />

    <!-- TextView pour afficher le sous-titre de l'activité -->
    <!-- TextView pour afficher la chaîne aléatoire -->

    <TextView
        android:id="@+id/textView2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/subHeading"
        android:textAlignment="center"
        android:textColor="@android:color/holo_blue_light"
        android:textSize="24sp"
        android:textStyle="bold"
        app:layout_constraintBottom_toTopOf="@+id/button"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.498"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.364" />

    <TextView
        android:id="@+id/textView"
        android:layout_width="411dp"
        android:layout_height="wrap_content"
        android:gravity="center"
        android:padding="8dp"
        android:text="@string/description"
        android:textAlignment="center"

        android:textAppearance="?android:attr/textAppearanceSearchResultTitle"
        app:layout_constraintBottom_toTopOf="@+id/button"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView2"
        app:layout_constraintVertical_bias="0.508" />
```

## Etape 3 : créer le fichier activity_main.xml

```
<!-- Bouton pour afficher la chaîne aléatoire suivante -->
<Button
   android:id="@+id/button"
   android:layout_width="0dp"
   android:layout_height="wrap_content"
   android:layout_margin="@android:dimen/notification_large_icon_height"
   android:background="#4CAF50"
   android:text="@string/buttonText"
   android:textAllCaps="true"
   android:textColor="@android:color/background_light"
   android:textSize="20sp"
   android:textStyle="bold"
   app:layout_constraintBottom_toBottomOf="parent"
   app:layout_constraintLeft_toLeftOf="parent"
   app:layout_constraintRight_toRightOf="parent"
   app:layout_constraintTop_toTopOf="parent"
   app:layout_constraintVertical_bias="0.79" />

<!-- Barre de progression à afficher avant d'afficher la chaîne suivante -->
<ProgressBar
   android:id="@+id/progressBar"
   style="?android:attr/progressBarStyleLarge"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:visibility="gone"
   app:layout_constraintBottom_toBottomOf="parent"
   app:layout_constraintLeft_toLeftOf="parent"
   app:layout_constraintRight_toRightOf="parent"
   app:layout_constraintTop_toTopOf="parent" />

  <ImageView
     android:id="@+id/imageView"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     app:layout_constraintBottom_toBottomOf="parent"
     app:layout_constraintEnd_toEndOf="parent"
     app:layout_constraintStart_toStartOf="parent"
     app:layout_constraintTop_toBottomOf="@+id/button"
     app:layout_constraintVertical_bias="1.0"
     app:srcCompat="@drawable/ic_launcher_foreground" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

**Mode Design :**


PARTIE 1

## Étape 4 : définir le fichier d'interface du contrat pour le modèle, la vue et le présentateur

Pour établir la communication entre le **View-Presenter** et le **Presenter-Model**, une interface est nécessaire.
Cette interface contiendra toutes les méthodes abstraites qui seront définies ultérieurement dans les classes Vue, Modèle et Présentateur.

```
public interface Contract {
    interface View {
        // méthode pour afficher la barre de progression
        // lorsque les détails du prochain parcours aléatoire
        // sont en train d'être récupérés.
        void showProgress();

        // méthode pour cacher la barre de progression
        // lorsque les détails du prochain cours aléatoire
        // sont récupérés.
        void hideProgress();


        // méthode pour définir un texte aléatoire texte aléatoire
        // sur le TextView.
        void setString(String string);
    }

    interface Model {
        // interface imbriquée à appeler
        interface OnFinishedListener {
            // fonction à appeler
            // une fois que le Handler de la classe Model
            // a terminé son exécution
            void onFinished(String string);
        }
        void getNextCourse(Contract.Model.OnFinishedListener onFinishedListener);
    }

    interface Presenter {
        // méthode à appeler quand on
        // clique sur le bouton
        void onButtonClick();
        // méthode pour détruire
        // le cycle de vie de MainActivity
        void onDestroy();
    }
}
```

PARTIE 1

PARTIE 1

## Étape 5 : créer la classe Modèle

Créer une nouvelle classe nommée Model pour séparer toutes les données de type chaîne et les méthodes pour récupérer ces données.
Cette classe ne connaîtra pas l'existence de la classe **View**.

```
public class Model implements Contract.Model {

    // liste des chaînes de caractères (des noms des cours) à partir desquelles
    // des chaînes aléatoires seront sélectionnées
    // pour les afficher dans l'activité
    private List<String> arrayList = Arrays.asList(
            "DIA_DEV_TS-09 : Acquérir les bases de développement Android"
            "DIA_DEV_TS-10 : Programmer en KOTLIN"
            "DIA_DEV_TS-11 : Découvrir la gestion de projet"
            "DIA_DEV_TS-12 : S'initier aux composants et modèle d'une application Android"
            "DIA_DEV_TS-13 : Développer des interfaces utilisateurs sous Android"
            "DIA_DEV_TS-14 : Elaborer une application Android sécurisée"
            "DIA_DEV_TS-15 : Découvrir les bases de développement des applications IOS"
            "DIA_DEV_TS-16 : Découvrir les bases de développement multiplateforme"

    @Override
    // cette méthode sera invoquée lorsque
    // l'utilisateur clique sur le bouton
    // et il faudra un délai de
    // 1200 millisecondes pour afficher les détails du prochain cours
    public void getNextCourse(final OnFinishedListener listener) {
        new Handler().postDelayed(new Runnable() {
            @Override
            public void run() {
                listener.onFinished(getRandomString());
            }
        }, 1200);
    }
    // méthode pour sélectionner une chaîne aléatoire
    // dans la liste des chaînes de caractères
    private String getRandomString() {
        Random random = new Random();
        int index = random.nextInt(arrayList.size());
        return arrayList.get(index);
    }
}
```

## Étape 6 : créer la classe Presenter

Les méthodes de cette classe contiennent la logique métier de base qui décidera quoi et comment l'afficher. Elle déclenche la classe **View** pour apporter les changements nécessaires à l'interface utilisateur.

```
public class Presenter implements Contract.Presenter,
      Contract.Model.OnFinishedListener {

   // création d'un objet de l'interface View
   private Contract.View mainView;

   // créer un objet de l'interface du modèle
   private Contract.Model model;

   // l'instanciation des objets de l'interface de vue et de modèle
   public Presenter(Contract.View mainView, Contract.Model model) {
      this.mainView = mainView;
      this.model = model;
   }

   @Override
   // opérations à effectuer
   // sur le clic du bouton
   public void onButtonClick() {
      if (mainView != null) {
         mainView.showProgress();
      }
      model.getNextCourse(this);
   }

   @Override
   public void onDestroy() {
      mainView = null;
   }

   @Override
   // méthode pour retourner la chaîne de caractères
   // qui sera affichée dans le
   // TextView qui affiche le détail du cours
   public void onFinished(String string) {
      if (mainView != null) {
         mainView.setString(string);
         mainView.hideProgress();
      }
   }
}
```

## Étape 7 : définir les fonctionnalités de la vue dans le fichier MainActivity

La classe **View** est responsable de la mise à jour de l'interface utilisateur en fonction des changements déclenchés par la couche **Presenter**. Les données fournies par le modèle seront utilisées par la classe **View** et les changements appropriés seront effectués dans l'activité.

```
public class MainActivity
      extends AppCompatActivity
      implements Contract.View {

   // création d'un objet de la classe TextView
   private TextView textView;

   // création d'un objet de la classe Button
   private Button button;

   // création d'un objet de la classe ProgressBar
   private ProgressBar progressBar;

   // création d'un objet de l'interface Presenter dans le contrat
   Contract.Presenter presenter;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      // attribution de l'ID du TextView
      textView = findViewById(R.id.textView);

      // attribution de l'ID du bouton
      button = findViewById(R.id.button);

      // attribution de l'ID de la ProgressBar
      progressBar = findViewById(R.id.progressBar);

      // instanciation d'un objet de l'interface du présentateur
      presenter = new Presenter(this, new Model());

      // opérations à effectuer lorsque
      // l'utilisateur clique sur le bouton
      button.setOnClickListener(new View.OnClickListener() {
         @Override
         public void onClick(View v) {
            presenter.onButtonClick();
         }
      });
   }

   @Override
   protected void onResume() {
      super.onResume();
   }

   @Override
   protected void onDestroy() {
      super.onDestroy();
      presenter.onDestroy();
   }
```

## Étape 7 : définir les fonctionnalités de la vue dans le fichier MainActivity

```
    @Override
    // méthode pour afficher le TextView de détail du cours
    public void showProgress() {
        progressBar.setVisibility(View.VISIBLE);
        textView.setVisibility(View.INVISIBLE);
    }

    @Override
    // méthode pour masquer le TextView de détail du cours
    public void hideProgress() {
        progressBar.setVisibility(GONE);
        textView.setVisibility(View.VISIBLE);
    }

    @Override
    // Méthode pour définir une chaîne aléatoire
    // dans le TextView de détail du cours
    public void setString(String string) {
        textView.setText(string);
    }
}
```

**Avantages de l'architecture MVP :**

- Pas de relations conceptuelles entre les composants Android.
- Facilite la maintenance et le test du code car le modèle, la vue et la couche de présentation de l'application sont séparés

**Inconvénients de l'architecture MVP :**

- Les développeurs qui ne respectent pas le principe de responsabilité unique pour décomposer le code ont tendance à faire de la couche présentatrice une énorme classe omnisciente.

# Activité n° 2
## Mise en œuvre d'une application MVP

### Challenge

On demande de refaire le même travail  réalisé avec le langage Java cette fois-ci avec le langage **Kotlin**.

PARTIE 1

# ACTIVITÉ n° 3

## Mise en œuvre d’une application MVVM

### Compétences visées :

- Création d’une application mobile MVVM

### Recommandations clés :

- Révision générale du résumé théorique
- Suivre les instructions du TP et organiser le dossier de travail

**5 heures**

# CONSIGNES

## Pour le formateur :

- Rappeler les composants d’un modèle MVVM
- Demander aux apprenants de suivre les étapes décrites dans l’activité

## Pour l’apprenant :

- Suivre les étapes indiquées dans l’activité

## Conditions de réalisation :

- Support de résumé théorique accompagnant
- Un ordinateur sur lequel est installée la dernière version stable d'Android Studio

## Critères de réussite :

- Le stagiaire doit être capable de :
  - connaître le rôle de chaque composant dans le modèle MVVM
  - réaliser une application mobile MVVM

# Activité n° 3

## Mise en œuvre d'une application MVVM

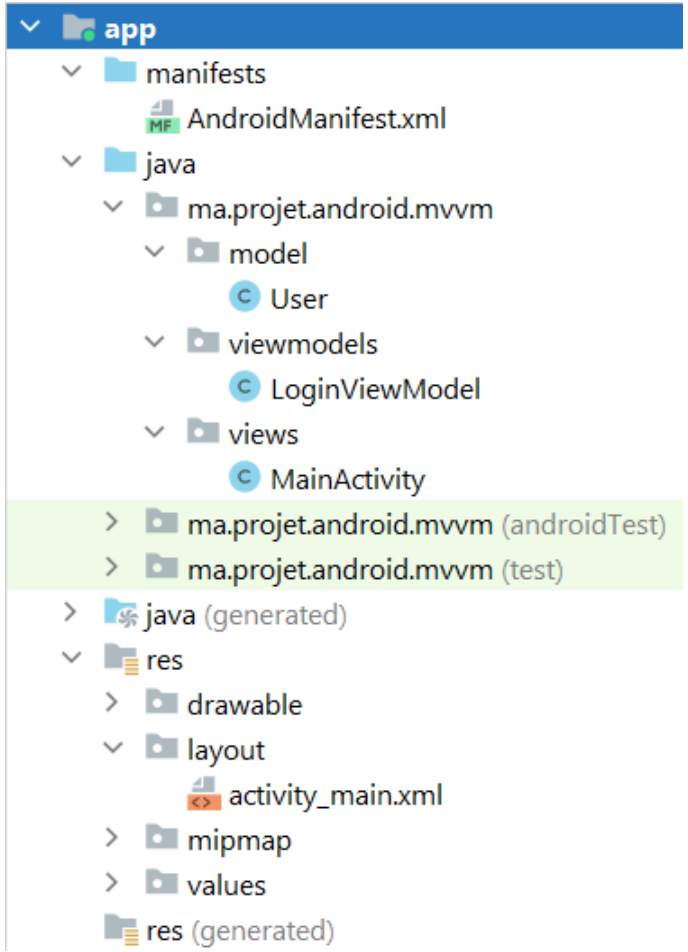
PARTIE 1

### Enoncé

Dans cette activité nous allons développer une d'application Android avec une seule activité pour l'authentification, afin de montrer la mise en œuvre du modèle d'architecture **MVVM**. L'application demande à l'utilisateur de saisir son identifiant et son mot de passe. En fonction des entrées reçues, le **ViewModel** notifie la vue et affiche message dans un toast. Le **ViewModel** n'aura pas de référence à la **vue**.

La **solution** de cette activé est présentée dans les diapositives qui suivent étape par étape :

### Structure de projet



# Activité n° 3

## Mise en œuvre d'une application MVVM

PARTIE 1

### Démo



# Activité n° 3
## Mise en œuvre d'une application MVVM

PARTIE 1

### Étape 1 : créer un nouveau projet

1. Cliquer sur File, puis **New -> New Project**
2. Choisir l'activité **Empty**
3. Sélectionner comme langage **Java**
4. Choisir le **SDK minimum** en fonction des besoins

### Étape 2 : modifier le fichier String.xml

Toutes les chaînes de caractères qui sont utilisées dans l'activité sont listées dans ce fichier.

```
<resources>
   <string name="app_name">Architecture MVVM"</string>
   <string name="heading">MVVM Architecture Pattern</string>
   <string name="email_hint">Entrez votre identifiant email</string>
   <string name="password_hint">Entrez votre mot de passe</string>
   <string name="button_text">Connexion</string>
</resources>
```

## Étape 3 : créer la classe Modèle

Créez une nouvelle classe nommée User qui contiendra l'email et le mot de passe saisi par l'utilisateur. Voici le code pour implémenter la classe User appropriée.

```
public class User {
   private String email;
   private String password;

   // constructeur pour initialiser
   // les variables
   public User(String email, String password) {
      this.email = email;
      this.password = password;
   }

   // méthodes getter et setter
   // pour la variable email
   public void setEmail(String email) {
      this.email = email;
   }


   public String getEmail() {
      return email;
   }

   // méthodes getter et setter
   // pour la variable password
   public void setPassword(String password) {
      this.password = password;
   }


   public String getPassword() {
      return password;
   }
}
```

PARTIE 1

## Étape 4 : modifier le fichier activity_main.xml

```xml
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:bind="http://schemas.android.com/tools">

    <!-- Lier l'objet du ViewModel au layout XML -->
    <data>
        <variable
            name="viewModel"
            type="ma.projet.android.mvvm.viewmodels.LoginViewModel" />
    </data>

    <!-- Fournir une mise en page linéaire pour les composants de l'activité -->
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_gravity="center"
        android:layout_margin="8dp"
        android:orientation="vertical">

        <!-- TextView pour le titre de l'activité -->
        <TextView
            android:id="@+id/textView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/heading"
            android:textAlignment="center"
            android:textColor="@color/purple_200"
            android:textSize="30sp"
            android:textStyle="bold" />

        <!-- Champ EditText pour l'Email -->
        <EditText
            android:id="@+id/inEmail"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginStart="10dp"
            android:layout_marginTop="60dp"
            android:layout_marginEnd="10dp"
            android:layout_marginBottom="20dp"
            android:hint="@string/email_hint"
            android:inputType="textEmailAddress"
            android:padding="8dp"
            android:text="@={viewModel.userEmail}" />

        <!-- Champ EditText pour le mot de passe -->
        <EditText
            android:id="@+id/inPassword"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginStart="10dp"
            android:layout_marginEnd="10dp"
            android:hint="@string/password_hint"
            android:inputType="textPassword"
            android:padding="8dp"
            android:text="@={viewModel.userPassword}" />

        <!-- Bouton de connexion de l'activité -->
        <Button
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginStart="20dp"
            android:layout_marginTop="60dp"
            android:layout_marginEnd="20dp"
            android:background="#4CAF50"
            android:onClick="@{()-> viewModel.onButtonClicked()}"
            android:text="@string/button_text"
            android:textColor="@android:color/background_light"
            android:textSize="20sp"
            android:textStyle="bold"
            bind:toastMessage="@{viewModel.toastMessage}" />

        <ImageView
            android:id="@+id/imageView"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="70dp"
            app:srcCompat="@drawable/ic_launcher_foreground" />

    </LinearLayout>
</layout>
```

PARTIE 1

## Étape 5 : créer la classe ViewModel

Cette classe contient toutes les méthodes qui doivent être appelées dans le layout de l'application. La classe **ViewModel** s'étendra à **BaseObservable** car elle convertit les données en flux et notifie la vue lorsque la propriété du message de toast est modifiée.

```
public class LoginViewModel extends BaseObservable {
   // création d'un objet de la classe Model
   private User user;

   // variables de chaîne pour
   // les messages de toast
   private String successMessage = "Login successful";
   private String errorMessage = "Email or Password is not valid";

   @Bindable
   // variable de chaîne pour
   // le message du toast
   private String toastMessage = null;

   // méthodes getter et setter
   // pour le message du toast
   public String getToastMessage() {
      return toastMessage;
   }
   private void setToastMessage(String toastMessage) {
      this.toastMessage = toastMessage;
      notifyPropertyChanged(BR.toastMessage);
   }

   // méthodes getter et setter
   // pour la variable email
   @Bindable
   public String getUserEmail() {
      return user.getEmail();
   }

   public void setUserEmail(String email) {
      user.setEmail(email);
      notifyPropertyChanged(BR.userEmail);
   }

   // méthodes getter et setter
   // pour la variable password
   @Bindable
   public String getUserPassword() {
      return user.getPassword();
   }

   public void setUserPassword(String password) {
      user.setPassword(password);
      notifyPropertyChanged(BR.userPassword);
   }
```

## Étape 5 : Créer la classe ViewModel

```
// constructeur de la classe LoginViewModel
public LoginViewModel() {
   // instanciation d'un objet de
   // classe de modèle
   user = new User("","");
}

// actions à effectuer
// lorsque l'utilisateur clique sur
// le bouton Connexion
public void onButtonClicked() {
   if (isValid())
      setToastMessage(successMessage);
   else
      setToastMessage(errorMessage);
}

// méthode permettant de vérifier
// que les champs variables ne doivent
//  pas être laissés vides par l'utilisateur
public boolean isValid() {
   return !TextUtils.isEmpty(getUserEmail()) &&
            Patterns.EMAIL_ADDRESS.matcher(getUserEmail()).matches()
              && getUserPassword().length() > 5;
   }
}
```

## Étape 6 : Définir les fonctionnalités de la vue dans le fichier MainActivity

La classe **View** est responsable de la mise à jour de l'interface utilisateur de l'application. En fonction des changements dans le message toast fourni par **ViewModel**, l'adaptateur de liaison déclenchera la couche **View**. Le setter du message **Toast** notifiera **l'observateur** (**View**) des changements dans les données. Après cela, la vue prendra les mesures appropriées.

```
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // ViewModel met à jour le Modèle
        // après avoir observé les changements dans la vue
        // Le modèle met également à jour la vue
        // via le ViewModel
        ActivityMainBinding activityMainBinding = DataBindingUtil.setContentView(this,
                                                  R.layout.activity_main);
        activityMainBinding.setViewModel(new LoginViewModel());
        activityMainBinding.executePendingBindings();

    }

    // toute modification de l'attribut toastMessage
    // définie sur le bouton avec le préfixe bind
    // invoque cette méthode
    @BindingAdapter({"toastMessage"})
    public static void runMe( View view, String message) {
        if (message != null)
            Toast.makeText(view.getContext(), message, Toast.LENGTH_SHORT).show();
    }
}
```

## Avantages et inconvénients

### Avantages de l'architecture MVVM :

- Améliore la réutilisabilité du code
- Tous les modules sont indépendants, ce qui améliore la testabilité de chaque couche
- Rend les fichiers du projet maintenables et faciles à modifier

### Inconvénients de l'architecture MVVM :

- Ce modèle de conception n'est pas idéal pour les petits projets
- Si la logique de liaison des données est trop complexe, le débogage de l'application sera un peu plus difficile

## Challenge

On demande de refaire le même travail réalisé avec le langage Java cette fois-ci avec le langage **Kotlin**.

# PARTIE 2

## Créer des tâches asynchrones et des tâches de fond

**Dans ce module, vous allez :**

- Planifier des tâches avec WorkManager
- Utiliser un  job scheduler
- Manipuler un thread

16 heures

# ACTIVITÉ n° 1

## Mise en œuvre d’une application de récupération des données depuis un fichier en utilisant le WorkManager

### Compétences visées :

- Planification d'une tâche avec WorkManager

### Recommandations clés :

- Révision générale du résumé théorique
- Suivre les instructions du TP et organiser le dossier de travail

**5 heures**

# CONSIGNES

## 1. Pour le formateur :

- Rappeler le fonctionnement de WorkManager;
- Demander aux apprenants de suivre les étapes décrites dans l'activité.

## 2. Pour l'apprenant :

- Suivre les étapes indiquées dans l'activité.

## 3. Conditions de réalisation :

- Support de résumé théorique accompagnant.
- Un ordinateur sur lequel est installée la dernière version stable d'Android Studio.

## 4. Critères de réussite :

- Le stagiaire doit être capable de :
  - planifier une tâche simple ;
  - afficher l'état du work (travail) dans logcat ;
  - connaître les paramètres d'entrée et de sortie ;
  - connaître les contraintes liées à un work (travail).

# ACTIVITÉ n° 1
## Récupération des données depuis un fichier en utilisant le WorkManager

PARTIE 2

### Enoncé

L'objectif de cette activité est de récupérer les données stockées dans un fichier texte en utilisant le **WorkManager**.

Le fichier est stocké dans le dossier assets de l'application mobile.

On demande d'afficher ligne par ligne dans le **logcat**.

**Fichier file.txt**

Je suis un développeur des applications Mobiles

Je développe avec :

Java

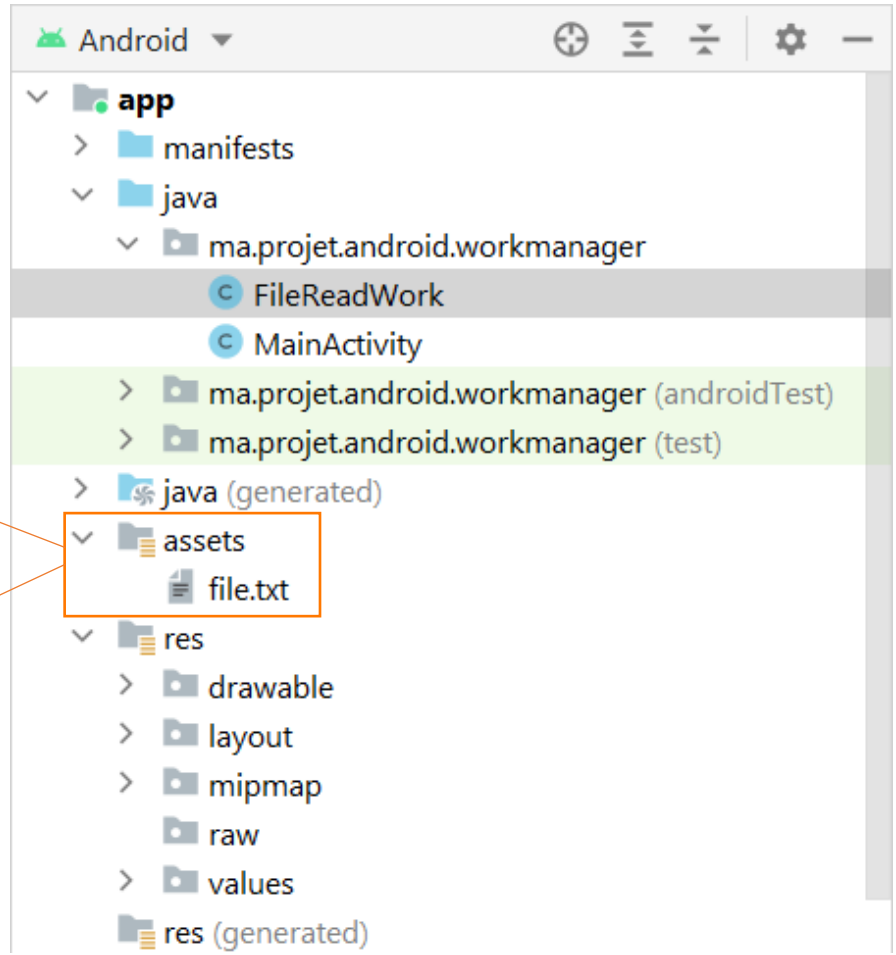
Kotlin

Dart

Swift

La **solution** de cette activité est présentée dans les diapositives qui suivent, étape par étape :

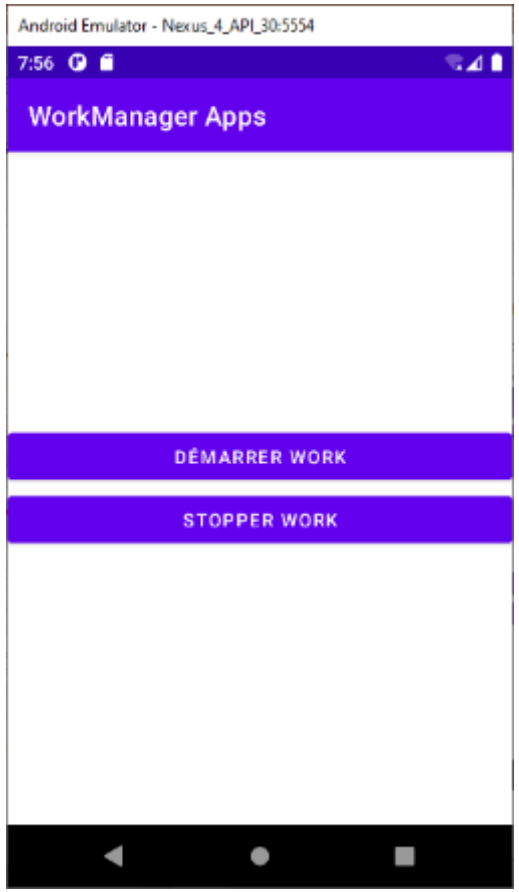
### Structure de projet



# ACTIVITÉ n° 1
## Récupération des données depuis un fichier en utilisant le WorkManager

### Etape 1 : développer l'interface graphique

1. Créer un projet Android, ensuite développer l'interface suivante :



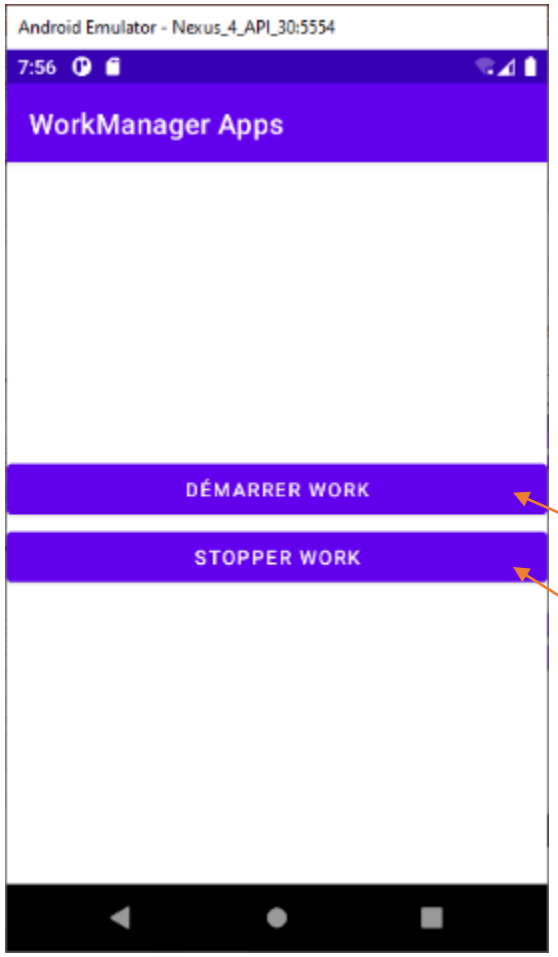

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:layout_centerInParent="true"
    android:gravity="center"
    tools:context=".MainActivity">


    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="startWork"
        android:text="Démarrer Work" />

    <Button
        android:id="@+id/button2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="stopWork"
        android:text="Stopper Work" />
</LinearLayout>
```

PARTIE 2

# ACTIVITÉ n° 1
## Récupération des données depuis un fichier en utilisant le WorkManager

PARTIE 2

### Etape 1 : développer l'interface graphique




### La classe MainActivity.java

2. Dans la classe **MainActivity**, ajouter les deux méthodes :

- **startWork (View v)**
- **stopWork (View v)**

Ensuite, associer respectivement les deux méthodes aux boutons « Démarrer Work » et « Stopper Work ».

```
public class MainActivity extends AppCompatActivity {

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

   }

   public void stratWork(View v) {
         //code pour démarrer le Work
   }

   public void stopWork(View v) {
         //code pour arrêter le Work
   }
}
```

# ACTIVITÉ n° 1
## Récupération des données depuis un fichier en utilisant le WorkManager

PARTIE 2

### Etape 2 : ajouter les dépendances

Ajouter la dépendance de **WorkManager** dans **build.gradle** (Module ) :

```
def work_version = "2.7.1"
implementation "androidx.work:work-runtime:$work_version«
```

Ensuite, synchroniser le projet.

### Etape 3 : créer la classe FileReadWork

Créer la classe **FileReadWork** qui hérite de la classe **Worker**. Ensuite, redéfinir la méthode **doWork()** et le constructeur de la classe.

```
public class FileReadWork extends Worker {
   private static final String TAG = "FileReadWork";
   private Context context;
   private WorkerParameters workerParameters;

   public FileReadWork(@NonNull Context context,
                       @NonNull WorkerParameters workerParams) {
      super(context, workerParams);
      this.context = context;
      this.workerParameters = workerParams;
   }
   @NonNull
   @Override
   public Result doWork() {
       //A compléter
       return null;
   }

}
```

# ACTIVITÉ n° 1
## Récupération des données depuis un fichier en utilisant le WorkManager

PARTIE 2

### Méthode doWork()

```
public Result doWork() {
   InputStream stream = null;
   try {
      stream = context.getAssets().open("file.txt");
      InputStreamReader reader = new InputStreamReader(stream);
      StringBuilder text = new StringBuilder();
      BufferedReader br = new BufferedReader(reader);
      String line;
      while ((line = br.readLine()) != null) {
         text.append(line);
         text.append('\n');
         Log.d(TAG, line.toString());
      }
      br.close();
   } catch (IOException e) {
      e.printStackTrace();
      Log.d(TAG, e.getMessage());
      return Result.failure();
   }
   return Result.success();
}
```

### Etape 4 : compléter les méthodes de la classe MainActivity

1. Dans la classe **MainActivity**, créer deux objets **WorkManager** et **WorkRequest** :

```
private WorkManager workManager;
private WorkRequest workRequest;
```

2. Dans la méthode **OnCreate()** initialiser les deux objets :

```
workManager = WorkManager.getInstance(getApplicationContext());
workRequest = new PeriodicWorkRequest.Builder(FileReadWork.class, 15,
TimeUnit.MINUTES ).build();
```

3. Dans la méthode **startWork ()** démarrer le Work (placer la requête dans la file d'attente):

```
workManager.enqueue(workRequest);
```

4. Dans la méthode **stopWork()** annuler l'exécution de Work :

```
workManager.cancelWorkById(workRequest.getId());
```

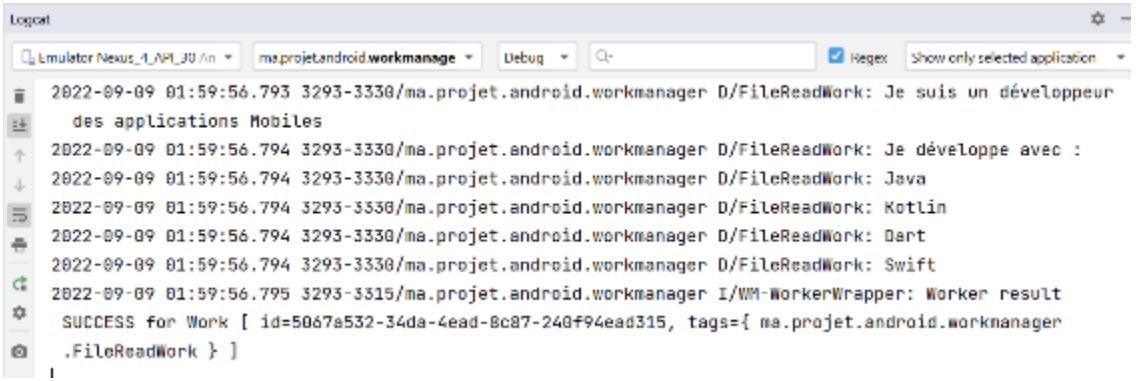
# ACTIVITÉ n° 1
## Récupération des données depuis un fichier en utilisant le WorkManager

PARTIE 2

### MainActivity.java

```
public class MainActivity extends AppCompatActivity {

    private WorkManager workManager;
    private WorkRequest workRequest;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);


        workManager = WorkManager.getInstance(getApplicationContext());
        workRequest = new PeriodicWorkRequest.Builder(FileReadWork.class, 15,
                                    TimeUnit.MINUTES ).build();
    }


    public void startWork(View view) {
        workManager.enqueue(workRequest);
    }

    public void stopWork(View view) {
        workManager.cancelWorkById(workRequest.getId());
    }
}
```

### Etape 5 : tester l'application



# ACTIVITÉ n° 2

## Mise en œuvre d’une application de planification d’une tache en background en utilisant le job scheduler

### Compétences visées :

- Utilisation de job scheduler

### Recommandations clés :

- Révision générale du résumé théorique
- Suivre les instructions du TP et organiser le dossier de travail

5 heures

# CONSIGNES

## 1. Pour le formateur :

- Rappeler le fonctionnement de job scheduler
- Expliquer le fonctionnement des notifications dans une application Android
- Demander aux apprenants de suivre les étapes décrites dans l'activité

## 2. Pour l'apprenant :

- Suivre les étapes indiquées dans l'activité

## 3. Conditions de réalisation :

- Support de résumé théorique accompagnant
- Un ordinateur sur lequel est installée la dernière version stable d'Android Studio

## 4. Critères de réussite :

- Le stagiaire doit être capable de :
  - implémenter un JobService
  - construire un objet JobInfo avec des contraintes spécifiques
  - planifier un JobService en fonction de l'objet JobInfo

# ACTIVITÉ n° 2

## Planifier une tâche en background avec le job scheduler

### TP n° 1 : énoncé

L'objectif de cette activité est de mettre en place une simple application Android basée sur le **JobSchduler**.

Dans ce projet, nous allons simuler l'exécution d'une tâche en **arrière plan** avec un seul **Thread**. Ensuite, nous allons définir les contraintes et les conditions pour programmer le **Job**.

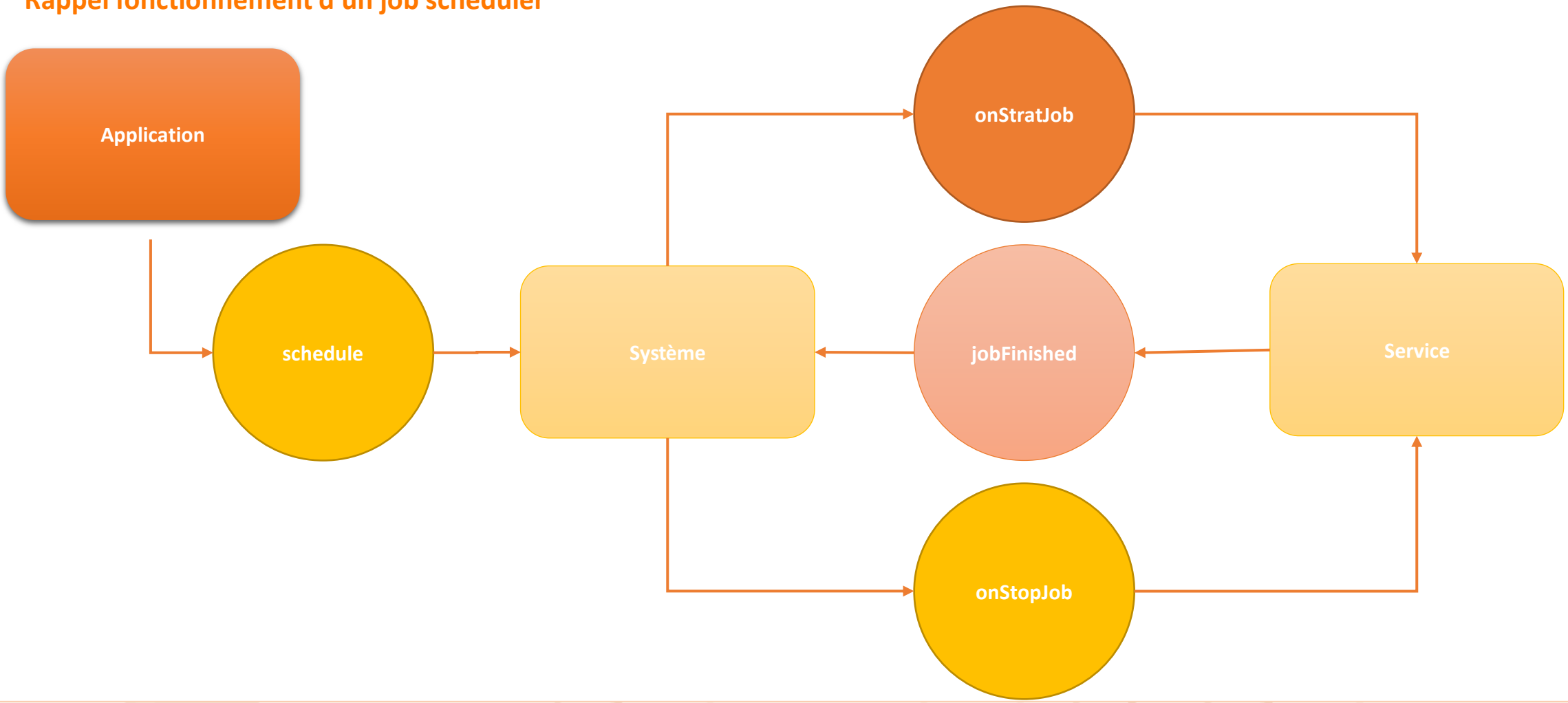
On demande de **suivre les étapes** de l'activité étape par étape.

PARTIE 2

# ACTIVITÉ n° 2

## Planifier une tâche en background avec le job scheduler

### Rappel fonctionnement d'un job scheduler



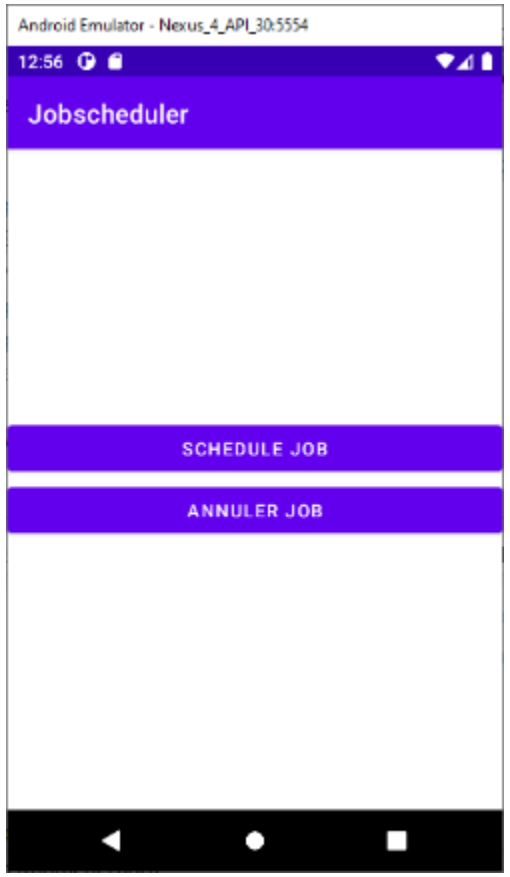

PARTIE 2

# ACTIVITÉ n° 2

## Planifier une tâche en background avec le job scheduler

### Etape 1 : développer l'interface graphique

1. Créer un projet Android avec un **minSDK = 21** (ou plus), ensuite développer l'interface suivante :



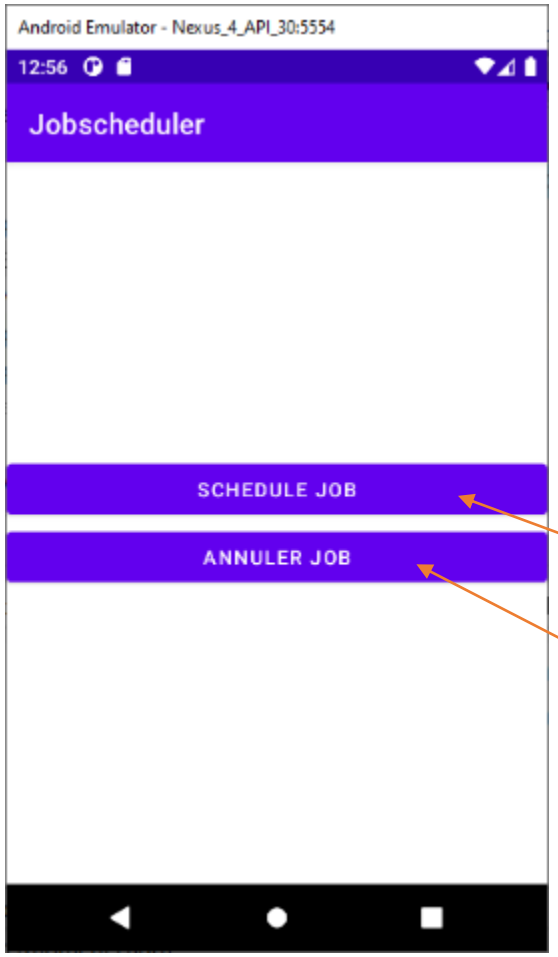

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:layout_centerInParent="true"
   android:gravity="center"
   tools:context=".MainActivity">
   <Button
      android:id="@+id/button"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:onClick="scheduleJob"
      android:text="Schedule Job" />
   <Button
      android:id="@+id/button2"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:onClick="cancelJob"
      android:text="Annuler Job" />
</LinearLayout>
```

PARTIE 2

# ACTIVITÉ n° 2

## Planifier une tâche en background avec le job scheduler

### Etape 1 : développer l'interface graphique




### La classe MainActivity.java

2. Dans la classe **MainActivity**, ajouter les deux méthodes :

- **scheduleJob (View v)**
- **cancelJob (View v)**

Ensuite, associer respectivement les deux méthodes aux boutons « Schedule Job » et « Annuler Job ».

```
public class MainActivity extends AppCompatActivity {

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

   }

   public void scheduleJob(View v) {
        //code pour lancer le Job
   }

   public void cancelJob(View v) {
        //code pour annuler le Job
   }
}
```

PARTIE 2

# ACTIVITÉ n° 2

## Planifier une tâche en background avec le job scheduler

PARTIE 2

### Etape 2 : développer le JobService

1. Créer la classe **ExempleJobService** qui hérite de la classe **JobService**, ensuite redéfinir les deux méthodes **onStartJob()** et **onStopJob ().**

```
public class ExempleJobService extends JobService {
    public static final String TAG = "ExempleJobService";
    public boolean jobCancelled = false;
    @Override
    public boolean onStartJob(JobParameters params) {
        return false;
    }

    @Override
    public boolean onStopJob(JobParameters params) {
        return false;
    }
}
```

2. Déclarer le service dans le fichier AndroidManifest.xml à l'intérieur de la balise <**application**> :

```
<service android:name=".ExempleJobService"
    android:permission="android.permission.BIND_JOB_SERVICE"/>
```

### Etape 3 : développer la tâche d'arrière plan

1. Dans la méthode **onStartJob ()**, créer un Thread permettant d'afficher des nombres de 0 à 9. Le traitement sera réalisé dans la méthode **doBackgroundWork ()**.

```
public boolean onStartJob(JobParameters params) {
    Log.d(TAG, "Job démarré");
    doBackgroundWork(params);
    return true;
}
```

2. Une fois que la méthode **doBackgroundWork()** termine son exécution, appeler la méthode **jobFinished ()** afin de notifier le système que la tâche est terminée.

3. Lors de l'exécution de la méthode **doBackgroundWork()** tester sur la valeur de la variable **JobCancelled**. Si elle est égale à **True** interrompre l'exécution.

# ACTIVITÉ n° 2

## Planifier une tâche en background avec le job scheduler

### Etape 3 : développer la tâche en arrière plan

La méthode **doBackgroundWork ()** :

```
private void doBackgroundWork(JobParameters params) {
   new Thread(new Runnable() {
      @Override
      public void run() {
         for (int i = 0; i < 10; i++){
            Log.d(TAG, "run : "+i);
            if(jobCancelled)
               return;
            try {
               Thread.sleep(1000);
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
         }
         Log.d(TAG, "job terminé");
         jobFinished(params, false);
      }
   }).start();
}
```

### Etape 4 : arrêter le job

1. Dans la méthode **onStopJob ()**, modifier la valeur de la variable **jobCancelled** par **True**. Cela va engendrer l'arrêt de la méthode **doBackgroundWork()**.

```
@Override
public boolean onStopJob(JobParameters params) {
   Log.d(TAG, "Job annulé");
   jobCancelled = true;
   return true;
}
```

PARTIE 2

# ACTIVITÉ n° 2

## Planifier une tâche en background avec le job scheduler

### Etape 5 : planifier le job

Dans la méthode **scheduleJob ()** de la classe **MainActivity.java** :

1. Définir les contraintes suivantes :
   - le job démarre si l'appareil est complétement chargé ;
   - le job nécessite une connectivité réseau ;
   - la tâche du job doit être maintenue lors d'un redémarrage du dispositif ;
   - ce job doit se répéter chaque 15 minutes.

2. Afficher un message pour indiquer à l'utilisateur si le Job a bien démarré ou a échoué.

```
public void scheduleJob(View v) {
   ComponentName componentName = new ComponentName(this, ExempleJobService.class);

   JobInfo info = new JobInfo.Builder(123, componentName)
         .setRequiresCharging(true)
         .setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED)
         .setPersisted(true)
         .setPeriodic(15 * 60 * 1000)
         .build();

   JobScheduler scheduler = (JobScheduler) getSystemService(JOB_SCHEDULER_SERVICE);
   int resultCode = scheduler.schedule(info);
   if (resultCode == JobScheduler.RESULT_SUCCESS) {
      Log.d(TAG, "Job programmé");
   } else {
      Log.d(TAG, "Echec de la programmation de Job");
   }

}
```

PARTIE 2

# ACTIVITÉ n° 2
## Planifier une tâche en background avec le job scheduler

PARTIE 2

### Etape 6 : annuler le job

Dans la méthode **cancelJob()** de la méthode **MainActivity.java**, annuler l'exécution de job.

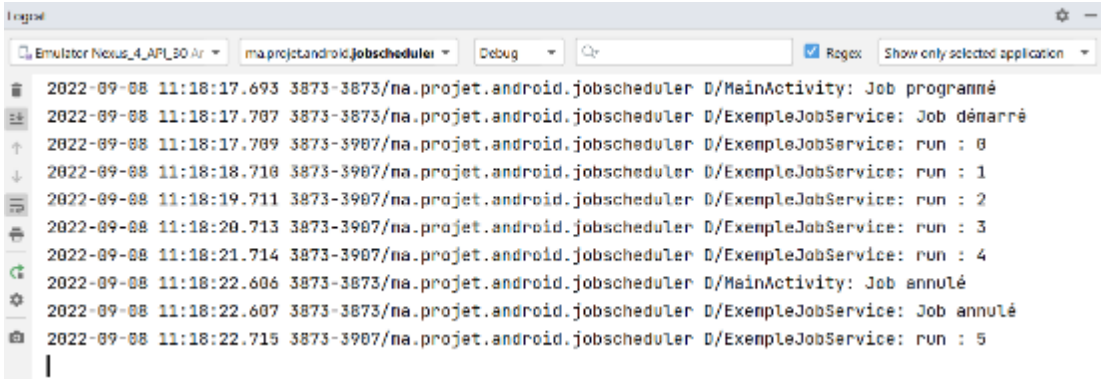
```
public void cancelJob(View v) {
    JobScheduler scheduler = (JobScheduler)
                getSystemService(JOB_SCHEDULER_SERVICE);
    scheduler.cancel(123);
    Log.d(TAG, "Job annulé");
}
```

**Remarques**

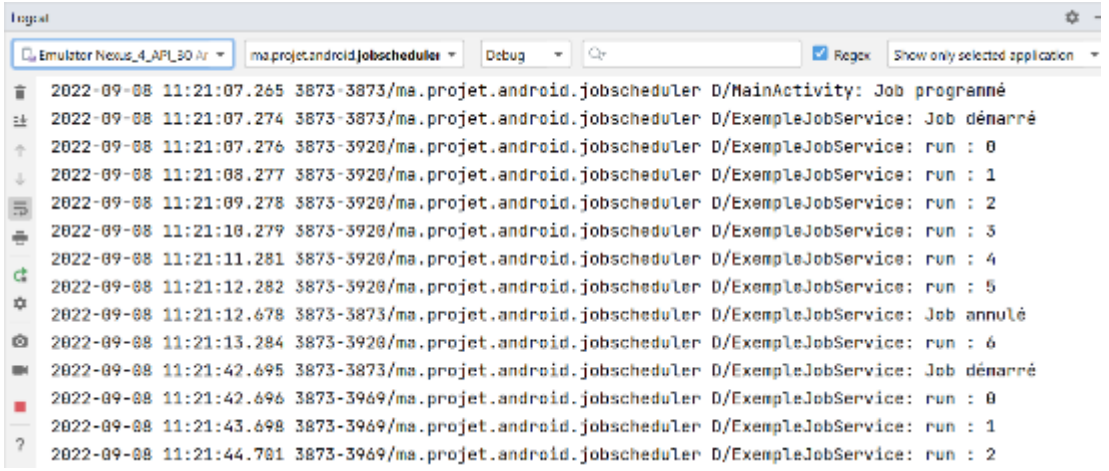
- Si le projet ne fonctionne pas, désactiver la 1ère contrainte.

### Etape 7 : tester l'application

**Scénario 1 :** Démarrer le job, ensuite arrêter le job.




**Scénario 2 :** Démarrer le job, ensuite désactiver et réactiver le Wifi.

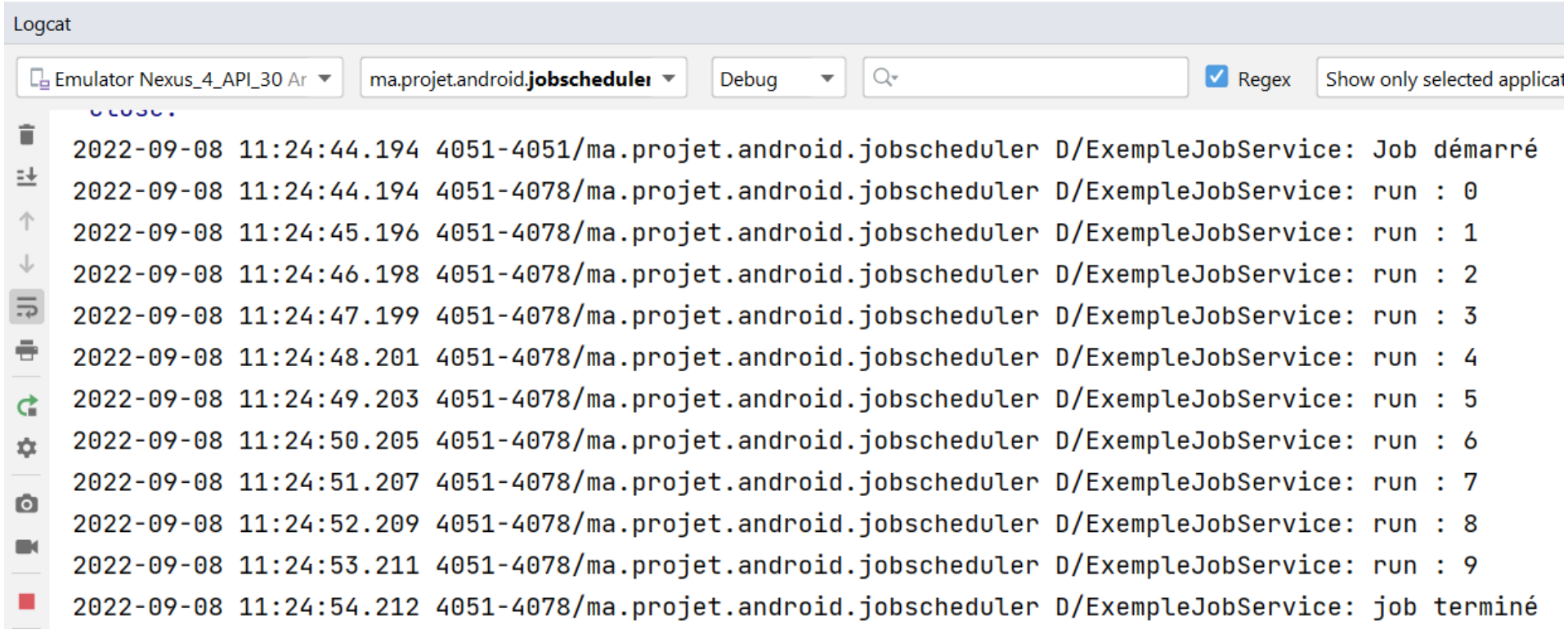


# ACTIVITÉ n° 2

## Planifier une tâche en background avec le job scheduler

### Etape 7 : tester l'application

**Scénario 3 :** démarrer le job, ensuite fermer l'application. Après, lancer l'application à nouveau. Qu'est ce que vous remarquez ?




PARTIE 2

# ACTIVITÉ n° 3

## Planifier une tâche en background avec le job scheduler

PARTIE 2

### TP n° 2 : énoncé

Dans cette activité, vous allez créer une application appelée "Notification Scheduler". L'application fera la démonstration du cadre **JobScheduler** en permettant à l'utilisateur de sélectionner des contraintes et de planifier un travail en arrière plan en utilisant le **JobScheduler**.

Lorsque la tâche sera exécutée, elle enverra une notification à l'utilisateur.

Comme vous l'avez vu dans le cours, pour utiliser le **JobScheduler**, vous avez besoin de deux parties supplémentaires : **JobService** et **JobInfo**. Un objet **JobInfo** contient l'ensemble des conditions qui déclencheront l'exécution du job. Un **JobService** est l'implémentation du **job** qui doit être exécuté dans ces conditions.

On demande de **suivre les étapes** de l'activité étape par étape.

### Tâche 1 : implémenter le JobService

Pour commencer, vous devez créer un service qui sera exécuté au moment déterminé par les conditions. Le **JobService** est automatiquement exécuté par le système, et les seules parties que vous devez implémenter sont :

- **onStartJob() callback :** appelée lorsque le système détermine que la tâche doit être exécutée. La tâche à implémenter doit être effectuée dans cette méthode.
- renvoie un booléen indiquant si le job doit continuer sur un thread séparé. Si la réponse est vraie, le job est déchargé sur un autre thread, et l'application doit appeler **jobFinished()** explicitement dans ce thread pour indiquer que le job est terminé. Si la valeur de retour est false, le framework sait que le travail est terminé à la fin de **onStartJob()** et il appellera automatiquement **jobFinished().**

**Remarques**

- onStartJob() est exécuté sur le thread principal. Par conséquent, toute tâche de longue durée doit être transférée à un autre thread. Dans ce cas, il s'agit simplement de poster une notification, ce qui peut être fait en toute sécurité sur le thread principal.

# ACTIVITÉ n° 3

## Planifier une tâche en background avec le job scheduler

PARTIE 2

### Tâche 1 : implémenter le JobService

**onStopJob() callback :**

- Est appelée si les conditions ne sont plus remplies, ce qui signifie que le travail doit être arrêté.
- Renvoie un booléen qui détermine ce qu'il faut faire si le travail n'est pas terminé. Si la valeur de retour est vraie, le travail sera reprogrammé, sinon, il sera abandonné.

### Etape 1 : créer le projet et le NotificationJobService

Vérifier que le SDK minimum utilisé est l'API 21. Avant l'API 21, **JobScheduler** ne fonctionne pas, car il ne comporte pas certaines des API requises.

1. Utiliser un modèle vide, et créer un nouveau projet appelé "Notification Scheduler"
2. Créer une nouvelle classe Java appelée **NotificationJobService** qui étend **JobService**
3. Ajouter les méthodes requises : **onStartJob()** et **onStopJob()**
4. Dans le fichier AndroidManfiest.xml, enregistrer le **JobService** avec la permission suivante à l'intérieur de la balise <**application**> :

```
<service
    android:name=".NotificationJobService"
    android:permission="android.permission.BIND_JOB_SERVICE"/>
```

# ACTIVITÉ n° 3

## Planifier une tâche en background avec le job scheduler

### Etape 2 : implémenter onStartJob()

1. Ajouter une icône de notification pour la notification "Job Running"
2. Dans la fonction **onStartJob(),** créer un **PendingIntent** pour lancer la **MainActivity** de l'application qui sera utilisée comme intention de contenu pour la notification
3. Dans la fonction **onStartJob(),** construire et délivrer une notification avec les attributs suivants :

| Attribut | Titre |
|---|---|
| Content Title | "Job Service" |
| Content Text | "Votre Job est en cours d'exécution!" |
| Content Intent | contentPendingIntent |
| Small Icon | R.drawable.ic_job_running |
| Priority | NotificationCompat.PRIORITY_HIGH |
| Defaults | NotificationCompat.DEFAULT_ALL |
| AutoCancel | True |

4. Assurer que **onStartJob()** retourne false, parce que tout le travail sera terminé dans ce callback
5. Assurer que **onStopJob()** renvoie **true**, afin que le travail soit re-planifié s'il échoue

PARTIE 2

# ACTIVITÉ n° 3

## Planifier une tâche en background avec le job scheduler

### Etape 2 : implémenter onStartJob()

```
@Override
public boolean onStartJob(JobParameters jobParameters) {

    // Créer le canal de notification.
    createNotificationChannel();

    // Configurer l'intent du contenu de la notification pour lancer l'application
    //  en cas de clic.

    PendingIntent contentPendingIntent = PendingIntent.getActivity
            (this, 0, new Intent(this, MainActivity.class),
                    PendingIntent.FLAG_UPDATE_CURRENT);

    NotificationCompat.Builder builder = new NotificationCompat.Builder
            (this, PRIMARY_CHANNEL_ID)
            .setContentTitle("Job Service")
            .setContentText("Votre Job est en cours d'exécution!")
            .setContentIntent(contentPendingIntent)
            .setSmallIcon(R.mipmap.ic_job_running)
            .setPriority(NotificationCompat.PRIORITY_HIGH)
            .setDefaults(NotificationCompat.DEFAULT_ALL)
            .setAutoCancel(true);

    mNotifyManager.notify(0, builder.build());
    return false;
}
```

### Tâche 2 : mettre en œuvre les conditions de Job

Maintenant que le **JobService** est en place, il est temps d'identifier les critères d'exécution de la tâche. Pour cela, utiliser le composant **JobInfo**.

Créez une série de conditions paramétrées pour l'exécution d'un job à l'aide de différents types de connectivité réseau et de l'état des périphériques.

Pour commencer, créez un groupe de boutons radio pour déterminer le type de réseau requis pour ce job.

#### Mise en œuvre de la contrainte de réseau :

L'une des conditions possibles pour l'exécution d'un job est l'état de la connectivité réseau de l'appareil. Il est possible de limiter l'exécution du **JobService** uniquement lorsque certaines conditions de réseau sont remplies. Les options sont les suivantes :

- **NETWORK_TYPE_NONE** : le job s'exécutera avec ou sans connexion réseau. Il s'agit de la valeur par défaut.
- **NETWORK_TYPE_ANY** : le job sera exécuté tant qu'un réseau (cellulaire, wifi) est disponible.
- **NETWORK_TYPE_UNMETERED** : le travail sera exécuté tant que le dispositif est connecté à un réseau wifi qui n'utilise pas de HotSpot.

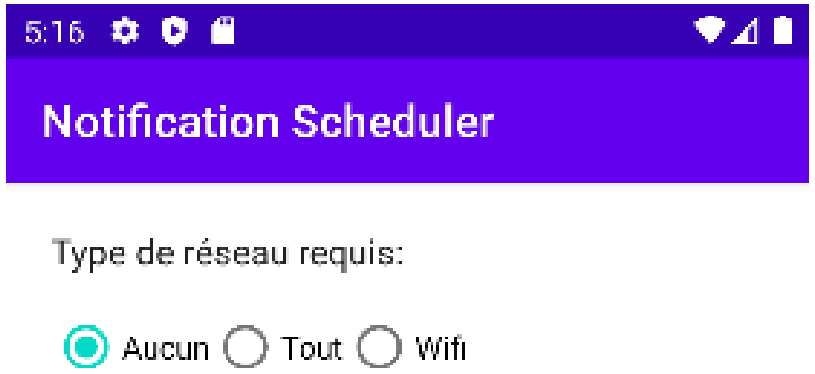
PARTIE 2

# ACTIVITÉ n° 3
## Planifier une tâche en background avec le job scheduler

### Etape 1 : mettre en œuvre les conditions de job

**Créer la mise en page de votre application**

1. Créer la mise en page de l'application pour afficher les boutons permettant à l'utilisateur de choisir les critères du réseau.




2. Dans le fichier activity_main.xml, remplacer l'élément racine par : **LinearLayout vertical**.

3. Changer le **TextView** pour qu'il ait les attributs suivants

| Attribut | Valeur |
|---|---|
| android:layout_width | "wrap_content" |
| android:layout_height | "wrap_content" |
| android:text | "Type de réseau requis : " |
| android:textAppearance | "@style/TextAppearance.AppCompat.Subhead" |
| android:layout_margin | "4dp" |

4. Ajouter un élément conteneur **RadioGroup** sous le **TextView** avec les attributs suivants :

| Attribut | Valeur |
|---|---|
| android:layout_width | "wrap_content" |
| android:layout_height | "wrap_content" |
| android:orientation | "horizontal" |
| android:id | "@+id/networkOptions" |
| android:layout_margin | "4dp" |

PARTIE 2

# ACTIVITÉ n° 3

## Planifier une tâche en background avec le job scheduler

### Etape 1 : mettre en œuvre les conditions de job

5. Ajouter trois boutons radio en tant qu'enfants du groupe radio avec leur hauteur et leur largeur de disposition définies sur "**wrap_content**" et les attributs suivants :

| **RadioButton 1** | |
|---|---|
| android:text | "Aucun" |
| android:id | "@+id/noNetwork" |
| android:checked | true |
| **RadioButton 2** | |
| android:text | "tout" |
| android:id | "@+id/anyNetwork" |
| **RadioButton 3** | |
| android:text | "Wifi" |
| android:id | "@+id/wifiNetwork" |

6. Ajouter deux boutons sous le groupe de boutons radio avec une hauteur et une largeur définies sur "**wrap content**" avec les attributs suivants :

| **Button 1** | |
|---|---|
| android:text | "Planifier le job" |
| android:onClick | "scheduleJob" |
| android:layout_gravity | "center_horizontal" |
| android:layout_margin | "4dp" |
| **Button 2** | |
| android:text | "Annuler les Jobs" |
| android:onClick | "cancelJobs" |
| android:layout_gravity | "center_horizontal" |
| android:layout_margin | "4dp" |

7. Ajouter les méthodes de base relatives aux deux méthodes **onClick()** dans **MainActivity**.

PARTIE 2

## Etape 2 : obtenir l'option de réseau sélectionnée

1. Dans **scheduleJob(),** trouver le **RadioGroup** par id et le sauvegarder dans une variable d'instance appelée **networkOptions**.

```
RadioGroup networkOptions = findViewById(R.id.networkOptions);
```

1. Obtenir l'identifiant du réseau sélectionné et le sauvegarder dans une variable entière :

```
int selectedNetworkID = networkOptions.getCheckedRadioButtonId();
```

2. Créer une variable entière d'option de réseau sélectionné et la définir comme égale à l'option de réseau par défaut (aucun réseau requis) :

```
int selectedNetworkOption = JobInfo.NETWORK_TYPE_NONE;
```

Créer une condition en utilisant l'identifiant de réseau sélectionné, et ajouter un cas pour chacun des identifiants possibles. Ensuite, affecter à l'option de réseau sélectionnée la constante de réseau **JobInfo** appropriée selon le cas :

```
switch (selectedNetworkID) {
   case R.id.noNetwork:
      selectedNetworkOption = JobInfo.NETWORK_TYPE_NONE;
      break;
   case R.id.anyNetwork:
      selectedNetworkOption = JobInfo.NETWORK_TYPE_ANY;
      break;
   case R.id.wifiNetwork:
      selectedNetworkOption = JobInfo.NETWORK_TYPE_UNMETERED;
      break;
}
```

# ACTIVITÉ n° 3
## Planifier une tâche en background avec le job scheduler

PARTIE 2

### Etape 3 : créer le JobScheduler et l'objet JobInfo

1. Créer une variable membre pour le **JobScheduler** :

```
private JobScheduler mScheduler;
```

2. Dans la méthode onCreate(), utiliser getSystemService() pour initialiser mScheduler :

```
mScheduler = (JobScheduler) getSystemService(JOB_SCHEDULER_SERVICE);
```

3. Créer une constante membre pour le JOB_ID, et lui donner la valeur 0 :

```
private static final int JOB_ID = 0;
```

4. Dans la méthode **scheduleJob(),** après le bloc Switch, créer un objet **JobInfo.Builder**. Le premier paramètre est le JOB_ID. Le second paramètre est le ComponentName du **JobService** précédemment créé. Le **ComponentName** est utilisé pour associer le **JobService** à l'objet **JobInfo**.

```
ComponentName serviceName = new ComponentName(getPackageName(),
      NotificationJobService.class.getName());
JobInfo.Builder builder = new JobInfo.Builder(JOB_ID, serviceName);
```

5. Appeler **setRequiredNetworkType()** sur l'objet **JobInfo.Builder**. Transmettre l'option de réseau sélectionnée :

```
.setRequiredNetworkType(selectedNetworkOption)
```

6. Appeler **schedule()** sur l'objet **JobScheduler**. Utiliser la méthode **build()** pour transmettre l'objet **JobInfo** :

```
JobInfo myJobInfo = builder.build();
mScheduler.schedule(myJobInfo);
```

7. Affiche un message **Toast**, indiquant à l'utilisateur que le travail a été planifié.

```
Toast.makeText(this, R.string.job_scheduled, Toast.LENGTH_SHORT)
      .show();
```

8. Dans la méthode **cancelJobs()**, vérifier si l'objet **JobScheduler** est null. Si ce n'est pas le cas, appeler **cancelAll()** sur l'objet pour supprimer tous les travaux en attente. Réinitialiser également l'objet **JobScheduler** à null et afficher un message d'avertissement pour informer l'utilisateur que le travail a été annulé.

## Etape 3 : créer le JobScheduler et l'objet JobInfo

```
if (mScheduler != null) {
    mScheduler.cancelAll();
    mScheduler = null;
    Toast.makeText(this, R.string.jobs_canceled, Toast.LENGTH_SHORT)
            .show();
}
```

9. Exécuter l'application. Il est désormais possible de définir des tâches soumises à des restrictions réseau et de déterminer le temps nécessaire à leur exécution. Dans ce cas, la tâche consiste à envoyer une notification.

L'application peut générer l'exception suivante, si on ne changer pas la contrainte de réseau en "Tout" ou "Wifi :

```
java.lang.IllegalArgumentException:
    You're trying to build a job with no constraints, this is not allowed.
```

Le crash se produit parce que la condition " Aucun réseau requis " est la condition par défaut, et cette condition ne compte pas comme une contrainte. Pour planifier correctement le **JobService**, le **JobScheduler** a besoin d'au moins une contrainte.

Dans la partie suivante, créer une variable conditionnelle qui est vraie lorsqu'au moins une contrainte est définie, et fausse sinon. Si la variable conditionnelle est vraie, l'application planifie la tâche. Si la variable conditionnelle est fausse, l'application affiche un message indiquant à l'utilisateur qu'il doit définir une contrainte.

# ACTIVITÉ n° 3
## Planifier une tâche en background avec le job scheduler

### Etape 4 : vérifier les contraintes

**JobScheduler** exige qu'au moins une contrainte soit définie. Dans cette tâche, créer un booléen qui permet de vérifier si cette condition est remplie, afin de pouvoir avertir l'utilisateur de définir au moins une contrainte s'il ne l'a pas encore fait. Lors de la création d'options supplémentaires dans les étapes suivantes, il faudra modifier ce booléen pour qu'il soit toujours vrai si au moins une contrainte est définie, et faux sinon.

Implémenter les étapes suivantes dans MainActivity.java, dans **scheduleJob()** :

1. Après la définition de **JobInfo.Builder**, au-dessus de la définition de **myJobInfo**, créer une variable booléenne appelée **constraintSet**. Cette variable est vraie si l'option de réseau sélectionnée n'est pas celle par défaut. (La valeur par défaut est **JobInfo.NETWORK_TYPE_NONE**).

```
boolean constraintSet = selectedNetworkOption
                        != JobInfo.NETWORK_TYPE_NONE
```

2. Après la définition de la contrainte (**constraintSet**), créer un bloc if/else en utilisant la variable **constraintSet**.
3. Déplacer le code qui planifie le travail et affiche le message d'avertissement dans le bloc if.
4. Si la variable **constraintSet** est fausse, afficher un message de rappel à l'utilisateur lui indiquant qu'il doit définir au moins une contrainte.

```
if (constraintSet) {
    JobInfo myJobInfo = builder.build();
    mScheduler.schedule(myJobInfo);
    Toast.makeText(this, R.string.job_scheduled, Toast.LENGTH_SHORT)
            .show();
} else {
    Toast.makeText(this, R.string.no_constraint_toast,
            Toast.LENGTH_SHORT).show();
}
```

PARTIE 2

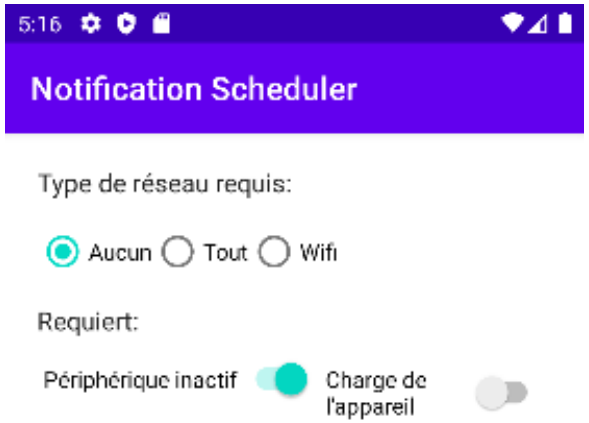
# ACTIVITÉ n° 3

## Planifier une tâche en background avec le job scheduler

### Tâche 3 : mise en œuvre des contraintes de l'inactivité et de la charge du dispositif

Grâce à **JobScheduler**, il est possible de faire en sorte que l'application attende pour exécuter le **JobService** que l'appareil soit en cours de chargement ou qu'il soit dans un état d'inactivité (écran éteint et CPU inactif).

Dans cette partie, nous ajoutons des **switchs** à l'application pour activer ces contraintes sur le **JobService**.




### Etape 1 : ajouter les éléments de l'interface utilisateur pour les nouvelles contraintes

Mettre en œuvre les étapes suivantes dans le fichier **activity_main.xml** :

1. Copier le **TextView** que vous avez utilisé pour l'étiquette de type réseau et le coller sous le **RadioGroup** ;
2. Changer l'attribut **android:text** en "Requiert :" ;
3. En dessous, ajouter un **LinearLayout horizontal** avec une marge de 4dp.

| Attribut | Valeur |
|---|---|
| android:layout_width | "match_parent" |
| android:layout_height | "wrap_content" |
| android:orientation | "horizontal" |
| android:layout_margin | "4dp" |

PARTIE 2

# ACTIVITÉ n° 3

## Planifier une tâche en background avec le job scheduler

### Etape 1 : ajouter les éléments de l'interface utilisateur pour les nouvelles contraintes

4. Créer deux vues Switch comme enfants du **LinearLayout horizontal**. Définir la hauteur et la largeur sur "**wrap_content**", et utiliser les attributs suivants :

| **Switch 1** | |
|---|---|
| android:text | "Périphérique inactif" |
| android:id | "@+id/idleSwitch" |
| **Switch 2** | |
| android:text | "Charge de l'appareil" |
| android:id | "@+id/chargingSwitch" |

PARTIE 2

# ACTIVITÉ n° 3
## Planifier une tâche en background avec le job scheduler

### Etape 2 : ajouter le code pour les nouvelles contraintes

Mettre en œuvre les étapes suivantes dans **MainActivity.java** :

1. Créer des variables membres appelées **mDeviceIdle** et **mDeviceCharging**, pour les **switchs**. Initialiser les variables dans **onCreate()** :

```
private Switch mDeviceIdleSwitch;
private Switch mDeviceChargingSwitch;
```

Dans onCreate() :

```
mDeviceIdleSwitch = findViewById(R.id.idleSwitch);
mDeviceChargingSwitch = findViewById(R.id.chargingSwitch);
```

2. Dans la méthode **scheduleJob(),** ajouter les appels suivants. Ces appels définissent des contraintes sur le **JobInfo.Builder** en fonction de la sélection de l'utilisateur dans les vues **Switch**, lors de la création de l'objet **builder** :

```
.setRequiresDeviceIdle(mDeviceIdleSwitch.isChecked())
.setRequiresCharging(mDeviceChargingSwitch.isChecked());
```

PARTIE 2

## Etape 2 : ajouter le code pour les nouvelles contraintes

3. Mettre à jour le code qui définit **constraintSet** pour prendre en compte les nouvelles contraintes :

```
boolean constraintSet = selectedNetworkOption
        != JobInfo.NETWORK_TYPE_NONE
        || mDeviceChargingSwitch.isChecked()
        || mDeviceIdleSwitch.isChecked() ;
```

4. Exécuter l'application, maintenant avec les contraintes supplémentaires. Essayer différentes combinaisons d'interrupteurs pour voir quand la notification qui indique que le travail a été exécuté est envoyée.

**Pour tester la contrainte d'état de charge dans un émulateur :**

a. Ouvrir le menu More (l'icône représentant des ellipses à côté de l'appareil émulé).

b. Aller dans le volet Batterie.

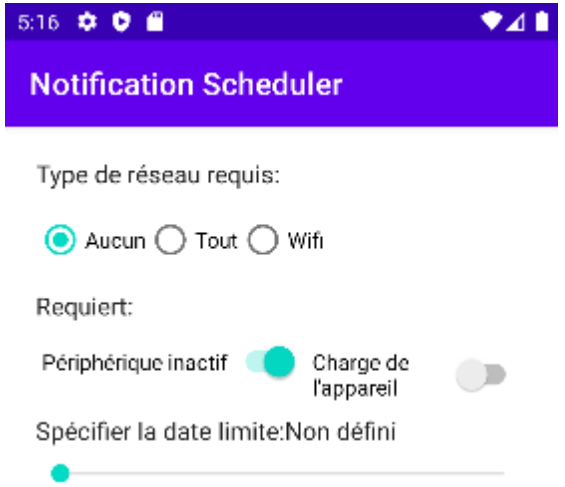
c. Basculer le menu déroulant État de la batterie. Il n'y a actuellement aucun moyen de mettre manuellement l'émulateur en mode veille.

Pour les tâches qui consomment beaucoup de batterie, comme le téléchargement ou la mise en ligne de fichiers volumineux, il est courant d'attendre que l'appareil soit au repos et connecté à une source d'alimentation.

## Tâche 4 : implémenter la contrainte de dépassement de délai

Jusqu'à présent, il n'y a aucun moyen de savoir précisément quand le framework exécutera la tâche. Le système prend en compte la gestion efficace des ressources, ce qui peut retarder la tâche en fonction de l'état de l'appareil, et ne garantit pas que la tâche sera exécutée à temps.

L'API **JobScheduler** permet de fixer une date limite stricte qui annule toutes les contraintes précédentes.



# ACTIVITÉ n° 3

## Planifier une tâche en background avec le job scheduler

### Etape 1 : ajouter une IU pour définir la date limite d'exécution de la tâche

Dans cette étape, utiliser une **SeekBar** pour permettre à l'utilisateur de fixer un délai entre 0 et 100 secondes pour l'exécution de de la tâche. L'utilisateur définit la valeur en faisant glisser la barre de recherche vers la gauche ou la droite.

Mettre en œuvre les étapes suivantes dans le fichier **activity_main.xml** :

1. Sous le **LinearLayout** qui contient les vues Switch, créer un **LinearLayout** horizontal. Ce nouveau **LinearLayout** est destiné aux étiquettes de la **SeekBar**.

| Attribut | Valeur |
|---|---|
| android:layout_width | "match_parent" |
| android:layout_height | "wrap_content" |
| android:orientation | "horizontal" |
| android:layout_margin | "4dp" |

2. Donner deux étiquettes à la barre de recherche : une étiquette statique comme celle du groupe de boutons radio, et une étiquette dynamique qui est mise à jour avec la valeur de la barre de recherche. Ajouter deux vues **TextView** à la **LinearLayout** avec les attributs suivants :

| TextView 1 | |
|---|---|
| android:layout_width | "wrap_content" |
| android:layout_height | "wrap_content" |
| android:text | "Spécifier la date limite : " |
| android:id | "@+id/seekBarLabel" |
| android:textAppearance | "@style/TextAppearance.AppCompat.Subhead" |

| TextView 2 | |
|---|---|
| android:layout_width | "wrap_content" |
| android:layout_height | "wrap_content" |
| android:text | "Non défini" |
| android:id | "@+id/seekBarProgress" |
| android:textAppearance | "@style/TextAppearance.AppCompat.Subhead" |

PARTIE 2

# ACTIVITÉ n° 3
## Planifier une tâche en background avec le job scheduler

PARTIE 2

### Etape 1 : ajouter une IU pour définir la date limite d'exécution de la tâche

3. Ajouter une vue **SeekBar** sous le **LinearLayout**. Utiliser les attributs suivants :

| Attribut | Valeur |
|---|---|
| android:layout_width | "match_parent" |
| android:layout_height | "wrap_content" |
| android:id | "@+id/seekBar" |
| android:layout_margin | "4dp" |

### Etape 2 : écrire le code pour ajouter la date limite

Mettre en œuvre les étapes suivantes dans MainActivity.java.

1. Créer une variable membre pour la **SeekBar** et initialiser la dans **onCreate()** :

```
private SeekBar mSeekBar;
```

Dans onCreate ():

```
mSeekBar = findViewById(R.id.seekBar);
```

## Etape 2 : écrire le code pour ajouter la date limite

2. Dans **onCreate(),** créer et initialiser une variable finale pour le **TextView** de progression de la barre de recherche. La variable sera accessible à partir d'une classe interne.

   final TextView seekBarProgress = findViewById(R.id.*seekBarProgress*);

3. Dans **onCreate(),** appeler **setOnSeekBarChangeListener()** sur la barre de recherche, en passant un nouveau **OnSeekBarChangeListener**. Android Studio devrait générer les méthodes requises.

```
mSeekBar.setOnSeekBarChangeListener(new SeekBar.OnSeekBarChangeListener() {
    @Override
    public void onProgressChanged(SeekBar seekBar, int i, boolean b) {
    }
    @Override
    public void onStartTrackingTouch(SeekBar seekBar) {
    }
    @Override
    public void onStopTrackingTouch(SeekBar seekBar) {
    }
});
```

## Etape 2 : écrire le code pour ajouter la date limite

4. Le deuxième argument de **onProgressChanged()** est la valeur actuelle de la barre de recherche. Dans le rappel **onProgressChanged()**, vérifier si la valeur entière est supérieure à 0 (ce qui signifie qu'une valeur a été définie par l'utilisateur). Si la valeur est supérieure à 0, définir l'étiquette de progression de la barre de recherche à la valeur entière, suivie de s pour indiquer les secondes. Dans le cas contraire, le **TextView** affiche " Non défini ".

```
if (i > 0) {
    seekBarProgress.setText(getString(R.string.seconds, i));
} else {
    seekBarProgress.setText(R.string.not_set);
}
```

5. La date limite de contournement ne doit être définie que si la valeur entière de la **SeekBar** est supérieure à 0. Dans la méthode **scheduleJob()**, créer un **int** pour stocker la progression de la barre de recherche. Créer également une variable booléenne qui est vraie si la barre de recherche a une valeur entière supérieure à 0.

```
int seekBarInteger = mSeekBar.getProgress();
boolean seekBarSet = seekBarInteger > 0;
```

6. Dans la méthode **scheduleJob()** après la définition du constructeur, si **seekBarSet** est vrai, appeler **setOverrideDeadline()** sur le **JobInfo.Builder**. Indiquer la valeur entière de la barre de recherche multipliée par 1000. Le paramètre est en millisecondes, et on veut que l'utilisateur définisse l'échéance en secondes.

```
if (seekBarSet) {
    builder.setOverrideDeadline(seekBarInteger * 1000);
}
```

7. Modifier le **constraintSet** pour inclure la valeur de **seekBarSet** comme une contrainte possible :

```
boolean constraintSet = selectedNetworkOption
        != JobInfo.NETWORK_TYPE_NONE
        || mDeviceChargingSwitch.isChecked()
        || mDeviceIdleSwitch.isChecked()
        || seekBarSet;
```

8. Exécuter l'application. L'utilisateur peut maintenant fixer un délai ferme, en secondes, avant lequel le **JobService** doit s'exécuter !

# PARTIE 3

## Manipuler les permissions

**Dans ce module, vous allez :**

- Développer une application de géolocalisation
- Maîtriser le mécanisme de demande des permissions

**17 heures**


# ACTIVITÉ n°1

## Mise en œuvre d'une application qui utilise les services de localisation

### Compétences visées :

- Exploitation de l'API de localisation
- Développement d'une application de géolocalisation

### Recommandations clés :

- Révision générale du résumé théorique
- Suivre les instructions du TP et organiser le dossier de travail

**17 heures**

# CONSIGNES

## 1. Pour le formateur :

- Rappeler les APIs de localisation
- Demander aux apprenants de suivre les étapes décrites dans l'activité

## 2. Pour l'apprenant :

- Suivre les étapes indiquées dans l'activité

## 3. Conditions de réalisation :

- Support de résumé théorique accompagnant
- Connexion internet pour installer les dépendances
- Un ordinateur sur lequel est installée la dernière version stable d'Android Studio

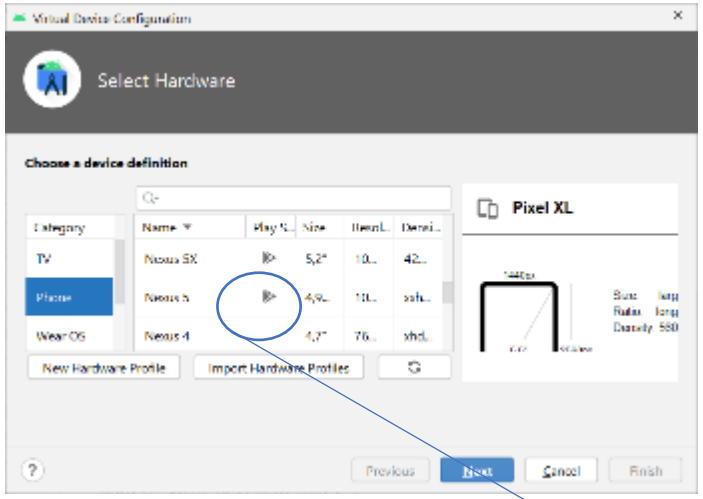
## 4. Critères de réussite :

- Le stagiaire doit être capable de :
  - manipuler l'API de localisation
  - développer une application de la géolocalisation

## TP n° 1 : API de localisation Android pour suivre la position courante

L'API de localisation Android peut être utilisée pour suivre la localisation courante d'un smartphone et afficher sa position dans l'application (IU, Map, etc.).

Dans cette activité, nous allons développer une application mobile qui récupère l'emplacement courant de l'utilisateur et l'affiche dans une interface graphique comme illustré dans la figure suivante :





**Remarque**

Pour tester l'application vous devez installer un émulateur qui dispose des services google.

On demande de **suivre les étapes** de l'activité étape par étape.

PARTIE 3

# ACTIVITÉ n° 1
## Utilisation des services de localisation

PARTIE 3

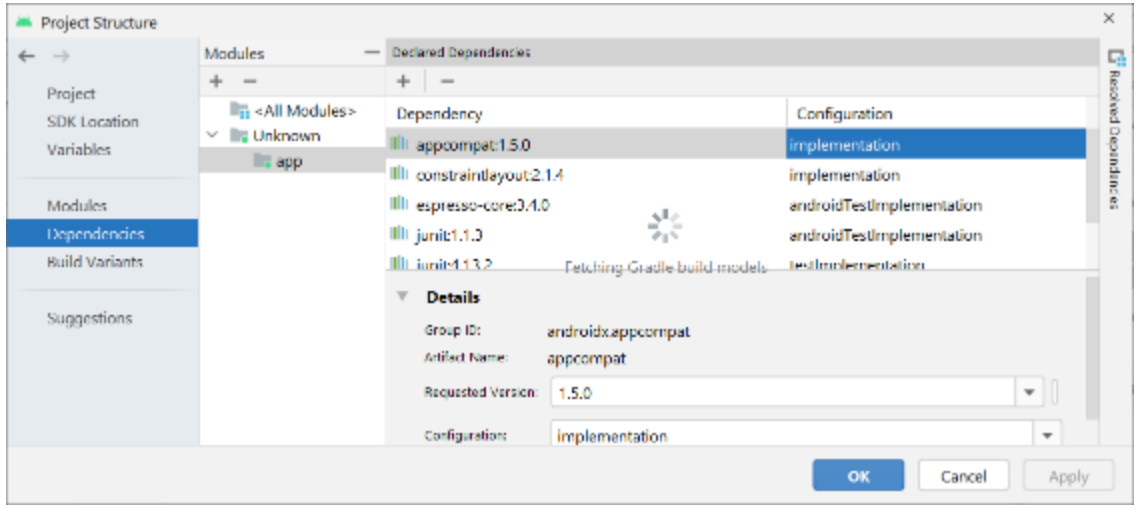
### Etape 1 : créer un projet Android

Créer un nouveau projet sous Android Studio :

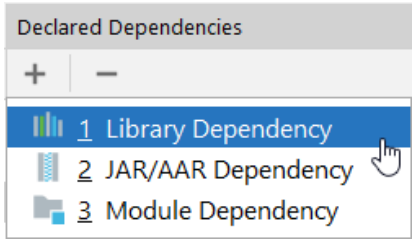
1. Cliquer sur File, puis **New -> New Project**
2. Choisir l'activité **Empty**
3. Sélectionner comme langage **Java** ;
4. Choisir le **SDK minimum** en fonction de vos besoins

### Etape 2 : ajouter la dépendance

1. **Clic droit sur app -> Open Module Settings -> Dependencies**




Clic ensuite sur **( + )**




2. Ensuite chercher **: play-services-location**
3. Choisir la dernière version et cliquer sur **OK**

## Etape 3 : créer l'interface graphique

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_main"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_gravity="center_horizontal"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity">

    <Button
        android:id="@+id/bt_location"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="GET LOCATION" />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="16dp"
        android:orientation="horizontal">

        <TextView
            android:id="@+id/text_view1"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1" />

        <TextView
            android:id="@+id/text_view2"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1" />
    </LinearLayout>

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:paddingTop="16dp"
        android:orientation="horizontal">

        <TextView
            android:id="@+id/text_view3"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1" />

        <TextView
            android:id="@+id/text_view4"
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1" />
    </LinearLayout>

    <TextView
        android:id="@+id/text_view5"
        android:paddingTop="16dp"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_weight="1" />

</LinearLayout>
```

# ACTIVITÉ n° 1
## Utilisation des services de localisation

PARTIE 3

### Etape 4 : initialiser les composants de l'interface graphique

1. Dans la classe **MainActivity.java** déclarer les composants de l'interface graphique :

```
Button bt_location;
TextView textView1, textView2, textView3, textView4, textView5;
```

2. Dans la méthode **onCreate()** initialiser les composants :

```
bt_location = findViewById(R.id.bt_location);
textView1 = findViewById(R.id.text_view1);
textView2 = findViewById(R.id.text_view2);
textView3 = findViewById(R.id.text_view3);
textView4 = findViewById(R.id.text_view4);
textView5 = findViewById(R.id.text_view5);
```

3. Associer un événement de clic au bouton :

```
bt_location.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
    }
});
```

### Etape 5 : déclarer et vérifier les permissions

1. Décaler les permissions de localisation dans le fichier **AndroidManifest.xml** :

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

2. Déclarer un objet de classe **FusedLocationProviderClient** dans la classe **MainActivity.java** :

```
FusedLocationProviderClient fusedLocationProviderClient;
```

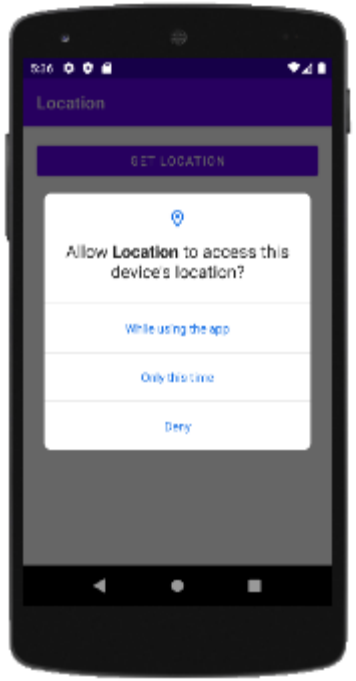
3. Initialiser l'objet dans la méthode **onCreate ()** :

```
fusedLocationProviderClient =
        LocationServices.getFusedLocationProviderClient(this);
```

4. Au clic sur le bouton **bt_location**, demander à l'utilisateur d'accorder les permissions dans une méthode **getLocation()** :

## Etape 5 : déclarer et vérifier les permissions

```
public void getLocation() {
   if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED &&
                  ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_COARSE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
                  ActivityCompat.requestPermissions(MainActivity.this,
                              new String[]{Manifest.permission.ACCESS_FINE_LOCATION, Manifest.permission.ACCESS_COARSE_LOCATION}, 44);
   } else {
     //Récupération de la dernière localisation
   }
}
```


PARTIE 3

## Etape 6 : récupérer la dernière position enregistrée

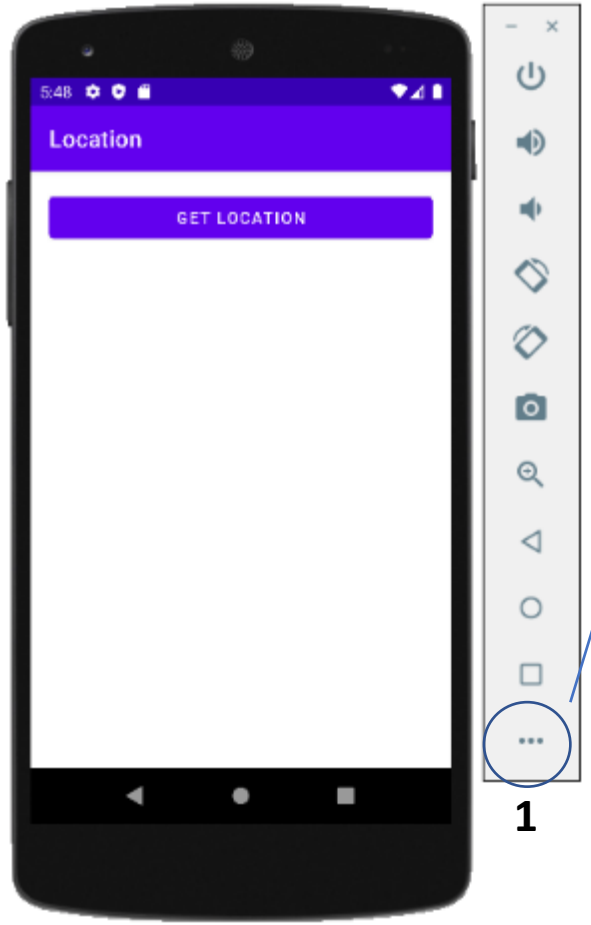
```
fusedLocationProviderClient.getLastLocation().addOnSuccessListener(new
OnSuccessListener<Location>() {
   @Override
   public void onSuccess(Location location) {
      if (location != null) {
         try {


            //initialiser geocoder
            Geocoder geocoder = new Geocoder(MainActivity.this,      Locale.getDefault());


            //initialiser l'adresse de localisation
            List<Address> addresses =  geocoder.getFromLocation(location.getLatitude(),
                     location.getLongitude(), 1);

            //afficher la latitude dans le textview
            textView1.setText(Html.fromHtml(
                  "<font color='#6200EE'><b>Latitude : </b><br></font>"
                        + addresses.get(0).getLatitude()
            ));
            // afficher la longitude dans le textview
            textView2.setText(Html.fromHtml(
                  "<font color='#6200EE'><b>Longitude : </b><br></font>"
                        + addresses.get(0).getLongitude()
            ));
            // afficher le pays dans le textview
            textView3.setText(Html.fromHtml(
                  "<font color='#6200EE'><b>Nom de pays : </b><br></font>"
                        + addresses.get(0).getCountryName()
            ));
            // afficher la localité dans le textview
            textView4.setText(Html.fromHtml(
                  "<font color='#6200EE'><b>Localité  : </b><br></font>"
                        + addresses.get(0).getLocality()
            ));
            // afficher l'adresse dans le textview
            textView5.setText(Html.fromHtml(
                  "<font color='#6200EE'><b>Adresse : </b><br></font>"
                        + addresses.get(0).getAddressLine(0)
            ));
         } catch (IOException e) {
                  e.printStackTrace();
         }
      } else {
         Toast.makeText(getApplicationContext(), "Aucune position enregistrée",
                  Toast.LENGTH_SHORT).show();
      }
   }
});
```

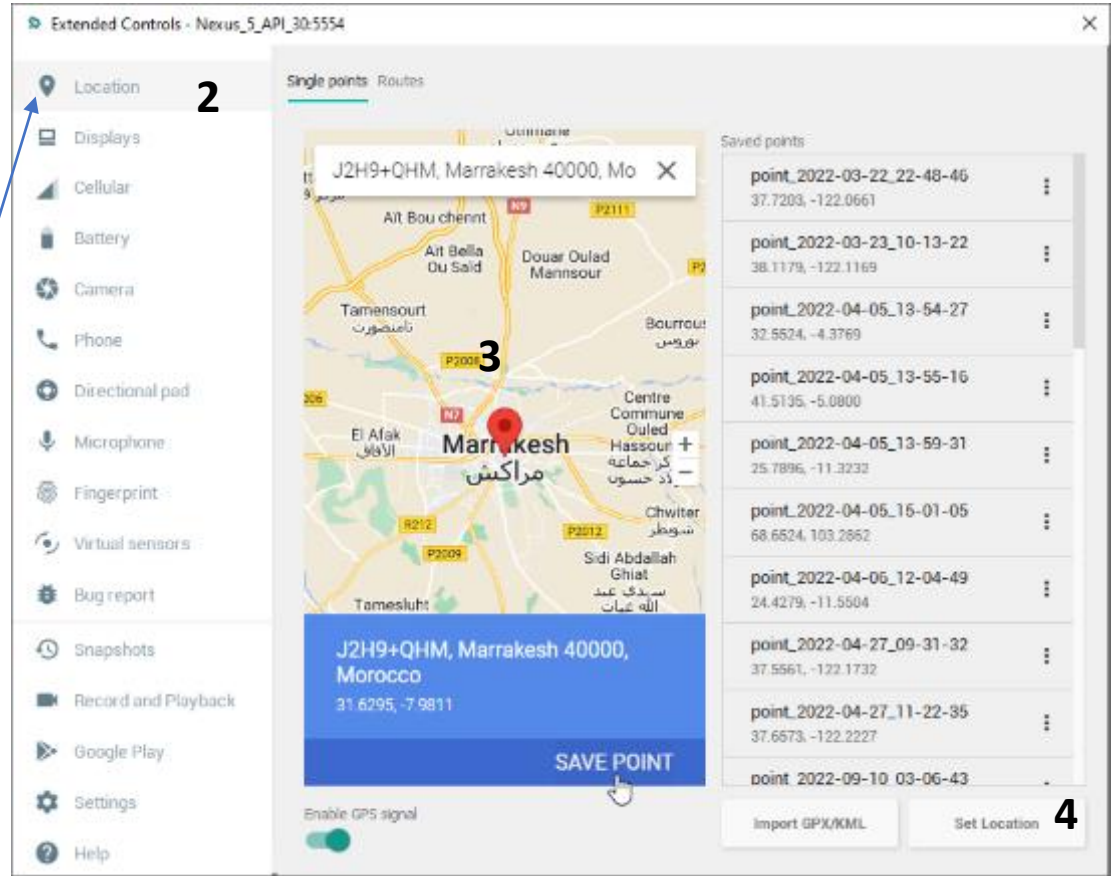
## Etape 6 : récupérer la dernière position enregistrée

```
fusedLocationProviderClient.getLastLocation().addOnFailureListener(new OnFailureListener() {
    @Override
    public void onFailure(@NonNull Exception e) {
        Toast.makeText(getApplicationContext(), e.getMessage(), Toast.LENGTH_SHORT).show();

    }
});
```

## Etape 7 : tester l'application

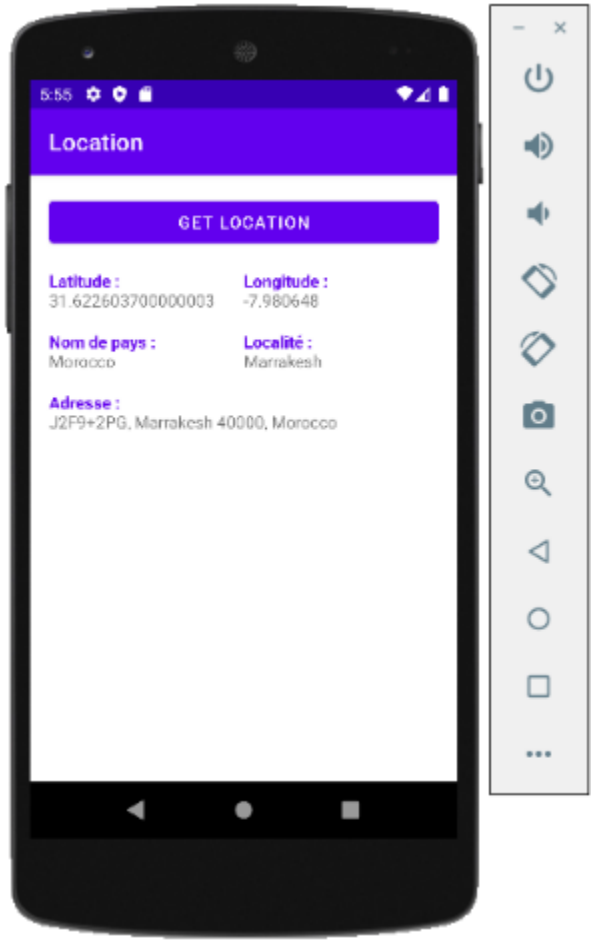
Enregistrer la position.




PARTIE 3

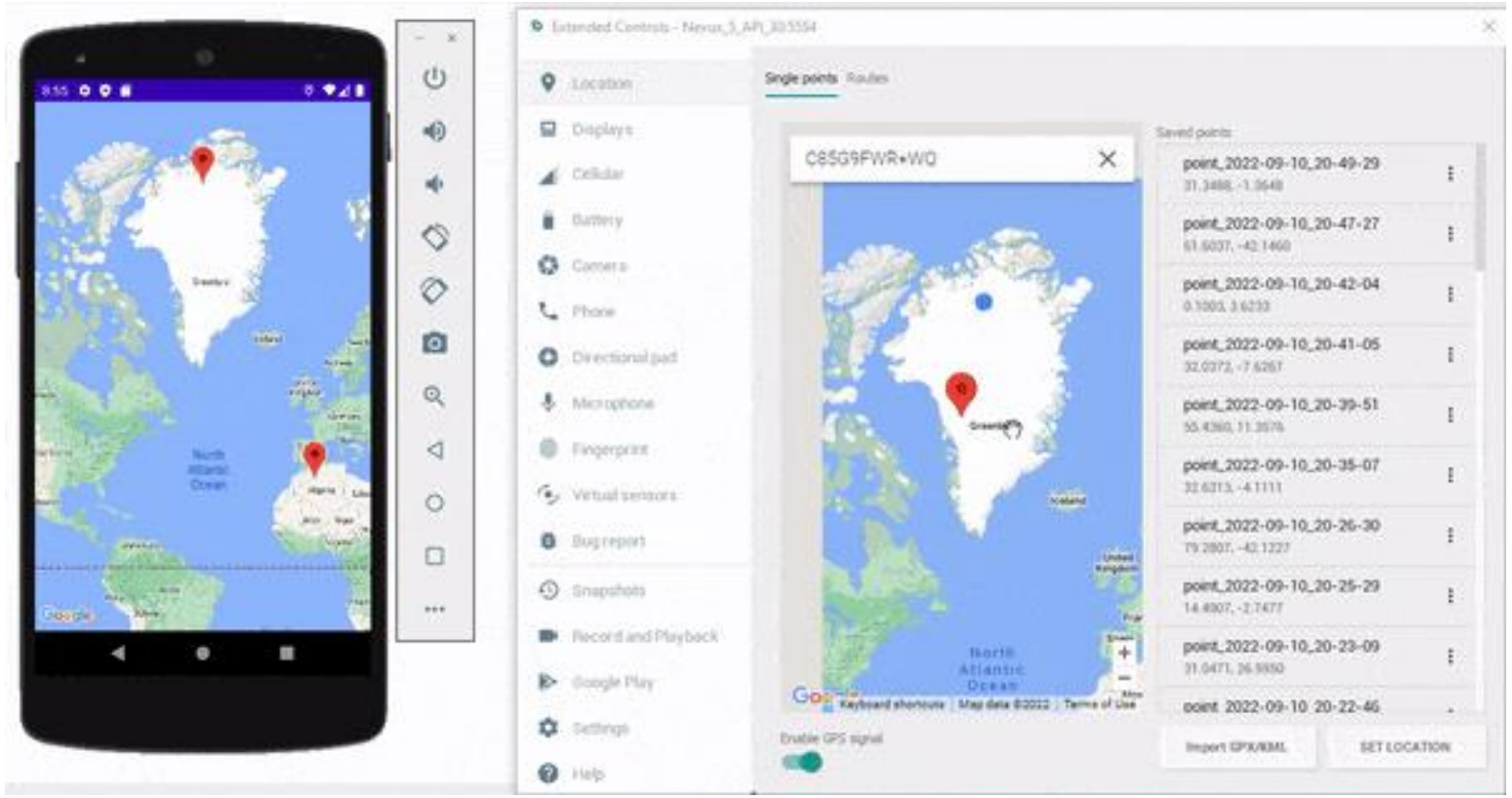
## Etape 7 : tester l'application



PARTIE 3

## TP n° 2 : exploiter la map pour afficher les positions en temps réel

Dans ce TP, nous allons créer une application mobile permettant de suivre les positions d'un smartphone en temps réel dans une Map. L'application va permettre d'afficher les positions enregistrées dans un **intervalle de temps** quand l'utilisateur quitte un **rayon** prédéfini. On demande de **suivre les étapes** de l'activité étape par étape.



# ACTIVITÉ n° 1
## Utilisation des services de localisation et de la map

PARTIE 3

### Etape 1 : créer un nouveau projet

Créer un nouveau projet sous Android Studio :

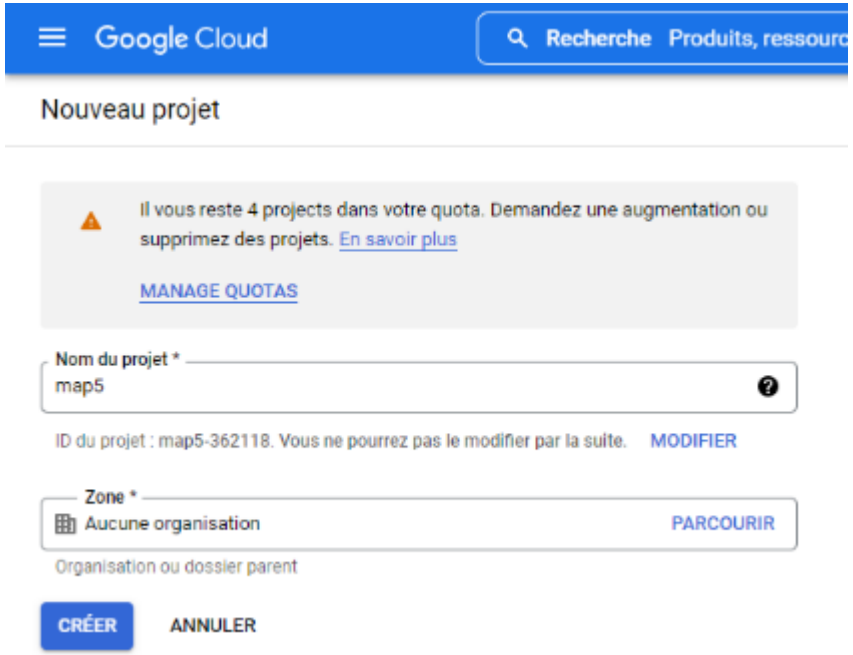
1. Cliquer sur File, puis **New -> New Project** ;
2. Choisir l'activité **Google Maps Activity** ;
3. Sélectionner comme langage **Java** ;
4. Choisir le **SDK minimum** en fonction de vos besoins

### Etape 2 : Récupérer la clé

1. Cliquer sur le lien qui figure dans le fichier **AndroidManifest.xml** pour générer la clé :

*https://developers.google.com/maps/documentation/android-sdk/get-api-key*

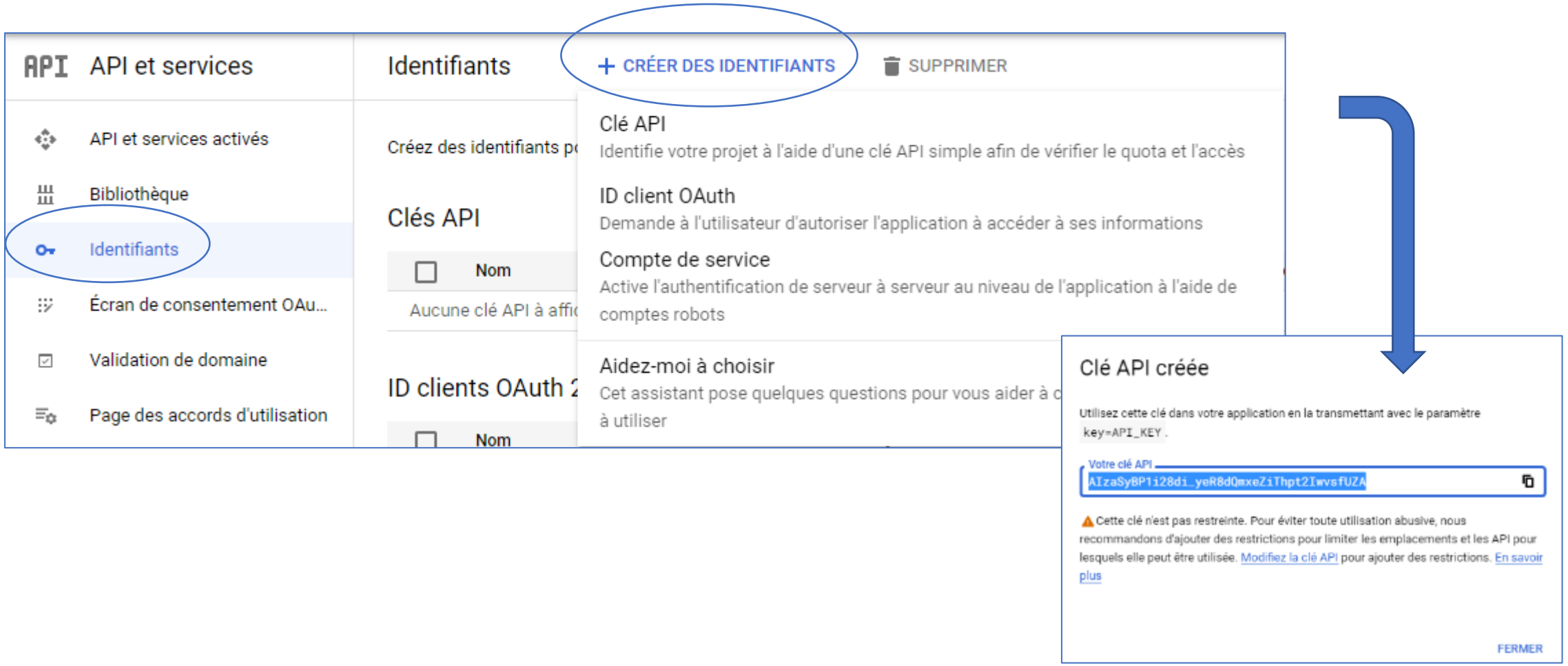
2. Créer un nouveau projet dans Google Cloud :



# ACTIVITÉ n° 1

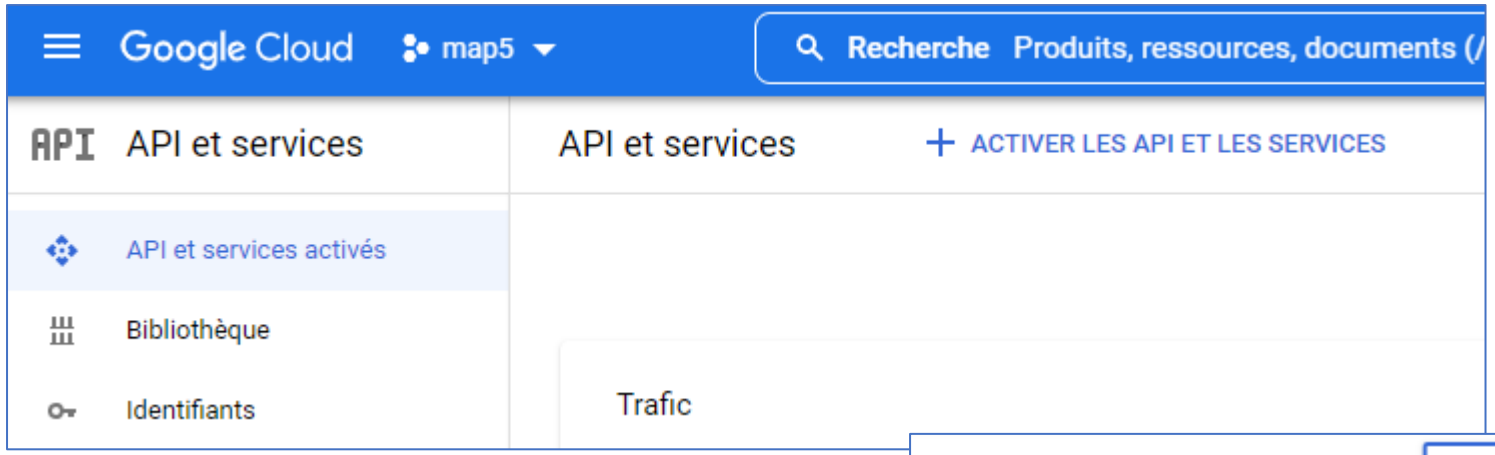
## Utilisation des services de localisation et de la map

### Etape 2 : générer la clé

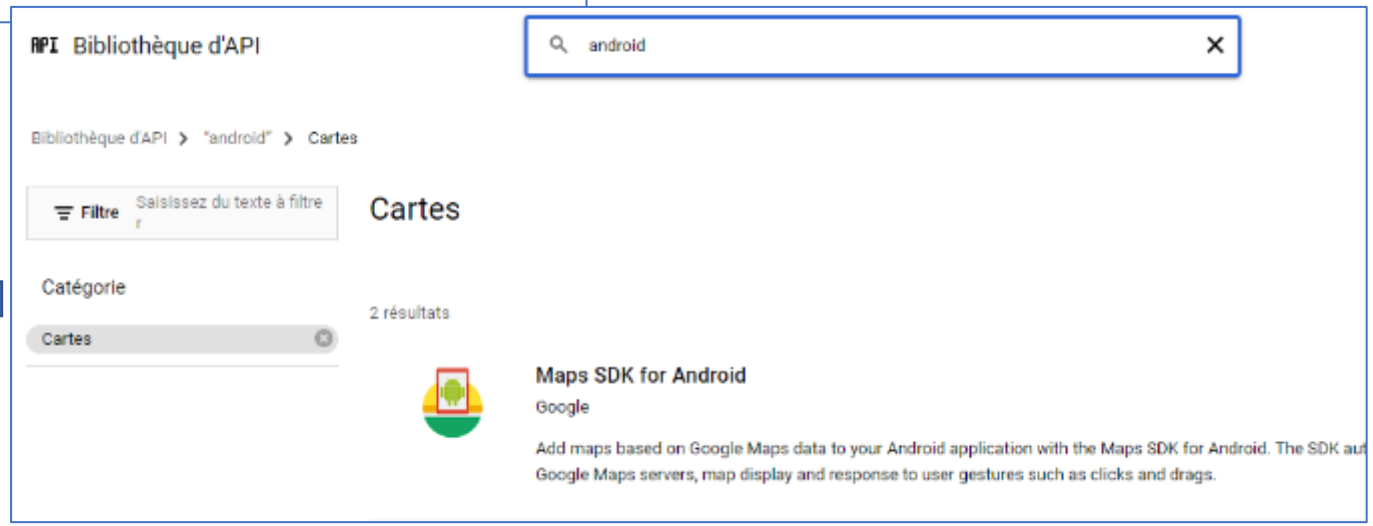



PARTIE 3

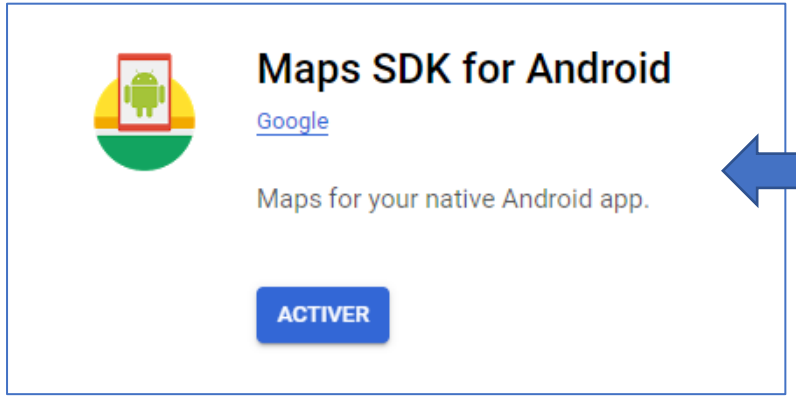
## Etape 2 : activer la clé pour une application Android




1




2




PARTIE 3

PARTIE 3

## Etape 3 : lancer l'application

Lancer l'application pour afficher la **Map** par défaut.


## Etape 4 : déclarer et activer les permissions

1. Dans la classe **MapsActivity** déclarer :

```
private LocationManager locationManager;
```

2. Initialiser l'objet **locationManager** dans la méthode **onCreate()** :

```
locationManager =
        (LocationManager) getSystemService(Context.LOCATION_SERVICE);
```

3. Ajouter les permissions dans le fichier **AndroidManifest.xml** :

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

## Etape 4 : déclarer et activer les permissions

4. Dans la méthode **onMapReady** demander l'activation des permissions :

```
public void onMapReady(GoogleMap googleMap) {
   mMap = googleMap;
   if (ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION) != PackageManager.PERMISSION_GRANTED &&
                  ActivityCompat.checkSelfPermission(this, Manifest.permission.ACCESS_COARSE_LOCATION) != PackageManager.PERMISSION_GRANTED) {
      ActivityCompat.requestPermissions(MapsActivity.this, new String[]{
                                 Manifest.permission.ACCESS_FINE_LOCATION, Manifest.permission.ACCESS_COARSE_LOCATION}, 44);
   } else { //Récupération des positions } }
```

## Etape 5 : afficher une position par défaut

1. Dans la méthode **onMapReady** créer une position par défaut :

```
LatLng def = new LatLng(31, -1);
mMap.addMarker(new MarkerOptions().position(def).title("Position initial"));
mMap.moveCamera(CameraUpdateFactory.newLatLng(def));
```

2. Lancer l'application et vérifier l'affichage de la position dans la **Map**.

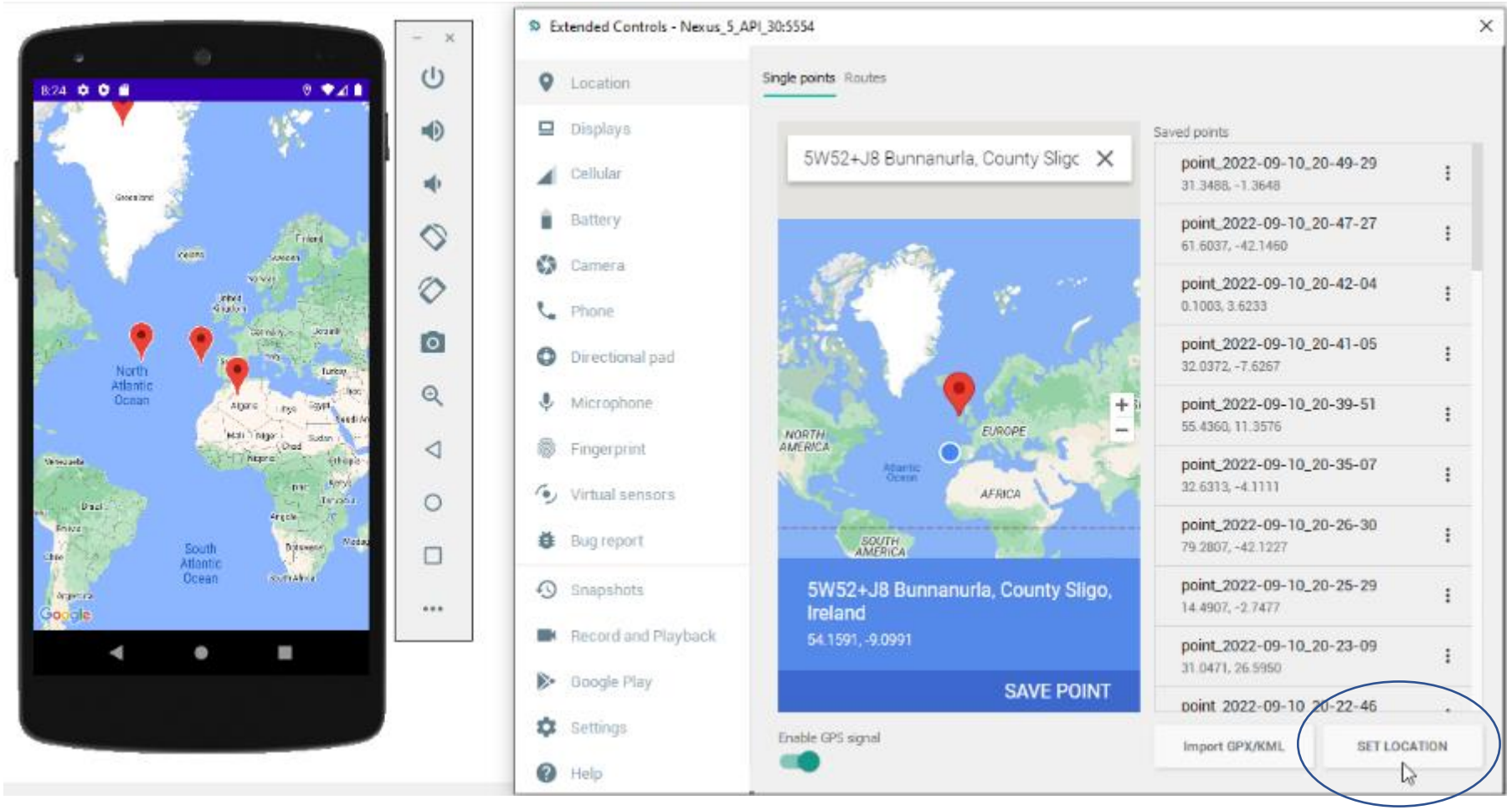
## Etape 6 : afficher les positions en temps réel

Afficher les positions dans la Map dans un intervalle de **5 seconds** si l'utilisateur quitte un **rayon de 100 m**.

```
locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 5000, 100, new LocationListener() {
    @Override
    public void onLocationChanged(@NonNull Location location) {
        LatLng pos = new LatLng(location.getLatitude(), location.getLongitude());
        mMap.addMarker(new MarkerOptions().position(pos).title("Position"));
        mMap.moveCamera(CameraUpdateFactory.newLatLng(pos));
    }
});
```

PARTIE 3

## Etape 7 : tester l'application



# PARTIE 4

## Créer des tests unitaires

**Dans ce module, vous allez :**

- Utiliser JUnit pour tester une application mobile
- Effectuer des tests avec Mockito
- Créer des tests automatisés

16 heures


# ACTIVITÉ n° 1

## Tester des classes en utilisant JUnit et Mockito

**Compétences visées :**

- Création des tests unitaires à l'aide de JUnit4
- Création des tests unitaires à l'aide de Mockito

**Recommandations clés :**

- Révision générale du résumé théorique
- Suivre les instructions du TP et organiser le dossier de travail

**8 heures**

# CONSIGNES

## 1. Pour le formateur :

- Demander aux apprenants de suivre les étapes décrites dans le résumé théorique du cours et d'appliquer les procédures
- Demander aux apprenants de réaliser le travail de synthèse

## 2. Pour l'apprenant :

Dans cette activité, vous allez apprendre comment :

- configurer les tests
- créer des tests unitaires à l'aide de JUnit et Mockito

## 3. Conditions de réalisation :

- Support de résumé théorique accompagnant
- Connexion internet pour installer les dépendances
- Un ordinateur sur lequel est installée la dernière version stable d'Android Studio
- Un appareil de test ou un émulateur avec au moins Android 2.3.3 (niveau API 10), ou une version ultérieure

## 4. Critères de réussite :

- Le stagiaire doit être capable de :
  - ➢ créer des tests unitaires avec JUnit
  - ➢ créer des tests unitaires avec Mockito

# ACTIVITÉ n° 1

## Tester des classes en utilisant JUnit

### TP n° 1 : tests avec JUnit

**Exercice 1 :**

Donner le résultat d'exécution de la classe de Test suivante :

```
public class JUnitAnnotation {

   @BeforeClass
   public static void beforeClass() {
      System.out.println("before class");
   }

   @AfterClass
   public static void  afterClass() {
      System.out.println("after class");
   }

   @Before
   public void before() {
      System.out.println("before");
   }

   @After
   public void after() {
      System.out.println("after");
   }

   @Test
   public void test() {
      System.out.println("test");
   }

   @Ignore
   public void ignoreTest() {
      System.out.println("ignore test");
   }
}
```

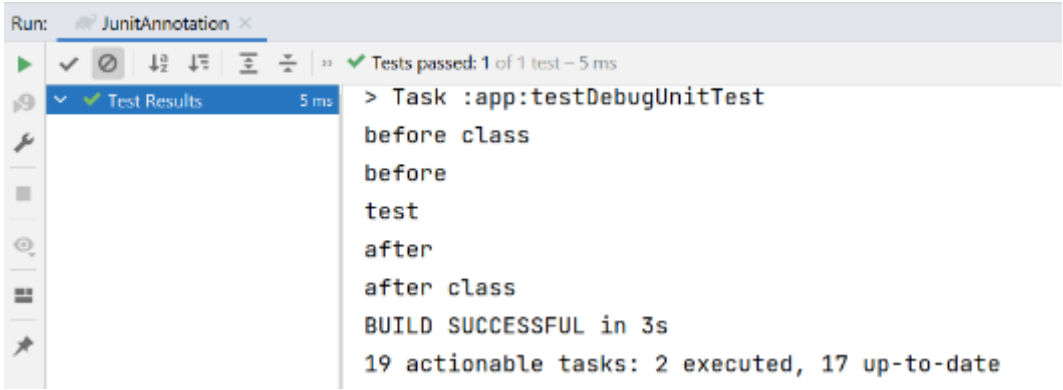
PARTIE 4

# ACTIVITÉ n° 1

## Tester des classes en utilisant JUnit

### Solution

```
public class JUnitAnnotation {
   //exécuter avant la classe
   @BeforeClass
   public static void beforeClass() {
      System.out.println("before class");
   }

   //exécuter après la classe
   @AfterClass
   public static void  afterClass() {
      System.out.println("after class");
   }

   //exécuter avant le test
   @Before
   public void before() {
      System.out.println("before");
   }

   //exécuter après le test
   @After
   public void after() {
      System.out.println("after");
   }

   //cas de test
   @Test
   public void test() {
      System.out.println("test");
   }

   //le cas de test est ignoré et ne sera pas exécuté
   @Ignore
   public void ignoreTest() {
      System.out.println("ignore test");
   }
}
```




PARTIE 4

# ACTIVITÉ n° 1

## Tester des classes en utilisant JUnit

### TP n° 1 : tests avec JUnit

**Exercice 2 :**

Ecrire une classe de test permettant de tester l'égalité de deux chaines de caractères.

PARTIE 4

# ACTIVITÉ n° 1

## Tester des classes en utilisant JUnit

### Solution

```
public class Ex1UnitTest {

   @Before
   public void setUp() throws  Exception{
      System.out.println("Before");
   }

   @After
   public void tearDown () throws Exception{
      System.out.println("After");
   }
   @Test
   public void equalString() {
      System.out.println("Je developpeur Mobile");
      String str1="Je developpeur Mobile";
      assertEquals("Je developpeur Mobile", str1);
   }
}
```

PARTIE 4

## TP n° 1 : tests avec JUnit

**Exercice 3 :**

Créer une classe de Test permettant de tester la classe **StudentList** suivante :

```
public class StudentList {

   private List<String> students = new ArrayList<String>();

   public void remove(String name) {
      students.remove(name);
   }

   public void add(String name) {
      students.add(name);
   }

   public void removeAll(){
      students.clear();
   }

   public int sizeOfStudent() {
      return students.size();
   }

}
```

# ACTIVITÉ n° 1

## Tester des classes en utilisant JUnit

### Solution

```
public class StudentListTest {
   StudentList list = null;

   @Before
   public void setUp() throws  Exception{
       list = new StudentList();
   }

   @After
   public void tearDown () throws Exception{
      System.out.println("After");
   }
   @Test
   public void testAdd() {
      list.add("Ali");
      list.add("Amal");
      list.add("Kamal");
      list.add("Amine");
      assertEquals("Ajout de 4 étudiants à la liste", 4, list.sizeOfStudent());
   }
   @Test
     public void testRemove() {
        list.add("Fatima");
        list.add("Amine");
        list.remove("Amine");
        assertEquals("Suppression d'un étudiant de la liste", 1, list.sizeOfStudent());
     }

     @Test
     public void removeAll() {
        list.removeAll();
     }
}
```

PARTIE 4

# ACTIVITÉ n° 1

## Tester des classes en utilisant JUnit

### TP n° 1 : tests avec JUnit

**Exercice 4 :**

Etant donné la classe suivante :

```
public class PrimeNumberChecker {
   public Boolean validate(final Integer primeNumber) {
      for (int i = 2; i < (primeNumber / 2); i++) {
         if (primeNumber % i == 0) {
            return false;
         }
      }
      return true;
   }
}
```

1. Que permet de faire la méthode **validate()** ?
2. Créer une classe de Test avec des scénarios de test pour tester la classe **PrimeNumberChecker**.

PARTIE 4

# ACTIVITÉ n° 1

## Tester des classes en utilisant JUnit

### Solution

1. La méthode validate() permet de vérifier si un nombre passé en paramètre est un nombre premier ou pas.
2. Classe de Test :

```
@RunWith(Parameterized.class)
public class PrimeNumberCheckerTest {
   private Integer numeroEntree;
   private Boolean resultatAttendu;
   private PrimeNumberChecker primeNumberChecker;

   @Before
   public void initialize() {
      primeNumberChecker = new PrimeNumberChecker();
   }
   // Chaque paramètre doit être placé comme un argument ici.
   // Chaque fois que le runner se déclenche, il passe les arguments
   // à partir des paramètres définis dans la méthode primeNumbers().
   public PrimeNumberCheckerTest(Integer numeroEntree,
                  Boolean resultatAttendu) {
      this.numeroEntree = numeroEntree;
      this.resultatAttendu = resultatAttendu;
   }

   @Parameterized.Parameters
   public static Collection primeNumbers() {
      return Arrays.asList(new Object[][] {
            { 2, true },
            { 6, false },
            { 19, true },
            { 22, false },
            { 23, true },
            { 73, true },
            { 121, false }
      });
   }

   // Ce test sera exécuté 7 fois puisque nous avons 7 paramètres définis.
   @Test
   public void testPrimeNumberChecker() {
      System.out.println("Nombre paramétré est : " + numeroEntree);
      assertEquals(resultatAttendu,
            primeNumberChecker.validate(numeroEntree));
   }
}
```

PARTIE 4

# ACTIVITÉ n° 1

## Tester des classes en utilisant Mockito

### TP n° 2 : tests avec Mockito

Dans ce TP, nous allons apprendre à intégrer **JUnit** et **Mockito** pour effectuer des tests unitaires. Ici, nous allons créer une application mathématique qui utilise **CalculatorService** pour effectuer des opérations mathématiques de base telles que l'addition, la soustraction, la multiplication et la division.

Nous utiliserons **Mockito** pour simuler l'implémentation fictive de **CalculatorService**. De plus, nous allons faire un usage intensif des annotations pour montrer leur compatibilité avec **JUnit** et **Mockito**.

Le processus est présenté **étape par étape** dans les diapositives qui suivent.

### Etape 1

Créer un nouveau projet sous Android Studio :

1. Cliquer sur File, puis **New -> New Project** ;
2. Choisir l'activité **Empty** ;
3. Sélectionner comme langage **Java** ;
4. Choisir le **SDK minimum** en fonction de vos besoins.

PARTIE 4

# ACTIVITÉ n° 1

## Tester des classes en utilisant Mockito

### Etape 2

Créer l'interface suivante dans le package principal de projet :

```
public interface CalculatorService {
   public double add(double input1, double input2);
   public double sous(double input1, double input2);
   public double multi(double input1, double input2);
   public double div(double input1, double input2);
}
```

### Etape 3

Créer la classe **CalculatorApplication** dans le package principal de projet :

```
public class CalculatorApplication {
   private CalculatorService calcService;

   public void setCalculatorService(CalculatorService calcService){
      this.calcService = calcService;
   }

   public double add(double input1, double input2){
      return calcService.add(input1, input2);
   }

   public double sous(double input1, double input2){
      return calcService.sous(input1, input2);
   }

   public double multi(double input1, double input2){
      return calcService.multi(input1, input2);
   }

   public double div(double input1, double input2){
      return calcService.div(input1, input2);
   }
}
```

PARTIE 4

# ACTIVITÉ n° 1
## Tester des classes en utilisant Mockito

PARTIE 4

### Etape 4

Créer la classe **CalculatorApplicationTester** dans le package de test de l'application :

- Attacher un runner avec la classe de test pour initialiser les données de test ;
- Créer et injecter l'objet fictif (CalculatorApplication) ;
- Créer l'objet fictif à injecter (CalculatorService) ;
- Tester la fonctionnalité d'ajout.

```
import org.JUnit.Test;
import org.JUnit.runner.RunWith;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.JUnit.Assert;
import static org.mockito.Mockito.when;
import org.mockito.JUnit.MockitoJUnitRunner;

// @RunWith attache un runner avec la classe de test pour initialiser les données
// de test
@RunWith(MockitoJUnitRunner.class)
public class CalculatorApplicationTester {

    //@InjectMocks annotation est utilisée pour créer et injecter l'objet fictif
    @InjectMocks
    CalculatorApplication calculatorApplication = new CalculatorApplication();

    //@Mock annotation est utilisée pour créer l'objet fictif à injecter
    @Mock
    CalculatorService calcService;

    @Test
    public void testAdd(){
        // ajouter le comportement de calcService pour additionner deux nombres
        when(calcService.add(10.0,20.0)).thenReturn(30.00);

        // tester la fonctionnalité d'ajout
        Assert.assertEquals(calculatorApplication.add(10.0, 20.0),30.0,0);
    }
}
```

On demande de tester les autres méthodes.


# ACTIVITÉ n° 2

## Tester les interfaces d’une application en utilisant Espresso et UI automator

**Compétences visées :**

- Création des tests des interfaces graphiques
- Création des tests automatisés

**Recommandations clés :**

- Révision générale du résumé théorique
- Suivre les instructions du TP et organiser le dossier de travail

**8 heures**

# CONSIGNES

## 1. Pour le formateur :

- Demander aux apprenants de suivre les étapes décrites dans le résumé théorique du cours et d'appliquer les procédures
- Demander aux apprenants de réaliser le travail de synthèse

## 2. Pour l'apprenant :

Dans cette activité, vous apprendrez comment :

- créer des tests des interfaces graphiques à l'aide de Espresso
- créer des tests des interfaces graphiques à l'aide de UI Automator

## 3. Conditions de réalisation :

- Support de résumé théorique accompagnant
- Connexion internet pour installer les dépendances
- Un ordinateur sur lequel est installée la dernière version stable d'Android Studio
- Un appareil de test ou un émulateur avec au moins Android 2.3.3 (niveau API 10), ou une version ultérieure

## 4. Critères de réussite :

- Le stagiaire doit être capable de :

➢ créer des tests des interfaces graphiques à l'aide de Espresso

➢ créer des tests des interfaces graphiques à l'aide de UI Automator

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant Espresso

### TP n° 1 : 1er Test avec Espresso

Dans ce TP vous allez réaliser le premier scénario de test Espresso, pour une simple application Hello World.

Pour réaliser ce TP, suivre **les étapes indiquées** dans les diapositives suivantes.

### Etape 1 :

Créer un nouveau projet sous Android Studio :

1. Cliquer sur File, puis **New -> New Project** ;
2. Choisir l'activité **Empty** ;
3. Sélectionner comme langage **Java** ;
4. Choisir le **SDK minimum** en fonction de vos besoins.

PARTIE 4

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant Espresso

### Etape 2 :

Attribuer un id pour le **TextView :**

```
<TextView
   android:id="@+id/textView"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Hello World!"
   app:layout_constraintBottom_toBottomOf="parent"
   app:layout_constraintLeft_toLeftOf="parent"
   app:layout_constraintRight_toRightOf="parent"
   app:layout_constraintTop_toTopOf="parent" />
```

### Etape 3 :

Ajouter les dépendances dans la **build.gradle** (Module) :

```
   testImplementation 'JUnit:JUnit:4.13.2'
   androidTestImplementation 'androidx.test.ext:JUnit:1.1.3'
   androidTestImplementation 'androidx.test:runner:1.4.0'
   androidTestImplementation 'androidx.test:rules:1.4.0'
   androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
```

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant Espresso

PARTIE 4

### Etape 4 :

Créer la classe de Test dans le package (androidTest) :

```
import androidx.test.filters.LargeTest;
import androidx.test.ext.JUnit.runners.AndroidJUnit4;
import androidx.test.rule.ActivityTestRule;
import org.JUnit.After;
import org.JUnit.Before;
import org.JUnit.Rule;
import org.JUnit.Test;
import org.JUnit.runner.RunWith;
import static androidx.test.espresso.Espresso.onView;
import static androidx.test.espresso.assertion.ViewAssertions.matches;
import static androidx.test.espresso.matcher.ViewMatchers.isDisplayed;
import static androidx.test.espresso.matcher.ViewMatchers.withId;


@RunWith(AndroidJUnit4.class)
@LargeTest
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class ExampleInstrumentedTest {

 @Rule
  public ActivityTestRule<MainActivity> mActivityRule = new ActivityTestRule<>(MainActivity.class);

  @Before
  public void setUp() throws Exception {
     //Avant l'exécution du cas de test
  }

  @Test
  public void test1ChatId() {
     onView(withId(R.id.textView)).check(matches(isDisplayed()));
  }

  @After
  public void tearDown() throws Exception {
     //Après l'exécution du cas de test
  }
}
```

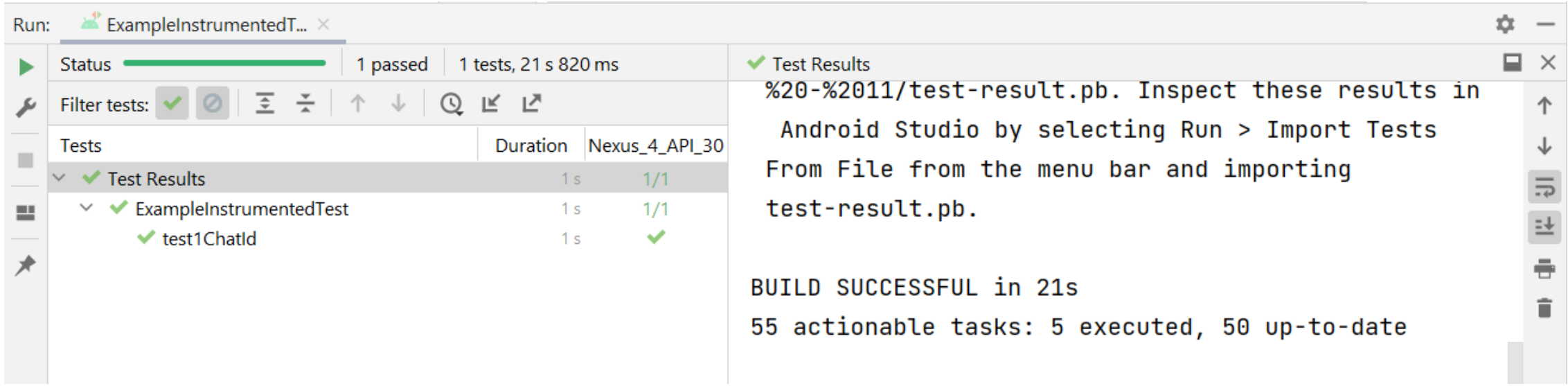
1. La classe utilise deux annotations **@LargeTest** et **@Runwith.** Ces deux annotations sont utilisées pour spécifier le comportement de la classe de test ;
2. La ligne de code suivante est utilisée pour exécuter le scénario de test dans un ordre particulier : @FixMethodOrder(MethodSorters.*NAME_ASCENDING*) ;
3. **@Before** et **@After** sont des annotations utilisées pour spécifier quelle méthode doit être exécutée avant chaque cas de test et quelle méthode doit être exécutée après l'achèvement de chaque cas de test ;
4. **onView()** est utilisée pour sélectionner une vue pour le test et **withId()** est utilisée pour localiser l'élément UI et **check ()** est utilisée ici pour vérifier si un élément spécifique est affiché ou non.

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant Espresso

### Etape 5 :

1. Exécuter le cas de test.




PARTIE 4

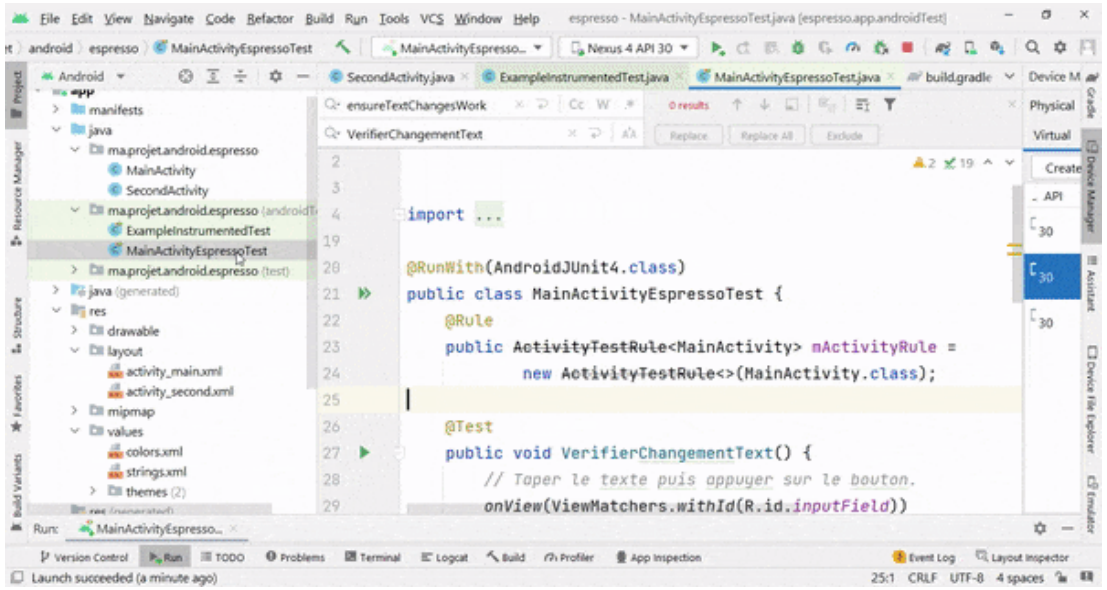
# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant Espresso

### TP n° 2 : scénarios de Test avec Espresso

Dans ce TP, vous allez développer une application mobile composée de deux activités. Ensuite, vous allez implémenter deux scénarios de Test pour vérifier le bon fonctionnement de l'application.

La figure suivante, présente le résultat de l'exécution des scénarios de Test.




Pour réaliser ce TP, suivre **les étapes indiquées** dans les diapositives suivantes.

PARTIE 4

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant Espresso

### Etape 1 : créer un nouveau projet

Créer un nouveau projet sous Android Studio :

1. Cliquer sur File, puis **New -> New Project** ;
2. Choisir l'activité **Empty** ;
3. Sélectionner comme langage Java ;
4. Choisir le **SDK minimum** en fonction de vos besoins.

### Etape 2 : créer la 1ère activité

Modifier le fichier **main_activity.xml** :

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   android:padding="10dp">
   <EditText
      android:id="@+id/inputField"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content" />
   <Button
      android:id="@+id/changeText"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onClick="onClick"
      android:text="Nouveau Bouton" />
   <Button
      android:id="@+id/switchActivity"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:onClick="onClick"
      android:text="Changer le Texte" />
</LinearLayout>
```

PARTIE 4

## Etape 3 : créer la classe MainActivity

Modifier la classe **MainActivity.java :**

```
public class MainActivity extends AppCompatActivity {
   EditText editText;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);
      editText = (EditText) findViewById(R.id.inputField);
   }

   public void onClick(View view) {
      switch (view.getId()) {
         case R.id.changeText:
            editText.setText("Dev mobile");
            break;
         case R.id.switchActivity:
            Intent intent = new Intent(this, SecondActivity.class);
            intent.putExtra("input", editText.getText().toString());
            startActivity(intent);
            break;
      }
   }
}
```

## Etape 4 : créer l'activité SecondActivity

Modifier le Layout de l'activité :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical">

   <TextView
      android:id="@+id/resultView"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="Large Text"
      android:textAppearance="?android:attr/textAppearanceLarge" />
</LinearLayout>
```

# ACTIVITÉ n° 2

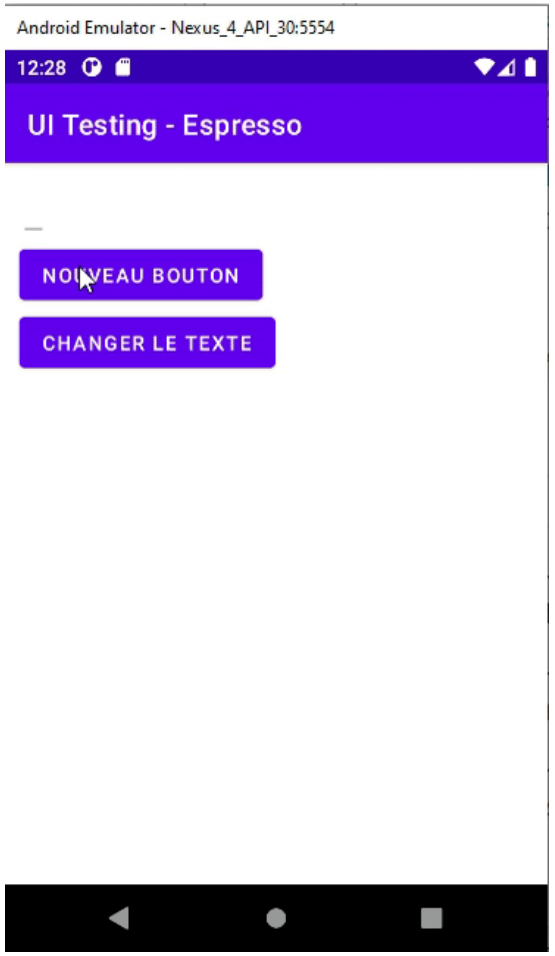
## Tester les interfaces d'une application en utilisant Espresso

### Etape 5 : créer la classe SecondActivity

Modifier la classe **SecondActivity.java** :

```
public class SecondActivity extends Activity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_second);
      TextView viewById = (TextView) findViewById(R.id.resultView);
      Bundle inputData = getIntent().getExtras();
      String input = inputData.getString("input");
      viewById.setText(input);
   }
}
```

### Etape 6 : exécuter l'application




PARTIE 4

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant Espresso

### Etape 7 : scénarios de test

Dans cette étape on demande d'implémenter deux scénarios de test :

**Scénario 1 :**

- Taper le texte puis appuyer sur le bouton « Nouveau bouton »
- Vérifier que le texte a été modifié

**Scénario 2 :**

- Taper le texte puis cliquer sur le bouton « Changer le texte »
- Vérifier que le texte a été envoyé vers la 2ème activité

PARTIE 4

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant Espresso

### Solution

```
@RunWith(AndroidJUnit4.class)
public class MainActivityEspressoTest {
    @Rule
    public ActivityTestRule<MainActivity> mActivityRule =
            new ActivityTestRule<>(MainActivity.class);

    @Test
    public void VerifierChangementText() {
        // Taper le texte puis appuyer sur le bouton.
        onView(ViewMatchers.withId(R.id.inputField))
                .perform(typeText("HELLO Dev"), closeSoftKeyboard());
        onView(withId(R.id.changeText)).perform(click());

        // Vérifier que le texte a été modifié.
        onView(withId(R.id.inputField)).check(matches(withText("Dev mobile")));
    }

    @Test
    public void changerTexteSecondActivity() {
        // Taper le texte puis cliquer sur le bouton.
        onView(withId(R.id.inputField)).perform(typeText("Nouveau Texte"),
                closeSoftKeyboard());
        onView(withId(R.id.switchActivity)).perform(click());

        // Cette vue est dans une activité différente, pas besoin de le signaler à
        // Espresso.
        onView(withId(R.id.resultView)).check(matches(withText("Nouveau Texte")));
    }
}
```

PARTIE 4

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant UI Automator

### Rappel : API d'UI Automator

Grâce aux API de UI Automator, il est possible d'écrire des tests solides sans connaître les détails de l'implémentation de l'application testée.

Utiliser l'ensemble d'API ci-dessous pour capturer les composants de l'interface utilisateur dans les applications mobiles :

- **UiDevice** : permet d'accéder aux informations d'état du périphérique, et permet également de simuler les actions de l'utilisateur sur l'appareil
- **UiCollection** : cette API permet d'énumérer les éléments d'interface utilisateur d'un conteneur pour compter ou cibler les sous-éléments en fonction de leur texte visible ou de leur propriété de description du contenu
- **UiObject** : représente un élément d'interface utilisateur visible sur le périphérique
- **UiScrollable** : recherche des éléments dans un conteneur d'interface utilisateur défilable
- **UiSelector** : utilisée pour interroger plus d'un élément d'interface utilisateur cible sur un périphérique
- **Configurator** : permet de définir les paramètres clés des tests UI Automator en cours d'exécution

Après avoir pris connaissance de ce qu'est UI Automator et de ses différentes API, allons voir comment réaliser l'automatisation d'une application Android à l'aide de UI Automator.

PARTIE 4

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant UI Automator

### TP n° 1 : énoncé

Dans ce TP nous allons implémenter trois scénarios de test dans un simple projet « HelloWorld ».

**Scénario 1 :**

- Tester le bouton de retour

**Scénario 2 :**

- Simuler l'orientation de l'appareil vers la gauche
- Ouvrir les notifications
- Ouvrir la fenêtre des paramètres généraux
- Simuler une courte action sur le bouton HOME

**Scénario 2 :**

- Ouvrir les notifications
- Simuler une action sur le bouton de retour
- Ouvrir la fenêtre des paramètres généraux
- Désactiver les capteurs et geler la rotation du dispositif à son état de rotation actuel
- Simuler l'orientation de l'appareil vers la gauche
- Simuler l'orientation de l'appareil vers la droite
- Simuler une action sur le bouton de retour
- Simuler une courte action sur le bouton HOME

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant UI Automator

### Solution

```
@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {
   @Rule
   public ActivityTestRule<MainActivity> mActivityRule =
         new ActivityTestRule<>(MainActivity.class);

   @After
   public void wrapUp() {
      System.out.println("Test done");
   }

   @Before
   public void setComponent() {
      System.out.println("Components initialize");
   }

   @Test
   public void testBackKeyPress() {
      UiDevice.getInstance(InstrumentationRegistry.getInstrumentation()).pressBack();

   }

  @Test
   public void testUi() throws RemoteException {
      UiDevice uiDevice = UiDevice.getInstance(InstrumentationRegistry.getInstrumentation());
      if (uiDevice.isScreenOn()) {
         uiDevice.setOrientationLeft();
         uiDevice.openNotification();
         uiDevice.openQuickSettings();
         uiDevice.pressHome();
      }
   }

   @Test
   public void testComplexUi() throws RemoteException {
      UiDevice uiDevice = UiDevice.getInstan
                              ce(InstrumentationRegistry.getInstrumentation());
      if (uiDevice.isScreenOn()) {
         uiDevice.openNotification();
         uiDevice.pressBack();
         uiDevice.openQuickSettings();
         uiDevice.freezeRotation();
         uiDevice.setOrientationLeft();
         uiDevice.setOrientationRight();
         uiDevice.pressBack();
         uiDevice.pressHome();
      }
   }
}
```

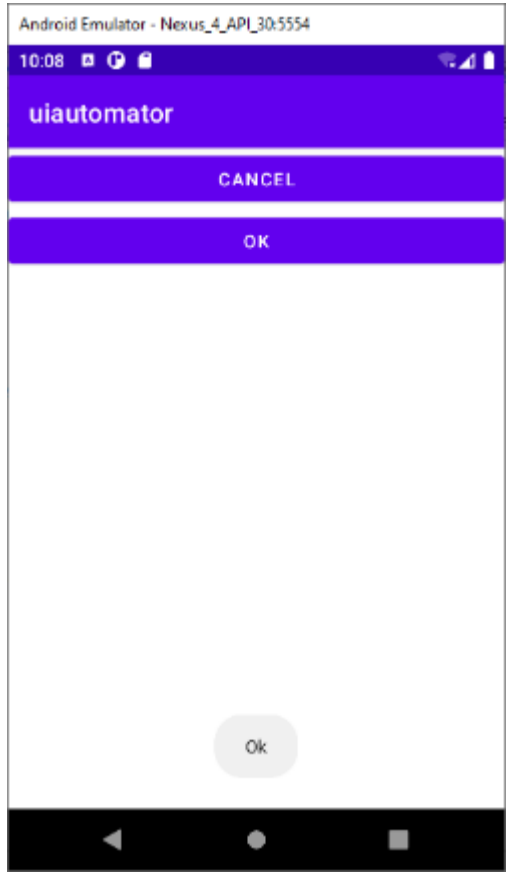
PARTIE 4

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant UI Automator

### TP n° 2 : énoncé

Dans ce TP nous allons simuler le clic sur un bouton.




On demande d'implémenter les étapes suivantes :

1. Initialiser l'instance UiDevice.
2. Démarrer à partir de l'écran d'accueil.
3. Attendre le lanceur.
4. Lancer l'application.
5. Effacer toutes les instances précédentes.
6. Attendre que l'application apparaisse.
7. Simuler un clic de l'utilisateur sur le bouton OK, si il existe.

PARTIE 4

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant UI Automator

PARTIE 4

### Solution

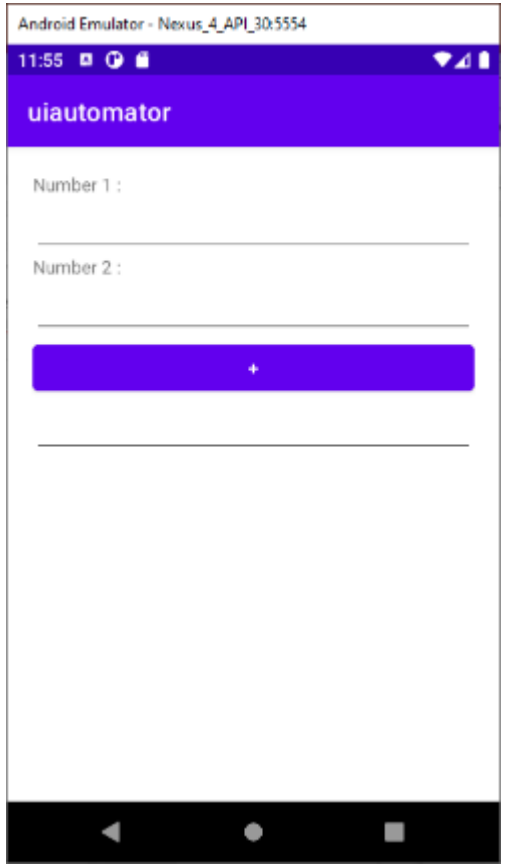
```
@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {
   private static final String BASIC_SAMPLE_PACKAGE
         = "ma.projet.android.uiautomator";
   private static final int LAUNCH_TIMEOUT = 5000;
   private UiDevice device;

   @Before
   public void startMainActivityFromHomeScreen() {
      // Initialiser l'instance UiDevice
      device = UiDevice.getInstance(InstrumentationRegistry.getInstrumentation());

      // Démarrer à partir de l'écran d'accueil
      device.pressHome();

      // Attendre le lanceur
      final String launcherPackage = device.getLauncherPackageName();
      assertThat(launcherPackage, notNullValue());
      device.wait(Until.hasObject(By.pkg(launcherPackage).depth(0)),
            LAUNCH_TIMEOUT);

      // Lancer l'application
      Context context = ApplicationProvider.getApplicationContext();
      final Intent intent = context.getPackageManager()
            .getLaunchIntentForPackage(BASIC_SAMPLE_PACKAGE);
      // Effacer toutes les instances précédentes
      intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
      context.startActivity(intent);
      // Attendre que l'application apparaisse
      device.wait(Until.hasObject(By.pkg(BASIC_SAMPLE_PACKAGE).depth(0)),
            LAUNCH_TIMEOUT);
   }
   @Test
   public void test() throws UiObjectNotFoundException {
      UiObject cancelButton = device.findObject(new UiSelector()
            .text("Cancel")
            .className("android.widget.Button"));
      UiObject okButton = device.findObject(new UiSelector()
            .text("OK")
            .className("android.widget.Button"));

      // Simuler un clic de l'utilisateur sur le bouton OK, si il existe.
      if(okButton.exists() && okButton.isEnabled()) {
         okButton.click();
      }
   }
}
```

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant UI Automator

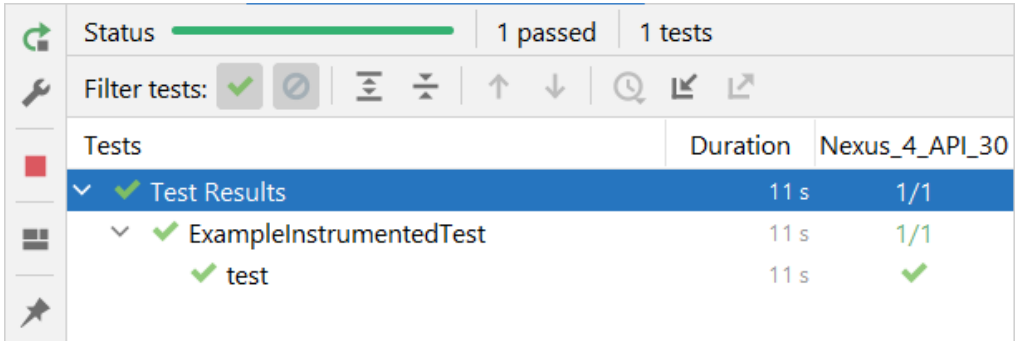
### TP n° 3 : énoncé

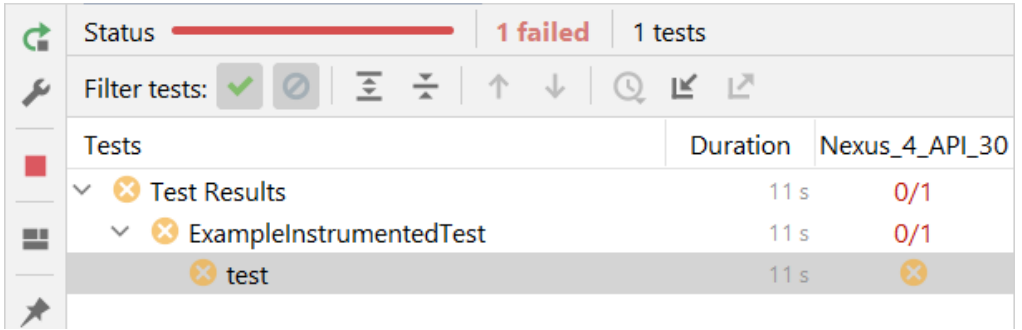
Dans ce TP, on demande de tester l'application suivante :




**Etape 1 :** développer une application mobile permettant d'effectuer la somme de deux nombres comme illustré dans la figure.

**Etape 2 :** tester l'application à l'aide de UI Automator.




PARTIE 4

# ACTIVITÉ n° 2

## Tester les interfaces d'une application en utilisant UI Automator

### Solution

**Etape 1 :** Ajouter les dépendances dans **build.gradle** (Module)

```
testImplementation 'JUnit:JUnit:4.13.2'
androidTestImplementation 'androidx.test.ext:JUnit:1.1.3'
androidTestImplementation 'androidx.test:rules:1.4.0'
androidTestImplementation 'androidx.test:runner:1.4.0'
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
androidTestImplementation "androidx.test.uiautomator:uiautomator:2.2.0"
```

**Etape 2 :** Créer la classe de Test dans le package (androidTest)

```
@RunWith(AndroidJUnit4.class)
public class ExampleInstrumentedTest {
    private static final String BASIC_SAMPLE_PACKAGE
            = "ma.projet.android.uiautomator";
    private static final int LAUNCH_TIMEOUT = 5000;
    private UiDevice device;

    @Before
    public void startMainActivityFromHomeScreen() {
        // Initialiser l'instance UiDevice
        device = UiDevice.getInstance(InstrumentationRegistry.getInstrumentation());

        // Démarrer à partir de l'écran d'accueil
        device.pressHome();

        // Attendre le lanceur
        final String launcherPackage = device.getLauncherPackageName();
        assertThat(launcherPackage, notNullValue());
        device.wait(Until.hasObject(By.pkg(launcherPackage).depth(0)),
                LAUNCH_TIMEOUT);

        // Lancer l'application
        Context context = ApplicationProvider.getApplicationContext();
        final Intent intent = context.getPackageManager()
                .getLaunchIntentForPackage(BASIC_SAMPLE_PACKAGE);
        // Effacer toutes les instances précédentes
        intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
        context.startActivity(intent);
        // Attendre que l'application apparaisse
        device.wait(Until.hasObject(By.pkg(BASIC_SAMPLE_PACKAGE).depth(0)),
                LAUNCH_TIMEOUT);
    }
    @Test
    public void test() throws UiObjectNotFoundException {
        UiObject number1 = device.findObject(new UiSelector()
                .resourceId("ma.projet.android.uiautomator:id/number1")
                .className("android.widget.EditText"));
        number1.setText(5+"");
        UiObject number2 = device.findObject(new UiSelector()
                .resourceId("ma.projet.android.uiautomator:id/number2")
                .className("android.widget.EditText"));
        number2.setText(7+"");
        UiObject plus = device.findObject(new UiSelector()
                .text("+")
                .className("android.widget.Button"));
        // Simuler un clic de l'utilisateur sur le bouton OK, si il existe.
        if(plus.exists() && plus.isEnabled()) {
            plus.click();
        }
        // Vérifier le résultat = 12
        UiObject2 result = device.findObject(By.res(BASIC_SAMPLE_PACKAGE, "result"));
        assertEquals("12", result.getText());
    }
}
```

PARTIE 4