RÉSUMÉ THÉORIQUE – FILIÈRE DÉVELOPPEMENT DIGITAL

Option – Applications mobiles

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

25 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
- Maitriser la communication entre composants applicatifs

6 heures

CHAPITRE 1

Maîtriser les architectures d'une application mobile Android

Ce que vous allez apprendre dans ce chapitre :

- Utilisation de l'architecture MVC
- Utilisation de l'architecture MVP
- Utilisation de l'architecture MVVM
- Utilisation de clean architecture

1 heure

CHAPITRE 1

Maîtriser les architectures d’une application mobile Android

1. **Utilisation de l’architecture MVC**
2. Utilisation de l’architecture MVP
3. Utilisation de l’architecture MVVM
4. Utilisation de clean architecture

Les modèles MVC, MVP, MVVM et Clean Arch.

Un modèle de conception est une forme réutilisable d'une solution à un problème de conception. C'est un style de codage par lequel nous pouvons gérer les différents composants du système que nous réalisons. Nous allons ici discuter de quatre de ces modèles de conception : MVC, MVP, MVVM et Clean Architecture.

- MVC (Model View Controller)
- MVP (Model View Presenter)
- MVVM (Model-View-View-Model)
- Clean Architecture Pattern

Le modèle MVC (Model View Controller)

Il s'agit de l'une des approches les plus utilisées dans le développement de logiciels. Le MVC se compose de trois éléments principaux :

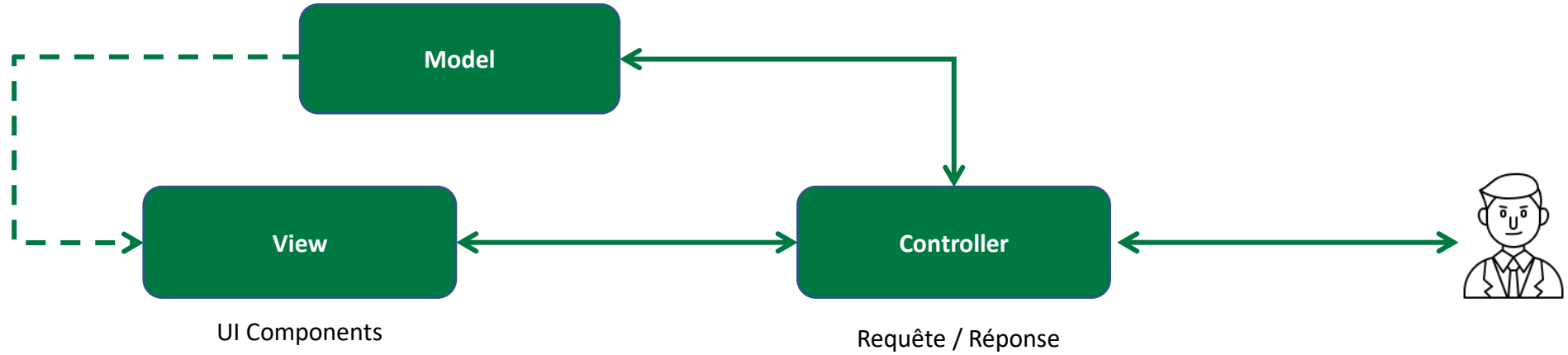
- **Model :** Le modèle représente l'objet dans l'application. Il contient la logique de l'endroit où les données doivent être récupérées. Il peut également contenir la logique par laquelle le contrôleur peut mettre à jour la vue. Dans Android, le modèle est principalement représenté par des classes d'objets.

- **View :** La vue est constituée des composants qui peuvent interagir avec l'utilisateur et est responsable de la façon dont le modèle est affiché dans l'application. Dans Android, la vue est principalement représentée par le XML où les mises en page peuvent être conçues.

- **Controller :** Le contrôleur agit comme un médiateur entre le modèle et la vue. Il contrôle le flux de données dans l'objet modèle et met à jour la vue lorsque les données changent. Dans Android, le contrôleur est principalement représenté par les activités et les Fragments.

Le modèle MVC (Model View Controller)

Tous ces composants interagissent les uns avec les autres et exécutent des tâches spécifiques, comme le montre la figure suivante :



Il a été remarqué qu'Android n'est pas capable de suivre complètement l'architecture MVC, car Activity/Fragment peut agir à la fois comme contrôleur et comme vue, ce qui fait que tous les codes sont regroupés au même endroit. L'Activity/Fragment peut être utilisé pour dessiner plusieurs vues pour un seul écran dans une application, ainsi les différents appels de données et les vues sont peuplés au même endroit. Par conséquent, pour résoudre ce problème, nous pouvons utiliser différents modèles de conception ou mettre en œuvre MVC soigneusement en prenant soin des conventions et en suivant les directives de programmation appropriées.

CHAPITRE 1

Maîtriser les architectures d’une application mobile Android

1. Utilisation de l’architecture MVC
2. **Utilisation de l’architecture MVP**
3. Utilisation de l’architecture MVVM
4. Utilisation de clean architecture

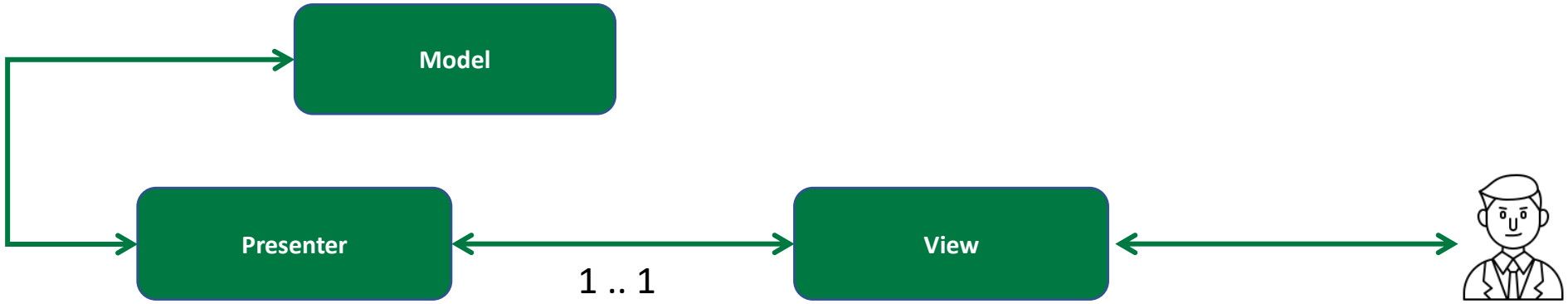
Le modèle MVP (Model View Presenter)

Model View Presenter (MVP) est dérivé du modèle MVC. Le MVP est utilisé pour minimiser la forte dépendance de la vue, ce qui est le cas dans le MVC. Il sépare la vue et le modèle en utilisant le présentateur. Le présentateur décide de ce qui doit être affiché sur la vue :

- **Model :** Le modèle représente les objets de l'application. Il contient la logique de l'endroit où les données doivent être récupérées.
- **View :** La vue rend les informations aux utilisateurs et contient un composant d'interface utilisateur (fichiers XML), une activité, des Fragments et un dialogue sous la couche de vue. Elle n'a pas d'autre logique implémentée.
- **Presenter :** La couche de présentateur exécute la tâche du contrôleur et sert de médiateur entre la vue et le modèle. Mais contrairement au contrôleur, elle ne dépend pas de la vue. La vue interagit avec le présentateur pour que les données soient affichées, et le présentateur prend ensuite les données du modèle et les renvoie à la vue dans un format présentable. Le présentateur ne contient aucun composant d'interface utilisateur ; il se contente de manipuler les données du modèle et de les afficher sur la vue.

Le modèle MVP (Model View Presenter)

L'interaction entre les différents composants du MVP est présentée dans la figure suivante :



Dans la conception MVP, le présentateur communique avec la vue par le biais d'interfaces. Les interfaces sont définies dans la classe du présentateur, à laquelle il transmet les données requises. L'activité/Fragment ou tout autre composant de la vue implémente les interfaces et rend les données de la manière qu'il souhaite. La connexion entre le présentateur et la vue est un à un.

CHAPITRE 1

Maîtriser les architectures d’une application mobile Android

1. Utilisation de l’architecture MVC
2. Utilisation de l’architecture MVP
3. **Utilisation de l’architecture MVVM**
4. Utilisation de clean architecture

Le modèle MVVM (Model View View Model)

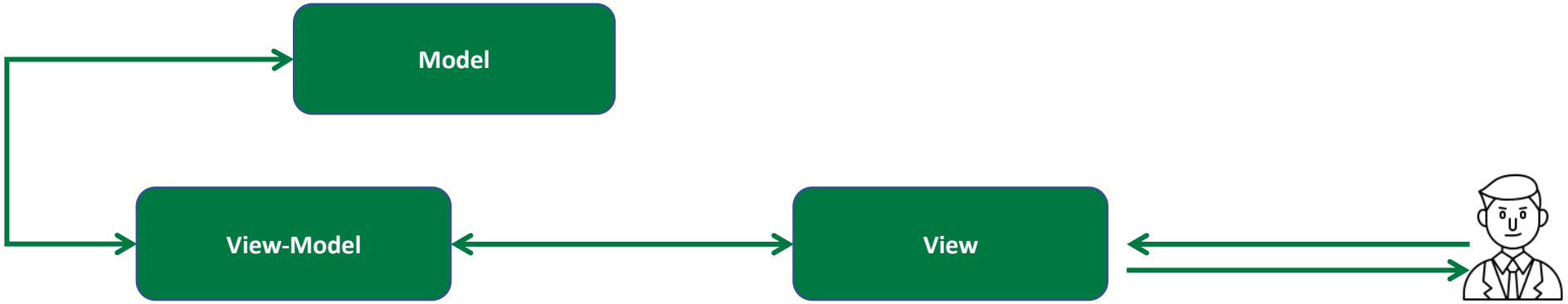
MVVM est l'abréviation de Model-View-View-Model. Il est similaire au modèle MVC, à la seule différence qu'il possède une liaison de données bidirectionnelle avec la vue et le modèle de vue. Les modifications de la vue sont propagées par le modèle de vue, qui utilise un modèle d'observateur pour communiquer entre le modèle de vue et le modèle. Dans ce cas, la vue est complètement isolée du modèle.

MVVM comporte les éléments suivants :

- **Model :** Le modèle représente les objets de l'application. Il contient la logique de l'endroit où les données doivent être récupérées.
- **View :** La vue est similaire à celle du modèle MVC, qui rend les informations aux utilisateurs et contient un fichier .xml de composant d'interface utilisateur, une activité, des Fragments et un dialogue sous la couche de vue. Elle n'a pas d'autre logique mise en œuvre.
- **View-model** : Le modèle de vue aide à maintenir l'état de la vue et apporte des modifications au modèle en fonction des données obtenues de la vue.

Le modèle MVVM (Model View View Model)

Le principal avantage de l'utilisation de MVVM est qu'il permet des changements automatiques dans la vue via le modèle de vue :



De nombreuses vues peuvent être liées à un modèle de vue, ce qui crée une relation de *plusieurs à un* entre la vue et un modèle de vue. En outre, une vue possède des informations sur le modèle de vue, mais le modèle de vue ne possède aucune information sur la vue. La vue n'est pas responsable de l'état des informations ; celles-ci sont gérées par le modèle de vue, et la vue et le modèle ne sont reflétés que par les modifications apportées au modèle de vue.

Comparaison des trois architectures

	MVC	MVP	MVVM
Maintenance	Difficile à maintenir	Facile à maintenir	Facile à maintenir
Difficulté	Facile à apprendre	Facile à apprendre	Plus difficile à apprendre en raison des fonctions supplémentaires
Type de relation	Relation plusieurs à un entre le contrôleur et la vue	Relation 1 à 1 entre le présentateur et la vue	Relation plusieurs à un entre la vue et le modèle de vue
Tests unitaires	En raison d'un couplage étroit, il est difficile à tester avec des tests unitaires	Bonnes performances	Excellentes performances
Point d'accès	Contrôleur	Vue	Vue
Références	La vue n'a pas de référence au contrôleur	La vue fait référence au presenter	La vue fait référence au modèle de vue

CHAPITRE 1

Maîtriser les architectures d’une application mobile Android

1. Utilisation de l’architecture MVC
2. Utilisation de l’architecture MVP
3. Utilisation de l’architecture MVVM
4. **Utilisation de clean architecture**

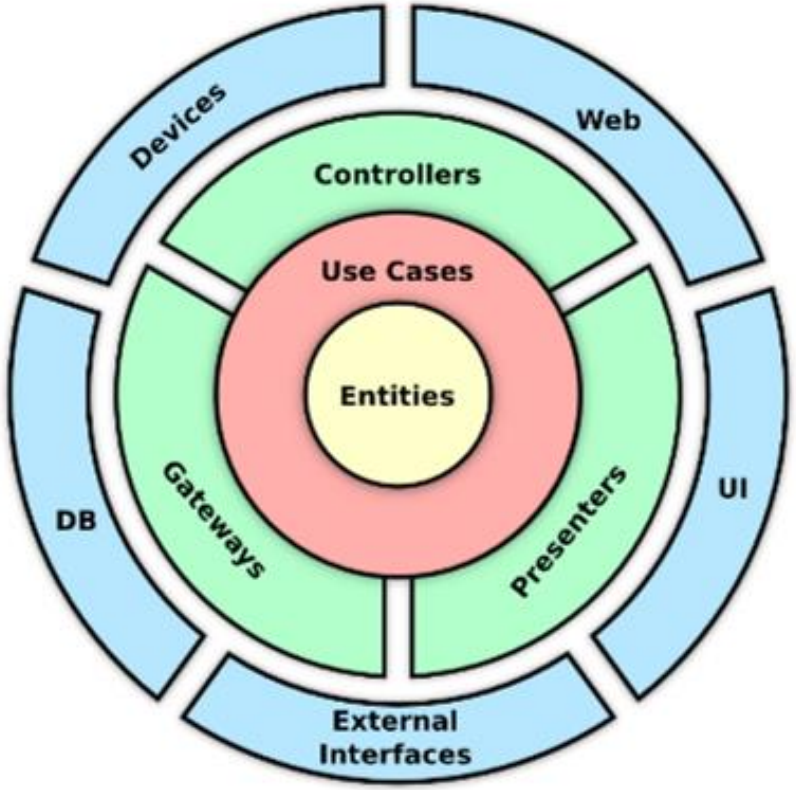
Modèle d'architecture propre (Clean Architecture Pattern)

Le modèle d'architecture propre, dans ses termes les plus simples, signifie écrire un code propre, en le séparant en couches, la couche externe étant vos implémentations et la couche interne étant la logique commerciale. Une interface relie ces deux couches, contrôlant la façon dont les couches externes utilisent les couches internes.

Ce type de modèle d'architecture de code est également connu sous le nom **d'architecture en Onion** en raison de ses différentes couches, comme le montre la figure suivante.

Les couches internes n'ont aucune idée des couches externes.

Les couches externes utilisent les composants des couches internes en fonction de leurs besoins. Ce qui signifie que les couches externes dépendent des implémentations de la logique métier des couches internes. La dépendance est donc orientée vers l'intérieur.



Réf : Clean Architecture: Patterns, Practices, and Principles

PARTIE 1

Modèle d'architecture propre (Clean Architecture Pattern)

- **Entities (couche 0) :** Les entités sont l'une des deux couches de domaine du cercle intérieur qui représentent la logique métier et de domaine. Il s'agit de règles métier à l'échelle de l'entreprise, ou de données qui seront toujours vraies ou statiques pour cette application. Les entités peuvent être des objets avec des méthodes ou simplement une collection de structures de données et de fonctions. Elles encapsulent les éléments les plus généraux ou de plus haut niveau et sont donc les moins susceptibles de changer suite à une modification d'une couche externe.

  Par exemple, les fonctionnalités de base d'une application ne seraient pas modifiées par le passage de framework frontal React à Angular.

- **Use cases (couche 1) :** Les cas d'utilisation constituent la deuxième couche du domaine. Elles définissent les règles de gestion spécifiques à l'application. Elles encapsulent et mettent en œuvre tous les cas d'utilisation approuvés pour l'application. Les cas d'utilisation contrôlent le flux vers et depuis les entités et peuvent demander aux entités d'utiliser leurs règles à l'échelle de l'entreprise pour accomplir certaines tâches utilisateur.

  Les changements dans cette couche n'affecteront pas les entités ou les couches plus externes. Toutefois, cette couche devra être modifiée si des couches externes sont modifiées.

Modèle d'architecture propre (Clean Architecture Pattern)

- **Adaptateurs d'interface (couche 2) :** Cette couche adapte l'entrée dans une forme qui est plus utilisable par les couches de cas d'utilisation et d'entités. Elle formate également la sortie des entités ou des cas d'utilisation dans une forme qui convient le mieux aux canaux externes. La couche des adaptateurs est la limite effective entre les couches du cercle intérieur et extérieur. Aucun code à l'intérieur de cette couche ne doit connaître ou référencer quoi que ce soit de plus externe que cette couche.

  Les adaptateurs d'interface peuvent être considérés comme un convertisseur qui convertit et relaie les informations de la manière la plus utilisable par les couches internes et externes respectivement.

- **Framework et Drivers (couche 3) :** Framework et Drivers est la couche de présentation qui est généralement composée de frameworks et d'outils tels que des bases de données, des frameworks web, etc. Cette couche ne comporte pas beaucoup de code mais contient plutôt toutes les références concrètes nécessaires à des détails spécifiques tels que des opérations spécifiques à la base de données ou des commandes spécifiques au framework actuel.

  Par exemple, elle contient entièrement l'architecture MVC d'une interface graphique.

Avantages d'architecture propre (Clean Architecture Pattern)

- **Hautement testable :** L'architecture propre est construite de A à Z pour être testée. Il est possible de créer des cas de test pour chaque couche afin de déterminer où les erreurs se produisent dans le cercle.
- **Indépendante du framework :** L'architecture propre ne s'appuie pas sur les outils d'un framework spécifique et n'utilise pas le framework comme dépendance dans le code.
- **Indépendante de la base de données :** La majorité des applications ne connaissent pas ou n'ont pas besoin de connaître la base de données dont elles sont issues. Cela permet d'adopter une nouvelle base de données sans modifier la majorité du code source.
- **Indépendante de l'IU :** Les frameworks UI existent sur la couche la plus externe et ne sont donc qu'un présentateur pour les données transmises par les couches internes. Ainsi, il est possible de changer l'interface utilisateur à tout moment.

CHAPITRE 2

Connaitre le cycle de vie des composants applicatifs

Ce que vous allez apprendre dans ce chapitre :

- Utilisation de l'Activity
- Utilisation du Fragment
- Manipulation des Services
- Création du broadcast receiver

3 heures

CHAPITRE 2

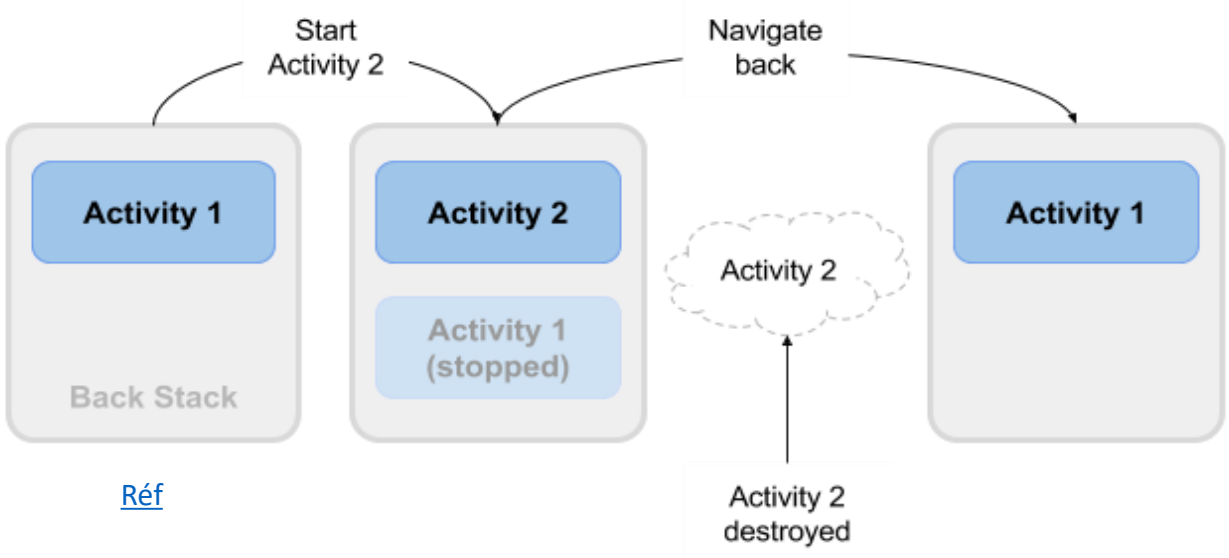
Connaitre le cycle de vie des composants applicatifs

1. **Utilisation de l’Activity**
2. Utilisation du Fragment
3. Manipulation des Services
4. Création du broadcast receiver

PARTIE 1

Qu'est-ce que le cycle de vie des activités ?

- L'ensemble des états dans lesquels une activité peut se trouver au cours de sa vie, depuis sa création jusqu'à sa destruction.
- Plus formellement :
  - Un **cycle de vie** est un **graphe dirigé** contenant tous les **états** dans lesquels une activité peut se trouver, ainsi que les rappels associés à la **transition** entre les différents états (voir la diapositive 27 : Graphique des états et des rappels d'activité).



Réf

États d'activité et visibilité des applications

- Créé (Created) = pas encore visible
- Démarré (Started) = visible
- Repris (Resume) = visible
- Mis en pause (Paused) = partiellement invisible
- Stoppé (Stopped) = caché
- Détruit (Destroyed) = disparu de la mémoire

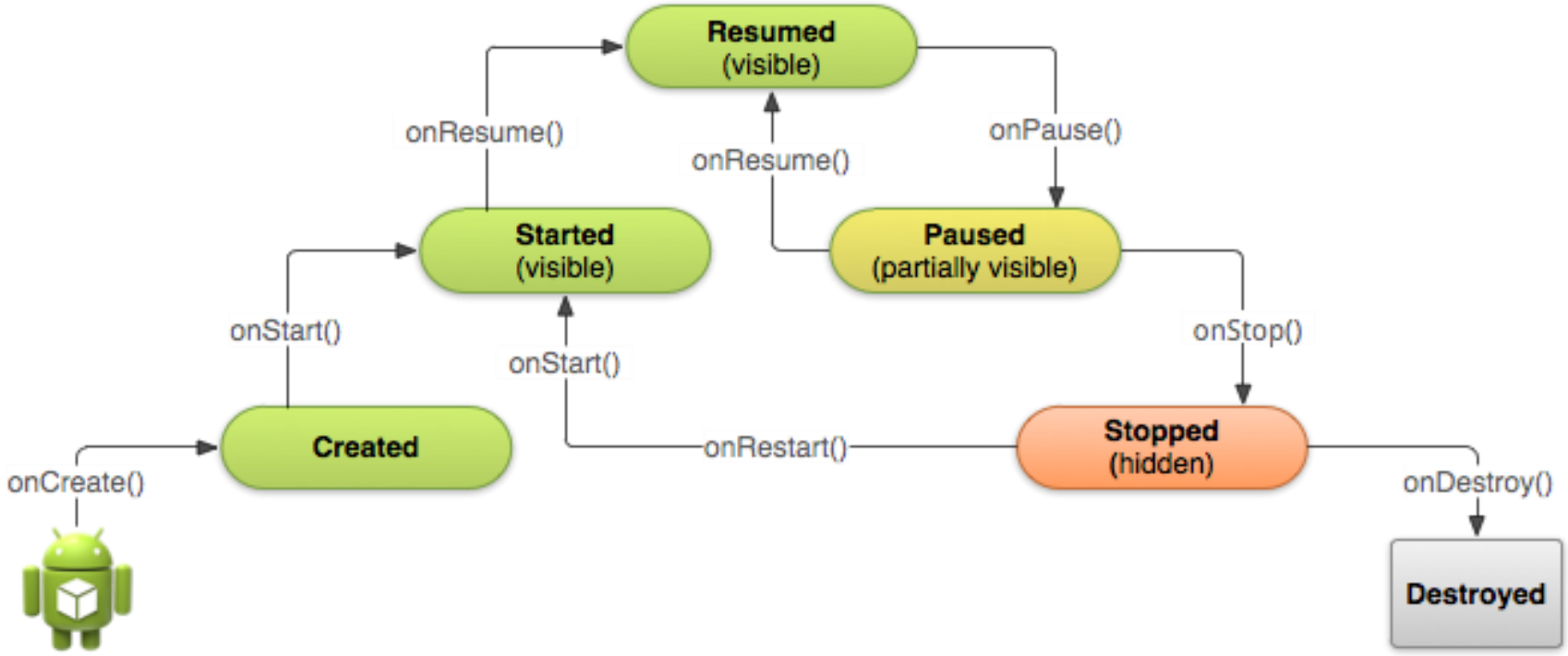
Les **changements d'état** sont déclenchés par l'action de **l'utilisateur**.

Les **changements de configuration** tels que la **rotation** du dispositif ou **l'action** du **système**.

Callbacks et quand ils sont appelés

- onCreate(Bundle savedInstanceState) - initialisation statique
  - onStart() - lorsque l'activité (l'écran) devient visible
  - onRestart() - appelée si l'activité a été arrêtée (appelle onStart())
    - onResume() - commence à interagir avec l'utilisateur
    - onPause() - sur le point de reprendre une activité PRÉCÉDENTE
  - onStop() - n'est plus visible, mais existe toujours et toutes les informations d'état sont conservées
- onDestroy() - dernier appel avant que le système Android ne détruise l'activité

Graphique des états et des rappels d'activité



Réf

Implémentation et remplacement des callbacks

- Seul **onCreate()** est nécessaire
- Remplacer les autres rappels pour modifier le comportement par défaut
- **onCreate() –> Created :**
  - Appelé lorsque l'activité est créée pour la première fois, par exemple lorsque l'utilisateur touche l'icône du lanceur
  - Effectue toute la configuration statique : création des vues, liaison des données aux listes...
  - Appelé une seule fois pendant la durée de vie d'une activité
  - Prend un Bundle avec l'état précédemment gelé de l'activité, s'il y en a un
  - L'état créé est toujours suivi par **onStart()**.
- **onCreate(Bundle savedInstanceState) :**

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    // L'activité est en cours de création.
}
```

onStart() –> Started

- Appelé lorsque l'activité devient visible pour l'utilisateur
- Peut être appelé plusieurs fois au cours du cycle de vie
- Suivi de **onResume()** si l'activité passe au premier plan, ou de **onStop()** si elle devient cachée

```
@Override
protected void onStart() {
    super.onStart();
    // L'activité est sur le point de devenir visible.
}
```

onRestart() –> Started

- Appelé après l'arrêt d'une activité, immédiatement avant son redémarrage ;
- État transitoire ;
- Toujours suivi de **onStart().**

```
@Override
protected void onRestart() {
    super.onRestart();
    // L'activité se situe entre l'arrêt et le démarrage.
}
```

onResume() –> Resumed/Running

- Appelé lorsque l'activité commence à interagir avec l'utilisateur
- L'activité est passée au sommet de la pile d'activités
- Commence à accepter les entrées de l'utilisateur
- État d'exécution
- Toujours suivi de **onPause().**

```
@Override
protected void onResume() {
    super.onResume();
    // L'activité est devenue visible
    // Elle est maintenant "reprise".
}
```

onPause() –> Paused

- Appelé lorsque le système est sur le point de reprendre une activité précédente
- L'activité est partiellement visible mais l'utilisateur quitte l'activité
- Généralement utilisé pour valider les modifications non sauvegardées des données persistantes, arrêter les animations et tout ce qui consomme des ressources
- Les implémentations doivent être rapides car l'activité suivante n'est pas reprise avant le retour de cette méthode
- Suivi par **onResume()** si l'activité revient à l'avant, ou **onStop()** si elle devient invisible pour l'utilisateur

```
@Override
protected void onPause() {
    super.onPause();
    // Une autre activité est en train de prendre le focus
    // Cette activité est sur le point d'être "mise en pause".
}
```

onStop() –> Stopped

- Appelé lorsque l'activité n'est plus visible pour l'utilisateur
- Une nouvelle activité est lancée, une activité existante est placée devant celle-ci ou celle-ci est détruite
- Opérations qui étaient trop lourdes pour **onPause()**
- Suivie par **onRestart()** si l'activité revient pour interagir avec l'utilisateur, ou **onDestroy()** si l'activité disparaît

```
@Override
protected void onStop() {
    super.onStop();
    // L'activité n'est plus visible
    // elle est maintenant "arrêtée".
}
```

onDestroy() –> Destroyed

- Dernier appel avant la destruction de l'activité
- L'utilisateur revient à l'activité précédente ou modifie la configuration
- L'activité se termine ou le système la détruit pour gagner de l'espace
- Appeler la méthode **isFinishing()** pour vérifier
- Le système peut détruire l'activité sans l'appeler, alors utiliser **onPause()** ou **onStop()** pour sauvegarder les données ou l'état

```
@Override
protected void onDestroy() {
    super.onDestroy();
    // L'activité est sur le point d'être détruite.
}
```

Quand la configuration change-t-elle ?

Les changements de configuration invalident la disposition actuelle ou d'autres ressources de l'activité lorsque l'utilisateur :

- Fait pivoter l'appareil
- Choisit une langue système différente, ce qui entraîne une modification des paramètres locaux
- Entre en mode multifenêtre (à partir d'Android 7)

Que se passe-t-il en cas de changement de configuration ?

Au changement de configuration, Android :

1. Arrête l'activité en appelant :

- onPause()
- onStop()
- onDestroy()

2. Redémarre l'activité en appelant :

- onCreate()
- onStart()
- onResume()

État de l'instance d'activité

- Les informations sur l'état sont créées pendant que l'activité est en cours, comme un compteur, un texte utilisateur, une progression de l'animation;
- L'état est détruit lorsque :
    - L'appareil est tourné
    - La langue change
    - L'on appuie sur le bouton arrière
    - Le système efface la mémoire

Sauvegarde et restauration de l'état de l'activité

- Le système sauvegarde uniquement :
    - L'état des vues avec un ID unique (android:id), comme le texte saisi dans EditText
    - L'intention qui a lancé l'activité et les données dans ses extras
- L'utilisateur est responsable de la sauvegarde des autres activités et des données de progression de l'utilisateur
- Implémenter **onSaveInstanceState()** dans l'activité :
    - Appelée par le runtime Android lorsqu'il y a une possibilité que l'activité soit détruite
    - Sauvegarde les données uniquement pour cette instance de l'activité pendant la session en cours

```
@Override
public void onSaveInstanceState(Bundle outState) {
    super.onSaveInstanceState(outState);

    // Ajouter les informations pour sauvegarder le compteur HelloToast
    // dans le bundle outState
    outState.putString("count",
                String.valueOf(mShowCount.getText()));
}
```

Sauvegarde et restauration de l'état de l'activité

- Deux façons de récupérer le Bundle sauvegardé :
  - Dans la méthode préférée **onCreate(Bundle mySavedState),** afin de garantir que l'interface utilisateur, y compris tout état sauvegardé, soit de nouveau opérationnelle aussi rapidement que possible.
  - Implémentation d'un callback (appelé après **onStart()) onRestoreInstanceState**(**Bundle mySavedState**).

```
@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   mShowCount = findViewById(R.id.show_count);

   if (savedInstanceState != null) {
       String count = savedInstanceState.getString("count");
       if (mShowCount != null)
           mShowCount.setText(count);
   }
}
```

Sauvegarde et restauration de l'état de l'activité

- onRestoreInstanceState(Bundle state) :

```
@Override
public void onRestoreInstanceState (Bundle mySavedState) {
   super.onRestoreInstanceState(mySavedState);

   if (mySavedState != null) {
       String count = mySavedState.getString("count");
       if (count != null)
           mShowCount.setText(count);
   }
}
```

État de l'instance et redémarrage de l'application

- Lors de l'arrêt et du redémarrage d'une nouvelle session d'application, les états de l'instance d'activité sont perdus et les activités reprennent leur apparence par défaut.
- Lorsque des données utilisateur doivent être sauvegardées entre deux sessions d'application, utiliser des préférences partagées ou une base de données.

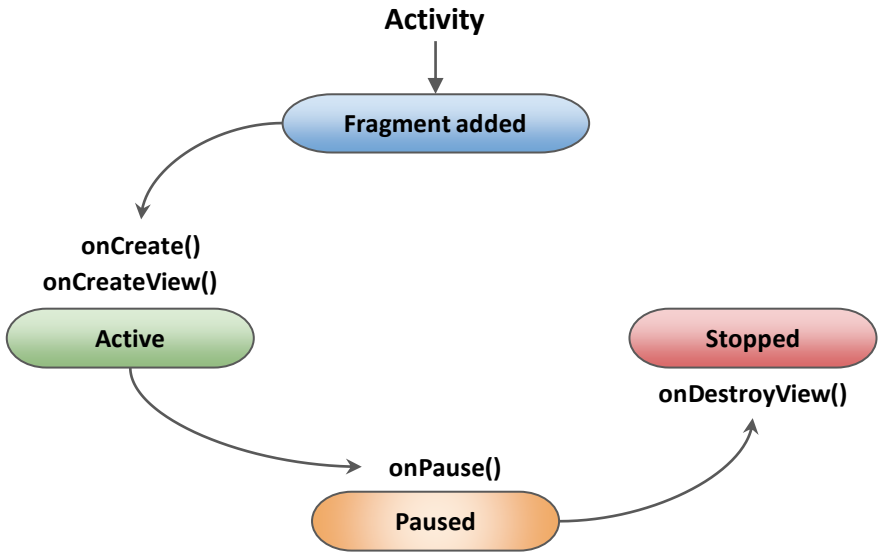
CHAPITRE 2

Connaitre le cycle de vie des composants applicatifs

1. Utilisation de l'Activity
2. **Utilisation du Fragment**
3. Manipulation des Services
4. Création du broadcast receiver

Classe Fragment

- Contient une partie de l'interface utilisateur et de son comportement
- Possède ses propres états de cycle de vie (comme une activité)
- Réutilisable : un Fragment peut être partagé entre plusieurs activités
- Chaque instance de Fragment est exclusivement liée à l'activité hôte
- Le code du Fragment définit la disposition et le comportement
- Représente des sections d'une interface utilisateur pour différentes mises en page (Layout)

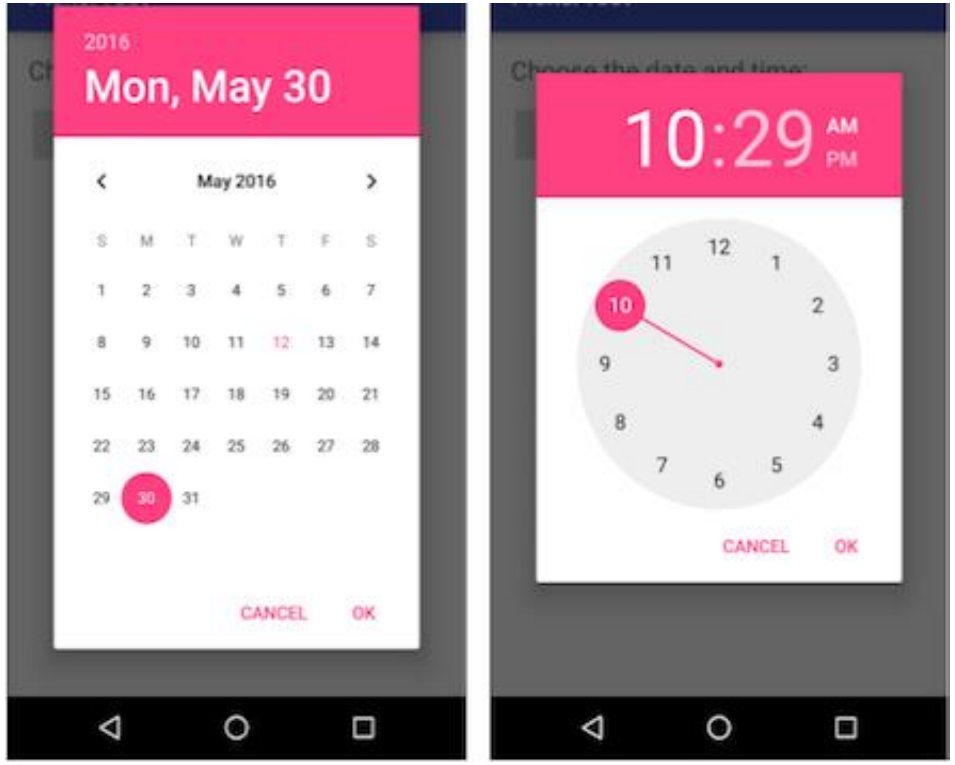


Réf

PARTIE 1

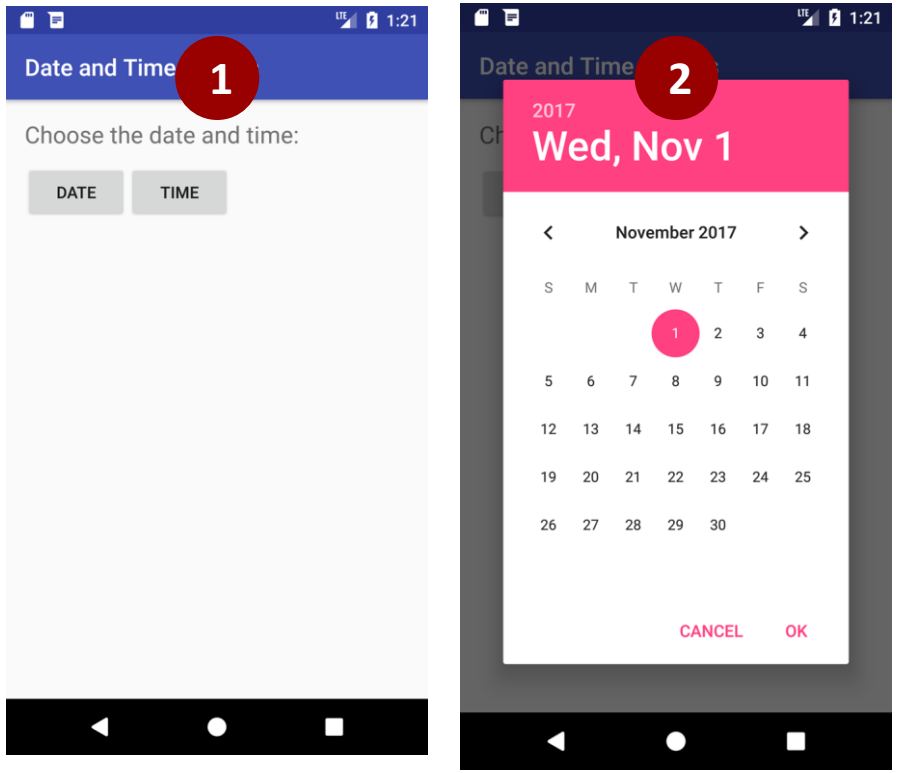
Exemple : Les sélecteurs utilisent DialogFragment

- Sélecteurs de date et d'heure : Extension de **DialogFragment** (sous-classe de **Fragment**).



DialogFragment hébergé par l'activité

1. Activité avant l'ajout du Fragment de sélecteur de date ;
2. Le Fragment du sélecteur de date apparaît en haut de l'activité.



Ajouter un Fragment à une activité

- Partie statique de l'interface utilisateur (dans le layout de l'activité) :
    - A l'écran pendant tout le cycle de vie de l'activité
- Partie dynamique de l'interface utilisateur :
    - Ajoutée et retirée pendant le déroulement de l'activité

Avantages de l'utilisation de Fragments

- Réutiliser un Fragment dans plus d'une activité
- Ajouter ou supprimer dynamiquement selon les besoins
- Intégrer une mini-UI dans une activité
- Conserver les instances de données après un changement de configuration
- Représenter des sections d'une mise en page pour différentes tailles d'écran

Étapes de l'utilisation d'un Fragment

1. Créer une sous-classe de **Fragment**
2. Créer un layout pour le **Fragment**
3. Ajouter un Fragment à une activité hôte :
    - Statiquement dans le layout
    - Dynamiquement en utilisant les transactions du Fragment

Ajouter un nouveau Fragment dans Android Studio

- Déplacer dans app > java dans le projet et sélectionner le nom du package
- Choisir **File > New > Fragment > Fragment (Blank)**
- Cocher l'option **Create layout XML** pour la mise en page **(layout)**
- Autres options :
    - Inclure les méthodes de la fabrique du Fragment pour inclure une méthode **newInstance()** afin d'instancier le Fragment
    - Inclure des callbacks d'interface pour définir une interface avec des callbacks

Nouveau Fragment

```
public class SimpleFragment extends Fragment {

    public SimpleFragment() {

        // Constructeur public vide obligatoire

    }

    ...

}
```

Étendre la classe Fragment

- Étendre la classe Fragment :
  - public class SimpleFragment extends Fragment
- Étendre une sous-classe spécifique de Fragment :
  - **DialogFragment** : Dialogue flottant (exemples : sélecteurs de date et d'heure)
  - **ListFragment** : Liste d'éléments gérés par l'adaptateur
  - **PreferenceFragment** : Hiérarchie d'objets de préférence (utile pour les paramètres)

Créer un layout pour un Fragment

- L'option **Create layout XML** ajoute un layout XML
- Le rappel du Fragment **onCreateView()** crée la vue
- Remplacer cette fonction pour inflater la disposition du Fragment
- Retourner la vue : racine de layout du Fragment

Inflater le layout de Fragment

```
@Override
public View onCreateView(LayoutInflater inflater,
            ViewGroup container, Bundle savedInstanceState) {
    // Inflater le layout du Fragment et la renvoie comme
    // vue racine.
    return inflater.inflate(R.layout.Fragment_simple,
            container, false);
}
```

- Le layout de Fragment est inséré dans le conteneur **ViewGroup** dans le layout Activity
- **LayoutInflater** inflate le layout et renvoie la racine de layout de la vue à l'activité
- **Bundle savedInstanceState** sauvegarde l'instance précédente de Fragment

return inflater.inflate(R.layout.Fragment_simple, container, false)

- ID de ressource de **layout (R.layout.Fragment_simple)** ;
- **ViewGroup** qui sera le parent de layout **inflaté** (conteneur)
- Booléen : indique si la vue doit être attachée au parent :
    - Devrait être false
    - Si un nouveau un Fragment est ajouté dans le code, ne pas passer true (crée un **ViewGroup** redondant)

Ajouter un Fragment à une activité

- Ajouter statiquement dans le layout de l'activité, visible pendant tout le cycle de vie de l'activité
- Ajouter (ou supprimer) dynamiquement, selon les besoins, au cours du cycle de vie de l'activité, à l'aide de transactions de Fragments

Ajouter un Fragment de manière statique

1. Déclarer le Fragment à l'intérieur du layout de l'activité (activity_main.xml) en utilisant la balise **<Fragment>**.
2. Spécifier les propriétés de layout pour le Fragment comme s'il s'agissait d'une vue.

Exemple de Fragment statique

Ajout de **SimpleFragment** à la mise en page de l'activité :

```
<Fragment android:name="com.example.appname.SimpleFragment"
            android:id="@+id/simple_Fragment"
            android:layout_weight="2"
            android:layout_width="0dp"
            android:layout_height="match_parent" />
```

Ajouter un Fragment de façon dynamique

1. Spécifier le **ViewGroup** pour le Fragment dans le layout.
2. Instanciation du Fragment dans l'activité.
3. Instancier le **FragmentManager** :
    - Utiliser getSupportFragmentManager() pour la compatibilité
4. Utiliser les transactions de Fragments.

Spécifier un ViewGroup pour le Fragment

Specify `ViewGroup` to place `Fragment` (comme `FrameLayout`) :

```
<FrameLayout
    android:id="@+id/Fragment_container"
    android:name="SimpleFragment"
    tools:layout="@layout/Fragment_simple"
    ... />
```

PARTIE 1

Ajouter un Fragment à une activité

Instancier le Fragment

1. Créer la méthode factory **newInstance()** dans Fragment :

```
public static SimpleFragment newInstance() {
        return new SimpleFragment();
}
```

2. Dans Activity, instancier le Fragment en appelant **newInstance()** :

```
SimpleFragment Fragment = SimpleFragment.newInstance();
```

Instancier FragmentManager

Dans l'activité, récupérer une instance de **FragmentManager** avec **getSupportFragmentManager()** :

```
FragmentManager FragmentManager =
                    getSupportFragmentManager() ;
```

Utiliser la version de la bibliothèque de support - **getSupportFragmentManager()** plutôt que **getFragmentManager()**- pour assurer la compatibilité avec les versions antérieures d'Android.

Utiliser les transactions Fragments

Les opérations sur les Fragments sont regroupées dans une transaction :

- Démarrer la transaction avec **beginTransaction()**
- Effectuer toutes les opérations sur les Fragments (ajout, suppression, etc.)
- Terminer la transaction avec **commit()**

Opérations des transactions Fragments

- Ajouter un Fragment avec **add()**
- Supprimer un Fragment avec **remove()**
- Remplacer un Fragment par un autre en utilisant **replace()**
- Masquer et afficher un Fragment avec **hide()** et **show()**
- Ajouter une transaction de Fragment à la pile arrière en utilisant : **addToBackStack(null)**

PARTIE 1

Exemples d'un Fragment transaction

Exemple 1 :

```
FragmentTransaction FragmentTransaction =
            FragmentManager.beginTransaction();
FragmentTransaction
            .add(R.id.Fragment_container, Fragment)
            .addToBackStack(null)
            .commit();
```

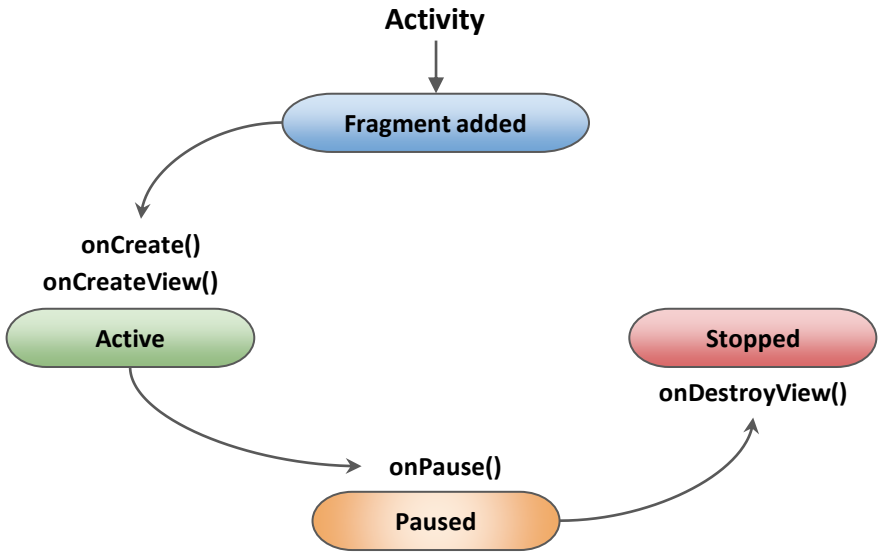
- Les arguments de **add()** :
  - Le **ViewGroup (Fragment_container)** ;
  - Le Fragment à ajouter.
- **addToBackStack**(**null**) :
  - Ajouter la transaction à la pile arrière des transactions du Fragment ;
  - Pile arrière gérée par l'activité ;
  - L'utilisateur peut appuyer sur le bouton Back pour revenir à l'état précédent du Fragment.

Exemple 2 : Suppression d'un Fragment

```
SimpleFragment Fragment = (SimpleFragment) FragmentManager
            .findFragmentById(R.id.Fragment_container);
if (Fragment != null) {
     FragmentTransaction FragmentTransaction =
                  FragmentManager.beginTransaction();
     FragmentTransaction.remove(Fragment).commit();
}
```

Le cycle de vie des Fragments

- Le cycle de vie des Fragments est similaire à celui des activités
- Les callbacks du cycle de vie définissent le comportement du Fragment dans chaque état
- Activité (hôte) ajoute Fragment
  - Etats dans lesquels un Fragment peut se trouver :
    - Actif (ou repris)
    - En pause
    - Arrêté



PARTIE 1

Comment l'activité affecte le cycle de vie des Fragments ?

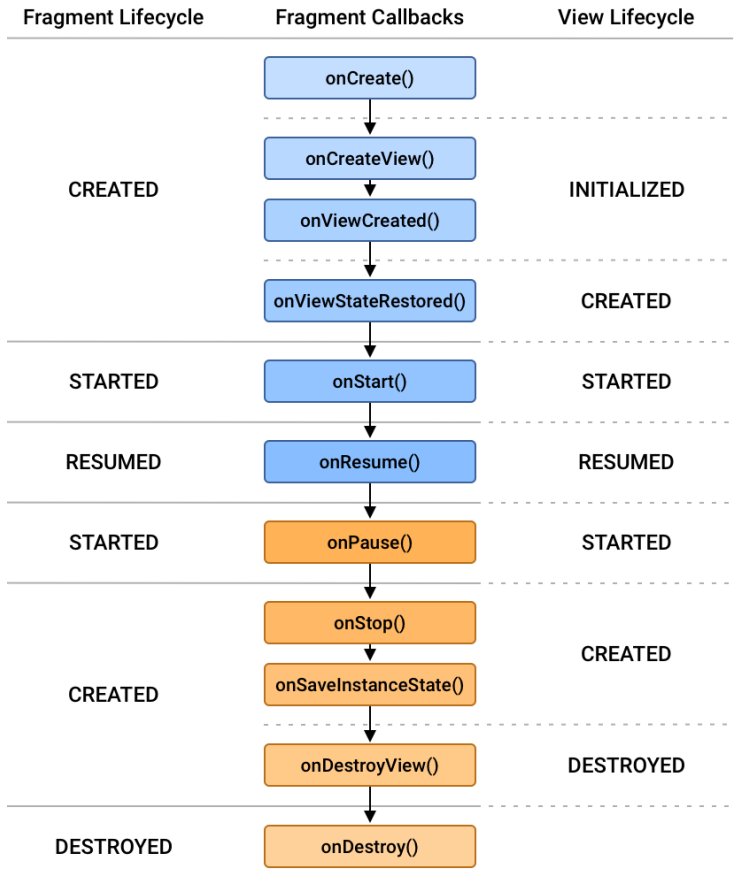
État d'activité	Rappels du Fragment déclenchés	Cycle de vie du Fragment
Created	onAttach() onCreate() onCreateView() onActivityCreated()	Le Fragment est ajouté et sa disposition est inflatée.
Started	onStart()	Le Fragment est actif et visible.
Resumed	onResume()	Le Fragment est actif et prêt pour l'interaction avec l'utilisateur.
Paused	onPause()	Le Fragment est en pause parce que l'activité est en pause.
Stopped	onStop()	Le Fragment est arrêté et n'est plus visible.
Destroyed	onDestroyView() onDestroy() onDetach()	Le Fragment est détruit.

Utilisation des rappels du cycle de vie des Fragments

- **Rappels (callbacks) pour rendre le Fragment actif :**
  - **onCreate()** : Initialise les composants et les variables du Fragment
  - **onCreateView()** : Inflate le layout XML du Fragment
- **Initialiser le Fragment dans onCreate() :**
  - Surcharge onCreate(Bundle savedInstanceState) :
    - Le système appelle **onCreate()** lorsque le Fragment est créé
    - Initialiser les composants et les variables du Fragment (préservés si le Fragment est mis en pause et repris)
    - Inclure toujours **super.onCreate(savedInstanceState)** dans les rappels du cycle de vie
- **Afficher le layout dans onCreateView() :**
  - Surcharge onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) :
    - **Inflate XML layout-requis** si le Fragment a une interface utilisateur
    - Le système appelle cette méthode pour rendre le Fragment visible
    - Doit retourner la vue racine de la disposition du Fragment ou null si le Fragment n'a pas d'interface utilisateur

Utilisation des rappels du cycle de vie des Fragments

Un récapitulatif des états du cycle de vie du Fragment et leur relation avec les rappels du cycle de vie du Fragment et le cycle de vie de la vue du Fragment sont présentés dans la figure suivante :



Réf

Plus de rappels du cycle de vie des Fragments

- **onAttach()** : Appelé lorsque le Fragment est attaché à l'activité pour la première fois ;
- **onPause()** : Appelé lorsque l'activité est en pause
- **onResume()** : Appelé par l'activité pour reprendre un Fragment visible
- **onActivityCreated()** : Appelé lorsque la méthode **onCreate()** de l'activité est retournée
- **onDestroyView()** : Appelé lorsque la vue précédemment créée par **onCreateView**() est détachée du Fragment

Rappel pour l'initialisation finale

- **onActivityCreated()** : Appelé lorsque la méthode **onCreate()** de l'activité est retournée
    - Appeler après **onCreateView()** et avant **onViewStateRestored()**
    - Récupérer les vues ou restaurer l'état
    - Utiliser **setRetainInstance()** pour conserver l'instance du Fragment lorsque l'activité est recréée

- Utilisation de onDestroyView :
    - Utiliser **onDestroyView()** pour effectuer une action après que le Fragment ne soit plus visible :
        - Appelé après **onStop()** et avant **onDestroy()**
        - La vue qui a été créée dans **onCreateView()** est détruite
        - Une nouvelle vue est créée la prochaine fois que le Fragment doit être affiché

Utilisation des méthodes des Fragments et du contexte de l'activité

- **Utiliser le contexte de l'activité :**
  - Lorsque le Fragment est actif ou repris :
    - Utiliser **getActivity()** pour obtenir l'activité qui a démarré le Fragment
    - Trouver une vue dans le layout de l'activité : **View listView = getActivity().findViewById(R.id.list)**.
- **Appeler des méthodes dans le Fragment :**
  - Obtenir le Fragment en appelant **findFragmentById()** sur FragmentManager :

```
ExempleFragment Fragment = (ExempleFragment) getFragmentManager().findFragmentById(R.id.example_Fragment)
  // ...
mData = Fragment.getSomeData()
```

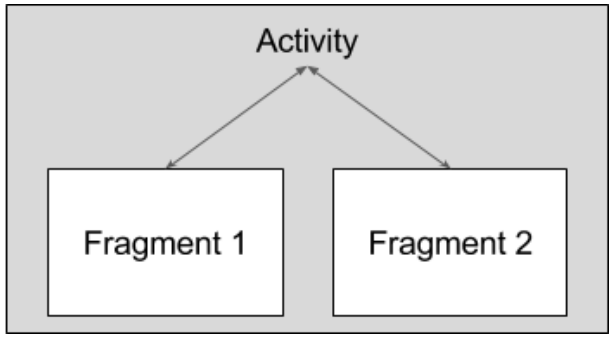
- **Utiliser le back stack :**
  - Ajouter le Fragment à la pile arrière (back stack) :

```
FragmentTransaction.add(R.id.Fragment_container, Fragment)
FragmentTransaction.addToBackStack(null)
FragmentTransaction.commit()
```

  - L'activité hôte maintient la pile arrière même après la suppression du Fragment
  - L'utilisateur peut revenir au Fragment avec le bouton Précédent (Back)

Communication par Fragment

- **Communication des activités et des Fragments :**
  - L'activité peut envoyer des données au Fragment et recevoir des données du Fragment ;
  - Toutes les communications entre Fragments se font par l'intermédiaire de l'activité hôte.



- **Envoyer des données au Fragment :**
  - Dans la méthode **factory** (fabrique) de Fragment **newInstance()** :
    - Créer un Bundle ;
    - Utiliser **setArguments(Bundle)** pour fournir les arguments de construction du Fragment;

```
public static SimpleFragment newInstance(int choice) {
      SimpleFragment Fragment = new SimpleFragment();
      Bundle arguments = new Bundle();
      arguments.putInt(CHOICE, choice);
      Fragment.setArguments(arguments);
      return Fragment;
}
```

    - Inclure des données (telles que mRadioButtonChoice) dans l'appel à Fragment à partir de l'activité :

```
SimpleFragment Fragment =
      SimpleFragment.newInstance(mRadioButtonChoice);
```

PARTIE 1

Communication par Fragment

- **Utiliser les données de l'activité dans le Fragment :**
  - Avant de créer une vue de Fragment, récupérer les arguments du Bundle en utilisant **getArguments()** ;
  - Utiliser **onCreate()** ou **onCreateView()**.

```
if (getArguments().containsKey(CHOICE)) {
    mRadioButtonChoice = getArguments().getInt(CHOICE);
    // ...
}
```

- **Récupérer les données du Fragment :**
  - Dans un Fragment :
    - Définir une interface (telle qu'un écouteur) avec une ou plusieurs méthodes de rappel
    - Surcharger **onAttach()** pour récupérer l'implémentation de l'interface (vérifier si l'activité implémente l'interface)
    - Appeler la méthode de l'interface pour passer des données en tant que paramètre
  - Dans une activité :
    - Toutes les classes d'activité utilisant le Fragment doivent implémenter l'interface
    - Utiliser la ou les méthodes de rappel de l'interface pour récupérer les données

Communication par Fragment

- **Définir l'interface et les méthodes de rappel :**
  - Dans le Fragment : définir une interface (comme un écouteur) avec une méthode de rappel (comme **onRadioButtonChoice()**) :

```
interface OnFragmentInteractionListener {
    void onRadioButtonChoice(int choice);
}
```

- **Récupérer l'implémentation de l'interface de l'activité :**

```
@Override
public void onAttach(Context context) {
    super.onAttach(context);
    if (context instanceof OnFragmentInteractionListener) {
        mListener = (OnFragmentInteractionListener) context;
    } else {
        // ...
```

- **Appeler la méthode de l'interface pour passer les données :**

```
public void onCheckedChanged(RadioGroup group, int checkedId) {
    // ...
    switch (index) {
        case YES: // Utiliser le choix "Yes."
            mListener.onRadioButtonChoice(YES);
            break;
        case NO: // Utiliser le choix "No."
            mListener.onRadioButtonChoice(NO);
            break;
        // ...
    }
```

PARTIE 1

Communication par Fragment

- Implémenter l'interface dans l'activité :
  - L'activité doit implémenter l'interface définie dans le Fragment :

```
public class MainActivity extends AppCompatActivity
                        implements SimpleFragment.OnFragmentInteractionListener {
```

  - Utiliser le rappel dans l'activité :
    - L'activité peut alors utiliser le callback **onRadioButtonChoice()** :

```
@Override
public void onRadioButtonChoice(int choice) {
    mRadioButtonChoice = choice;
    // Utiliser mRadioButtonChoice dans l'activité
    // ...
}
```

CHAPITRE 2

Connaitre le cycle de vie des composants applicatifs

1. Utilisation de l'Activity
2. Utilisation du Fragment
3. **Manipulation des Services**
4. Création du broadcast receiver

Qu'est-ce qu'un service ?

Un service est un composant d'application qui peut effectuer des opérations de longue durée en arrière-plan et ne fournit pas d'interface utilisateur.

À quoi servent les services ?

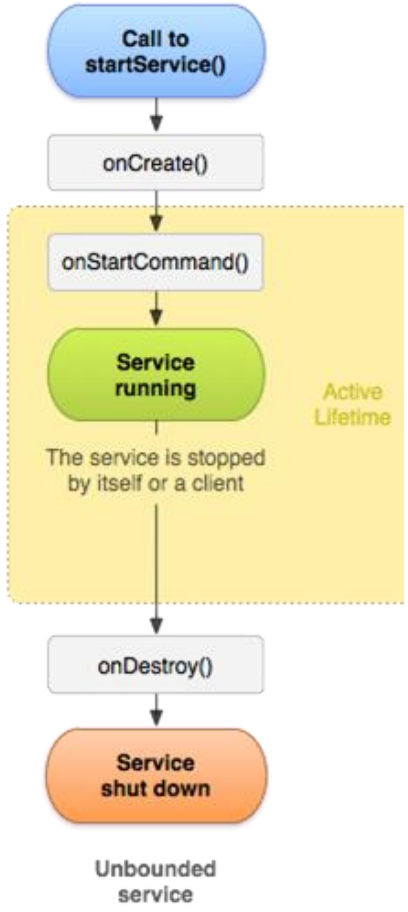
- Effectuer des transactions en réseau
- Jouer de la musique
- Effectuer des entrées / sorties de fichiers
- Interagir avec une base de données

Caractéristiques des services :

- Démarré avec un Intent
- Peut rester en cours d'exécution lorsque l'utilisateur change d'application
- Cycle de vie : à gérer
- D'autres applications peuvent utiliser le service : pour gérer les permissions
- S'exécute dans le fil principal de son processus d'hébergement

Formes de services : started

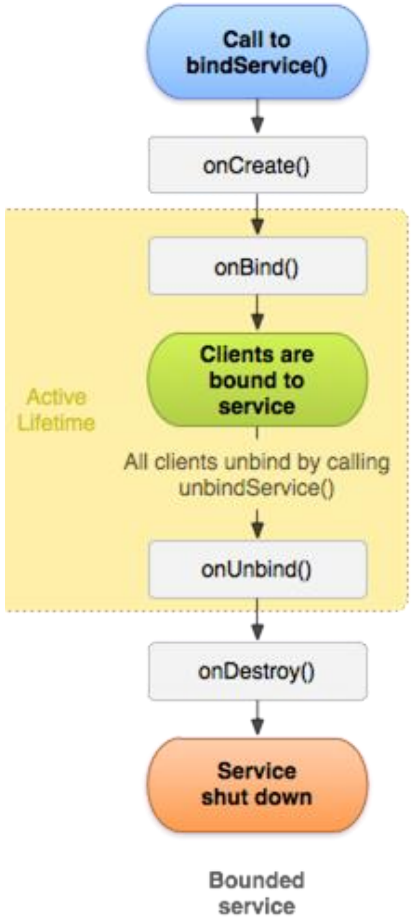
- Démarré avec **startService()**
- Fonctionne jusqu'à ce qu'il s'arrête de lui-même
- En général, il ne met pas à jour l'interface utilisateur



Réf

Formes de services : bound (lié)

- Offre une interface client-serveur qui permet aux composants d'interagir avec le service
- Les clients envoient des requêtes et obtiennent des résultats
- Démarrage avec **bindService()**
- Se termine lorsque tous les clients se sont déliés



Réf

Services et Threads

- Bien que les services soient séparés de l'interface utilisateur, ils s'exécutent toujours sur le Thread principal par défaut (sauf IntentService)
- Décharge le travail intensif du CPU sur un Thread séparé au sein du service

Mise à jour de l'application

- Si le service ne peut pas accéder à l'interface utilisateur, comment mettre à jour l'application pour afficher les résultats ? Utiliser un **broadcast receiver**

Services de premier plan

- Fonctionne en arrière-plan mais nécessite que l'utilisateur soit conscient de son existence (par exemple, un lecteur de musique utilisant un service musical).
    - Il est plus prioritaire que les services d'arrière-plan car l'utilisateur remarquera son absence et il est peu probable que le système le tue
    - Il doit fournir une notification que l'utilisateur ne peut pas ignorer tant que le service est en cours d'exécution

Limites des services d'arrière-plan

- À partir de l'API 26 (Android 8), les applications d'arrière-plan ne sont pas autorisées à créer un service d'arrière-plan
- Une application d'avant-plan peut créer et exécuter des services d'avant-plan et d'arrière-plan
- Lorsqu'une application passe en arrière-plan, le système arrête les services d'arrière-plan de l'application
- La méthode **startService()** lève désormais une **IllegalStateException** si l'application cible l'API 26
- Ces limitations n'affectent pas les services de premier plan ou les services liés

Création d'un service

- `<service android:name=".ExampleService" />`
- Gérer les permissions
- Sous-classer la classe **IntentService** ou **Service**
- Implémenter des méthodes de cycle de vie
- Démarrer le service depuis l'activité
- S'assurer que le service peut être arrêté

Arrêter un service

- Un service démarré doit gérer son propre cycle de vie
- S'il n'est pas arrêté, il continuera à fonctionner et à consommer des ressources
- Le service doit s'arrêter lui-même en appelant **stopSelf()**
- Un autre composant peut l'arrêter en appelant **stopService()**
- Le service lié (bound) est détruit lorsque tous les clients se détachent
- **IntentService** est détruit après le retour de **onHandleIntent()**

IntentService

- Service simple avec un cycle de vie simplifié
- Utilise des fils de travail pour répondre aux demandes
- S'arrête de lui-même lorsqu'il a terminé
- Idéal pour une longue tâche sur un seul Thread d'arrière-plan

Limitations de IntentService

- Ne peut pas interagir avec l'interface utilisateur
- Ne peut exécuter qu'une seule requête à la fois
- Ne peut pas être interrompu

Restrictions de IntentService

- **IntentService** sont soumis aux nouvelles restrictions sur les services d'arrière-plan
- Pour les applications ciblant l'API 26, Android Support Library 26.0.0 introduit un nouveau **JobIntentService**
- **JobIntentService** fournit la même fonctionnalité que **IntentService** mais utilise des jobs au lieu des services

Mise en œuvre de IntentService

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

    @Override
  protected void onHandleIntent(Intent intent) {
      try {
          // Faire un travail
      } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
      }
  } // Lorsque cette méthode retourne, IntentService arrête le service, selon le cas.
}
```

CHAPITRE 2

Connaitre le cycle de vie des composants applicatifs

1. Utilisation de l'Activity
2. Utilisation du Fragment
3. Manipulation des Services
4. **Création du broadcast receiver**

Broadcasts

- Les broadcasts sont des messages envoyés par le système Android et d'autres applications Android, lorsqu'un événement intéressant se produit ;
- Les broadcasts sont enveloppés dans un objet Intent. Cet objet Intent contient les détails de l'événement, comme **android.intent.action.HEADSET_PLUG**, envoyé lorsqu'un casque filaire est branché ou débranché.

Types des broadcasts

- Broadcasts système
- Broadcasts personnalisés

Broadcasts système

Les broadcasts système sont les messages envoyés par le système Android lorsqu'un événement système se produit, qui peut affecter l'application.

Quelques exemples :

- Une intention avec l'action **ACTION_BOOT_COMPLETED** est diffusée lorsque le dispositif démarre
- Une intention avec l'action **ACTION_POWER_CONNECTED** est diffusée lorsque le dispositif est connecté à l'alimentation externe

Broadcasts personnalisés

- Les broadcasts personnalisés sont des broadcasts que l'application envoie, à l'instar du système Android ;
- **Par exemple**, lorsque vous souhaitez faire savoir à une ou plusieurs autres applications que des données ont été téléchargées par l'application et qu'elles sont disponibles pour leur usage.

Envoi d'un broadcast personnalisé

Android propose trois façons d'envoyer un broadcast :

- Broadcast ordonné
- Broadcast normal
- Broadcast local

Broadcast ordonné

- Le broadcast ordonné est envoyé à un seul récepteur à la fois
- Pour envoyer un broadcast ordonnée, utiliser la méthode **sendOrderedBroadcast()**
- Les récepteurs peuvent propager le résultat au récepteur suivant ou même interrompre la diffusion
- Contrôler l'ordre de diffusion avec l'attribut **android:priority** dans le fichier **manifest**
- Les récepteurs ayant la même priorité s'exécutent dans un ordre arbitraire

Broadcast normal

- Délivré à tous les récepteurs enregistrés en même temps, dans un ordre indéfini
- C'est le moyen le plus efficace d'envoyer une diffusion
- Les récepteurs ne peuvent pas propager les résultats entre eux, et ils ne peuvent pas interrompre la diffusion
- La méthode **sendBroadcast()** est utilisée pour envoyer un broadcast normal

Broadcast local

- Envoie des diffusions à des récepteurs dans l'application
- Pas de problème de sécurité car il n'y a pas de communication interprocessus
- Pour envoyer une diffusion locale :
    - Obtenir une instance de **LocalBroadcastManager**
    - Appeler **sendBroadcast()** sur l'instance

```
LocalBroadcastManager.getInstance(this)
                                .sendBroadcast(customBroadcastIntent)
```

Broadcast personnalisés

- L'expéditeur et le destinataire doivent convenir d'un nom unique pour **l'intent** (nom de l'action)
- Définir dans l'activité et **broadcast receiver** :

```
private static final String ACTION_CUSTOM_BROADCAST =
                    "com.example.android.powerreceiver.ACTION_CUSTOM_BROADCAST";
```

Qu'est-ce qu’un broadcast receiver ?

- Les **broadcasts receivers** sont des composants de l'application
- Ils s'inscrivent à divers broadcasts du système ou à des broadcasts personnalisés
- Ils sont notifiés (via un Intent) :
    - Par le système, lorsque se produit un événement système pour lequel l’application est enregistrée
    - Par une autre application, y compris la vôtre, si l’application est enregistrée pour cet événement personnalisé

Enregistrer le broadcast receiver

- Les **broadcast receivers** peuvent être enregistrés de deux manières :
    - Receivers statiques :
        - Enregistrés dans AndroidManifest.xml, également appelés receivers déclarés dans le manifeste (Manifest-declared receivers)
    - Receivers dynamiques :
        - Enregistrés en utilisant le contexte de l'application ou des activités dans les fichiers Java, également appelés receivers enregistrés dans le contexte (Context-registered receivers)

Réception d'un broadcast système

- À partir d'Android 8.0 (niveau 26 de l'API), les **receivers statiques** ne peuvent pas recevoir la plupart des diffusions du système
- Utiliser un **receiver dynamique** pour enregistrer ces broadcasts
- Si les **broadcasts système** sont enregistrés dans le manifeste, le système Android ne les transmettra pas à l'application
- Quelques **broadcasts** sont exemptés de cette restriction. Consulter la liste complète des exceptions de **broadcast implicite**

Implémentation du broadcast receiver

- **Pour créer un broadcast receiver :**
  - Etendre de classe **BroadcastReceiver** et redéfinir la méthode **onReceive()** ;
  - Enregistrer le broadcast receiver et spécifier les filtres d'intent (**Intent-filters**) :
    - Statiquement, dans le Manifeste ;
    - Dynamiquement, avec **registerReceiver().**
- **Qu'est-ce que Intent-filters ?**
  - **Intent-filters** spécifient les types d'intents qu'un broadcast receiver peut recevoir. Ils filtrent les intents entrants en fonction des valeurs d'intent comme l'action.
  - Pour ajouter un **Intent-filters** :
    - Dans le fichier **AndroidManifest.xml,** utiliser la balise **<intent-filter>.**
    - Dans le fichier Java, utiliser l'objet **IntentFilter**.

Sous-classe de broadcast receiver

- **Dans Android studio, File > New > Other > BroadcastReceiver :**

```
public class CustomReceiver extends BroadcastReceiver {
   @Override
   public void onReceive(Context context, Intent intent) {
       // Cette méthode est appelée lorsque le BroadcastReceiver
       // reçoit une diffusion d'un Intent.
       throw new UnsupportedOperationException(" Pas encore implémenté");
   }
}
```

- **Implémenter onReceive() :**
  - Exemple d'implémentation de la méthode **onReceive()** qui tient compte de l'alimentation branchée et débranchée.

```
@Override
public void onReceive(Context context, Intent intent) {
   String intentAction = intent.getAction();
   switch (intentAction){
       case Intent.ACTION_POWER_CONNECTED:
           break;
       case Intent.ACTION_POWER_DISCONNECTED:
           break;
   }
}
```

Enregistrement statique dans le manifeste Android

- L'élément **<receiver>** à l'intérieur de la balise **<application>**
- **<intent-filter>** enregistre le receiver pour des intents spécifiques

```
<receiver
   android:name=".CustomReceiver"
   android:enabled="true"
   android:exported="true">
   <intent-filter>
      <action android:name="android.intent.action.BOOT_COMPLETED"/>
   </intent-filter>
</receiver>
```

Enregistrement dynamique

- Enregistrer le receiver dans **onCreate()** ou **onResume()** :
    - // Enregistrer le receiver en utilisant le contexte de l'activité
    - this.registerReceiver(mReceiver, filter)
- Désenregistrement dans **onDestroy()** ou **onPause()** :
    - // Désenregistrement du receiver ;
    - this.unregisterReceiver(mReceiver) ;

Enregistrement d'un broadcast receiver local

- Enregistrer les **receivers locaux** de façon dynamique, car l'enregistrement statique dans le manifeste n'est pas possible pour une diffusion locale
- Pour enregistrer un **receiver** pour les diffusions locales :
  - Obtenir une instance de **LocalBroadcastManager**
  - Appeler **registerReceiver()**

```
LocalBroadcastManager.getInstance(this).registerReceiver
                    (mReceiver, new IntentFilter(CustomReceiver.ACTION_CUSTOM_BROADCAST));
```

Désenregistrer un broadcast receiver local

- Pour désenregistrer un broadcast receiver local :
    - Récupérer une instance de **LocalBroadcastManager**.
    - Appeler **LocalBroadcastManager.unregisterReceiver()**.

```
LocalBroadcastManager.getInstance(this) .unregisterReceiver(mReceiver) ;
```

Restriction des broadcasts

- Il est fortement recommandé de restreindre la diffusion (broadcast)
- Une diffusion (broadcast) non restreinte peut constituer une menace pour la sécurité
- Par exemple : si la diffusion de vos apps n'est pas restreinte et qu'elle comprend des informations sensibles, une app contenant un logiciel malveillant pourrait s'enregistrer et recevoir vos données

Façons de restreindre une diffusion

- Si possible, utiliser un **LocalBroadcastManager**, qui conserve les données à l'intérieur de l'application, évitant ainsi les fuites de sécurité
- Utiliser la méthode **setPackage()** et indiquer le nom du paquet. La diffusion est limitée aux applications qui correspondent au nom de paquet spécifié
- Les autorisations d'accès peuvent être appliquées par l'expéditeur ou le récepteur

Appliquer les permissions par expéditeur

- Pour appliquer une permission lors de l'envoi d'un broadcast :
    - Fournir un argument de permission non nul à **sendBroadcast()**
    - Seuls les récepteurs qui demandent cette permission à l'aide de la balise **<uses-permission>** dans leur fichier AndroidManifest.xml peuvent recevoir le broadcast

Appliquer les permissions par récepteur

- Pour appliquer une permission lors de la réception d'une diffusion :
    - Pour le cas d'un **receiver** de manière dynamique, fournir une permission non nulle à **registerReceiver()**
    - Pour le cas d'un **receiver** de manière statique, utiliser l'attribut **android:permission** dans la balise **<receiver>** du fichier **AndroidManifest.xml**

Bonnes pratiques

- Assurer que l'espace de nom pour l'intent est unique et que vous en êtes le propriétaire
- Limiter les broadcasts receivers
- D'autres applications peuvent répondre à la diffusion que l'application envoie : utiliser les autorisations pour contrôler cela
- Préférer les receivers dynamiques aux receivers statiques
- N'exécuter jamais une opération de longue durée dans le broadcast receiver

CHAPITRE 3

Maîtriser la communication entre les composants applicatifs

Ce que vous allez apprendre dans ce chapitre :

- Utilisation de l’Intent
- Utilisation des Bundles
- Création d’un ViewModel

2 heures

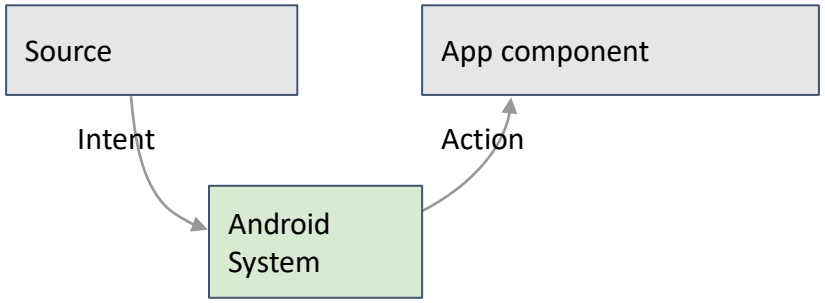
CHAPITRE 3

Maîtriser la communication entre les composants applicatifs

1. **Utilisation de l’Intent**
2. Utilisation des Bundles
3. Création d’un ViewModel

Qu'est-ce qu'un Intent ?

- Un Intent est une description d'une opération à effectuer
- Un Intent est un objet utilisé pour demander une action à un autre composant de l'application via le système Android



Que peuvent faire les Intents ?

- **Démarrer une activité :**
  - Un clic sur un bouton lance une nouvelle activité pour la saisie de texte ;
  - Un clic sur le bouton partager, ouvre une application qui permet de poster une photo
- **Lancer un service :**
  - Lancer le téléchargement d'un fichier en arrière-plan
- **Diffuser un message :**
  - Le système informe tout le monde que le téléphone est en train de se charger

Intents explicites et implicites

- **Intent explicite**
  - Lance une activité spécifique :
    - Demande de thé au lait livré par Nikita
    - L'activité principale lance l'activité ViewShoppingCart
- **Intent implicite**
  - Demande au système de trouver une activité qui peut traiter cette demande :
    - Trouver un magasin ouvert qui vend du thé vert
    - Cliquer sur partager ouvre un sélecteur avec une liste d'applications

Démarrer une activité avec un Intent explicite

- Pour lancer une activité spécifique, utiliser un Intent explicite :

1. Créer un Intent :
    - Intent intent = new Intent(this, ActivityName.class) ;
2. Utiliser l'Intent pour lancer l'activité :
    - startActivity(intent) ;

Démarrer une activité avec un Intent implicite

- Pour demander à Android de trouver une activité pour traiter une demande, utiliser un Intent implicite.

1. Créer un Intent :
    - Intent intent = new Intent(action, uri) ;
2. Utiliser l'Intent pour démarrer l'activité :
    - startActivity(intent) ;

Exemples : Implicit Intents

- **Afficher une page web** :

```
Uri uri = Uri.parse("http://www.google.com");
Intent it = new Intent(Intent.ACTION_VIEW,uri);
startActivity(it);
```

- **Composer un numéro de téléphone :**

```
Uri uri = Uri.parse("tel:0587871234");
Intent it = new Intent(Intent.ACTION_DIAL, uri);
startActivity(it);
```

- **Éviter les exceptions et les pannes :**

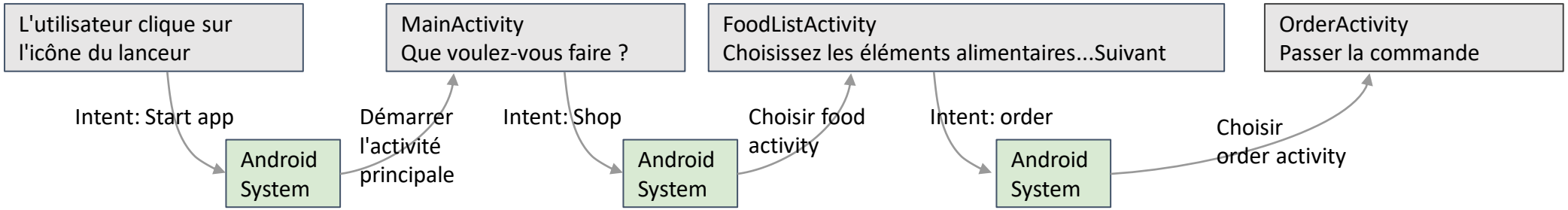
Avant de lancer une activité implicite, utiliser le gestionnaire de paquets pour vérifier qu'il existe un paquet avec une activité qui correspond aux critères donnés.

```
Intent myIntent = new Intent(Intent.ACTION_CALL_BUTTON);
if (intent.resolveActivity(getPackageManager()) != null) {
    startActivity(intent);
}
```

PARTIE 1

Comment s'exécute les activités ?

- Toutes les instances d'activité sont gérées par le runtime Android
- Démarrage par un "Intent", un message adressé au moteur d'exécution Android pour lancer une activité



Envoyer et recevoir les données

- **Deux types d'envoi de données avec des Intents :**
  - Data - un élément d'information dont l'emplacement des données peut être représenté par un URI ;
  - Extras : un ou plusieurs éléments d'information sous la forme d'une collection de paires clé-valeur dans un Bundle.

- **Envoi et récupération de données :**
  - Dans la première activité (d'envoi) :
    1. Créer l'objet Intent ;
    2. Placer des données ou des éléments supplémentaires dans l'objet Intent ;
    3. Démarrer la nouvelle activité avec **startActivity().**
  - Dans la deuxième activité (réceptrice) :
    1. Récupérer l'objet Intent avec lequel l'activité a été lancée ;
    2. Récupérer les données ou les extras de l'objet Intent.

- **Placer un URI comme donnée d'Intent :**

```
// L'URL d'une page web
intent.setData(
    Uri.parse("http://www.google.com"));

// un URI du fichier échantillon

intent.setData(
     Uri.fromFile(new File("/sdcard/sample.jpg")));
```

- **Mettre l'information en extras d'Intent :**
  - `putExtra(String name, int value)`
    ⇒ `intent.putExtra("level", 406);`
  - `putExtra(String name, String[] value)`
    ⇒ `String[] foodList = {"Rice", "Beans", "Fruit"};`
    `intent.putExtra("food", foodList);`
  - `putExtras(bundle);`

    ⇒ Si beaucoup de données, créer d'abord un paquet et passer le paquet.

Envoyer et recevoir les données

- **Envoi de données à une activité avec des extras :**

```
public static final String EXTRA_MESSAGE_KEY =
    "com.example.android.twoactivities.extra.MESSAGE";

Intent intent = new Intent(this, SecondActivity.class);
String message = "Hello Activity!";
intent.putExtra(EXTRA_MESSAGE_KEY, message);
startActivity(intent);
```

- **Récupérer des données à partir d'Intents :**
  - `getData();`
    ⇒ `Uri locationUri = intent.getData();`
  - `int getIntExtra (String name, int defaultValue)`
    ⇒ `int level = intent.getIntExtra("level", 0);`
  - `Bundle bundle = intent.getExtras();`
    ⇒ Obtenir toutes les données en une seule fois sous forme bundle.

- **Renvoyer les données à l'activité de départ :**

1. Utiliser **startActivityForResult()** pour lancer la deuxième activité.
2. Pour renvoyer les données de la deuxième activité :
   - Créer un nouvel Intent
   - Placer les données de la réponse dans l'Intent en utilisant **putExtra()**
   - Donner au résultat la valeur **Activity.RESULT_OK** ou **RESULT_CANCELED**, si l'utilisateur a annulé l'activité
   - Appeler finish() pour fermer l'activité
3. Implémenter **onActivityResult()** dans la première Activity.

- **startActivityForResult() :**

  startActivityForResult(intent, requestCode)
  - Démarre l'activité (intent), lui attribue un identifiant (requestCode)
  - Renvoie les données via les extras de l'intention
  - Lorsque c'est fait, on retire la pile, on retourne à l'activité précédente et on exécute le callback **onActivityResult()** pour traiter les données renvoyées
  - Utiliser le requestCode pour identifier l'activité qui a "retourné" les données

PARTIE 1

Exemple

1. **startActivityForResult() :**

```
public static final int CHOOSE_FOOD_REQUEST = 1;

Intent intent = new Intent(this, ChooseFoodItemsActivity.class);
startActivityForResult(intent, CHOOSE_FOOD_REQUEST);
```

2. **Retourner les données et terminer la deuxième activité :**

```
 // Créer un intent
Intent replyIntent = new Intent();

// Mettre les données à retourner dans l'extra

replyIntent.putExtra(EXTRA_REPLY, reply);

// Donner la valeur RESULT_OK au résultat de l'activité
setResult(RESULT_OK, replyIntent);

// Terminer l'activité en cours

finish();
```

3. **Implémenter onActivityResult()**

```
public void onActivityResult(int requestCode,
                             int resultCode, Intent data) {
  super.onActivityResult(requestCode, resultCode, data);
  if (requestCode == TEXT_REQUEST) { // Identifier l'activité
    if (resultCode == RESULT_OK) { // Activité réussie
String reply =
data.getStringExtra(SecondActivity.EXTRA_REPLY);
      // … faire quelque chose avec les données
 }}}
```

Actions communes pour un Intent implicite

Les actions courantes incluent :

- ACTION_SET_ALARM
- ACTION_CAPTURE_D'IMAGE
- ACTION_CRÉER_DOCUMENT
- ACTION_SENDTO

Des applications qui gèrent les actions courantes

Les actions courantes sont généralement gérées par les applications installées (applications système et autres applications), telles que :

- Alarme, Calendrier, Appareil photo, Contacts
- Courriel, Stockage des fichiers, Cartes, Musique/Vidéo
- Notes, Téléphone, Recherche, Paramètres
- Messagerie texte et navigation sur le Web

→ *Liste des actions communes pour intent implicite*

→ *Liste de toutes les actions disponibles*

Réception d'un Intent implicite

- **Enregistrer l'application pour recevoir un message Intent :**
  - Déclarer un ou plusieurs filtres d'Intent pour l'activité dans AndroidManifest.xml ;
  - Le filtre annonce la capacité de l'activité à accepter un Intent implicite ;
  - Le filtre pose des conditions sur l'Intent que l'activité accepte.

- **Filtre Intent dans AndroidManifest.xml :**

```
<activity android:name="ShareActivity">
  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    <category android:name="android.intent.category.DEFAULT"/>
    <data android:mimeType="text/plain"/>
  </intent-filter>
</activity>
```

- **Filtres d'Intent : action et catégorie**
  - **Action** - Correspond à une ou plusieurs constantes d'action :
    - **android.intent.action.VIEW** : correspond à n'importe quel Intent avec ACTION_VIEW ;
    - **android.intent.action.SEND** : correspond à n'importe quelle Intent avec ACTION_SEND.
  - **Catégorie** - Informations supplémentaires (liste de catégories) :
    - **android.intent.category.BROWSABLE :** peut être lancé par un navigateur Web ;
    - **android.intent.category.LAUNCHER :** affiche l'activité sous forme d'icône de lanceur.
- **Filtres d'Intent : données**
  - Data - Filtre sur les URI de données, type MIME
    - **android:scheme="https" :** exige que les URIs soient du protocole https
    - **android:host="developer.android.com"** : accepte uniquement les Intents provenant des hôtes spécifiés
    - **android:mimeType="text/plain"** : limite les types de documents acceptables

Une activité peut avoir plusieurs filtres

```
<activity android:name="ShareActivity">

  <intent-filter>
    <action android:name="android.intent.action.SEND"/>
    ...
  </intent-filter>

  <intent-filter>
    <action android:name="android.intent.action.SEND_MULTIPLE"/>
    ...
  </intent-filter>

</activity>
```

Un filtre peut avoir plusieurs actions et données

```
<intent-filter>
  <action android:name="android.intent.action.SEND"/>
  <action android:name="android.intent.action.SEND_MULTIPLE"/>
  <category android:name="android.intent.category.DEFAULT"/>
  <data android:mimeType="image/*"/>
  <data android:mimeType="video/*"/>
</intent-filter>
```

CHAPITRE 3

Maîtriser la communication entre les composants applicatifs

1. Utilisation de l’Intent
2. **Utilisation des Bundles**
3. Création d’un ViewModel

Classe Bundle

- Utilisée pour passer des données entre activités ou instances d'une même activité :
  - Éléments de type clé/valeur (clé étant de type String)
  - Valeurs peuvent être :
    - Types « simples » : entiers, caractères…
    - Chaînes de caractères
    - Tableaux et listes

Qu'est-ce que le Bundle savedInstanceState ?

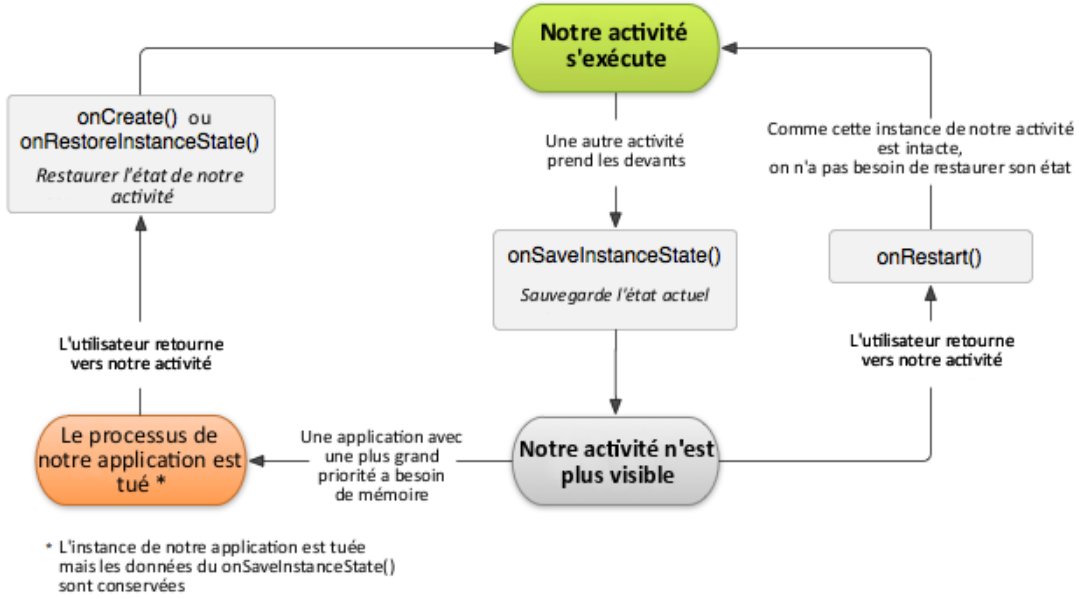
Le **savedInstanceState** est une référence à un objet Bundle qui est passé dans la méthode onCreate de chaque activité Android. Les activités ont la possibilité, dans des circonstances particulières, de se restaurer à un état antérieur en utilisant les données stockées dans ce bundle. S'il n'y a pas de données d'instance disponibles, le **savedInstanceState** sera nul. Par exemple, le **savedInstanceState** sera toujours nul la première fois qu'une activité est lancée, mais peut être non nul si une activité est détruite pendant la rotation.

Persistance de l'état d'une activité

- Ajout à l'activité la méthode :
  - **public void onSaveInstanceState(Bundle outState)**
  - Ajouter dans **outState** les données à sauvegarder
- Récupération des données dans la méthode :
  - **public void onCreate (Bundle saveInstanceState)**
  - **public void onRestoreInstanceState(Bundle saveInstanceState)**
- Quelques méthodes de la **classe Bundle** :
  - put/get (Int, Double, String, StringArray, CharArray ...)

Exemple :

```
@Override
public void onSaveInstanceState(Bundle outState) {
  outState.putString("message", " C'est mon message à recharger ");
  super.onSaveInstanceState(outState);
}
```



Réf

Utilisation des Bundles

Exemple 1 : Transférer les données entres les activités

Dans MainActivity :

```
// création d'un Intent
Intent intent = new Intent(this, SecondActivity.class);
// création d'un objet Bundle
Bundle bundle = new Bundle();

// stockage de la valeur de la chaîne dans le Bundle
// qui est mappé à la clé
bundle.putString("key1", "Main :- Main Activity");

// passer le Bundle à l'Intent
intent.putExtras(bundle);
// démarrer l'Intent
startActivity(intent);
```

Récupération des données dans SecondActivity :

```
// récupération de Bundle
Bundle bundle = getIntent().getExtras();
// récupérer la chaine
String title = bundle.getString("key1", "Default");
```

Exemple 2 : Transférer les données de l'activité au Fragment à l'aide du Bundle

Tous les Fragments doivent avoir un constructeur vide (c'est-à-dire une méthode de construction sans argument d'entrée). Par conséquent, afin de transmettre les données au Fragment en cours de création, il convient d'utiliser la méthode **setArguments().** Cette méthode obtient un Bundle, dans lequel sont stockées les données, et stocke le Bundle dans les arguments. Par la suite, ce Bundle peut être récupéré dans les appels **onCreate()** et **onCreateView()** du Fragment.

Dans l'activité :

```
Bundle bundle = new Bundle();
String myMessage = "Dev mobile, c'est cool!";
bundle.putString("message", myMessage );
FragmentClass fragInfo = new FragmentClass();
fragInfo.setArguments(bundle);
transaction.replace(R.id.Fragment_single, fragInfo);
transaction.commit();
```

Dans le Fragment :

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState) {
   String myValue = this.getArguments().getString("message");
   ...
}
```

PARTIE 1

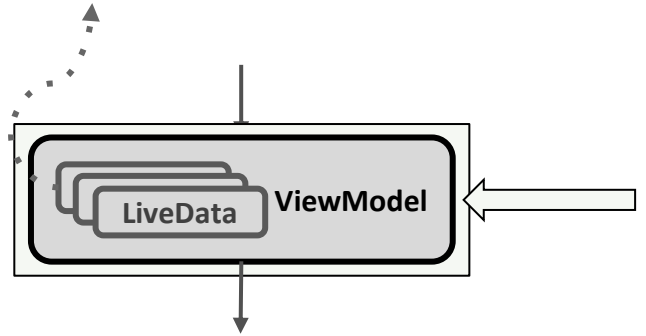
CHAPITRE 3

Maîtriser la communication entre les composants applicatifs

1. Utilisation de l’Intent
2. Utilisation des Bundles
3. **Création d’un ViewModel**

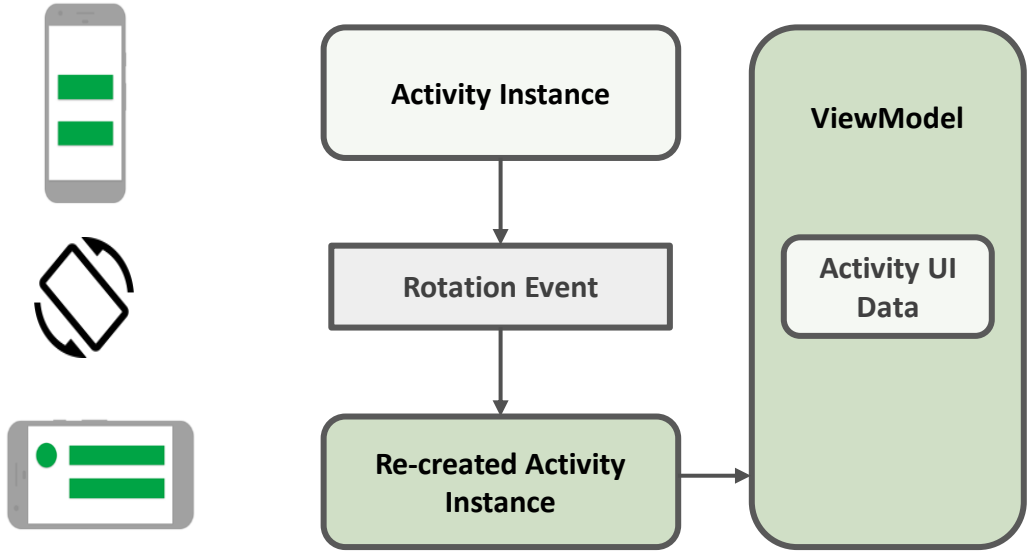
Qu'est ce qu'un ViewModel ?

Les modèles de vue sont des objets qui fournissent des données aux composants de l'interface utilisateur et qui survivent aux changements de configuration.



Un ViewModel :

- Fournit des données à l'interface utilisateur
- Survit aux changements de configuration
- Est utilisé pour partager des données entre les Fragments
- Fait partie de la bibliothèque du cycle de vie



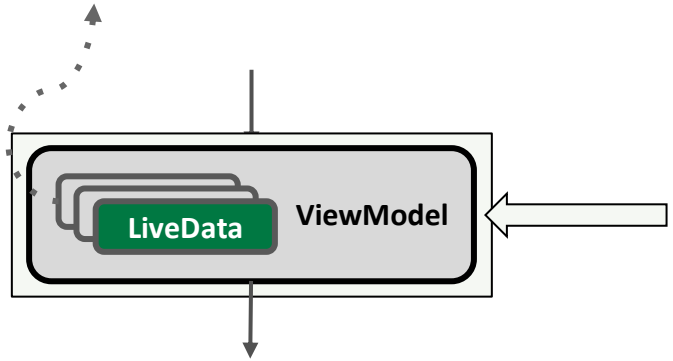
Survit aux changements de configuration

Réf

PARTIE 1

LiveData

- **LiveData** est une classe de support de données qui est consciente des événements du cycle de vie. Elle conserve une valeur et permet d'observer cette valeur
- Utiliser **LiveData** pour maintenir à jour l'interface utilisateur avec la dernière mise à jour des données
- **LiveData** est une donnée **observable**
- Notifie l'observateur lorsque les données changent
- Est conscient du cycle de vie : sait quand l'appareil tourne ou quand l'application s'arrête



Utiliser LiveData pour maintenir l'interface utilisateur à jour :

- Créer un observateur qui observe les **LiveData**
- **LiveData** notifie les objets observateurs lorsque les données observées changent
- L'observateur peut mettre à jour l'interface utilisateur chaque fois que les données changent

Réf

PARTIE 1

Cycle de vie d'un ViewModel

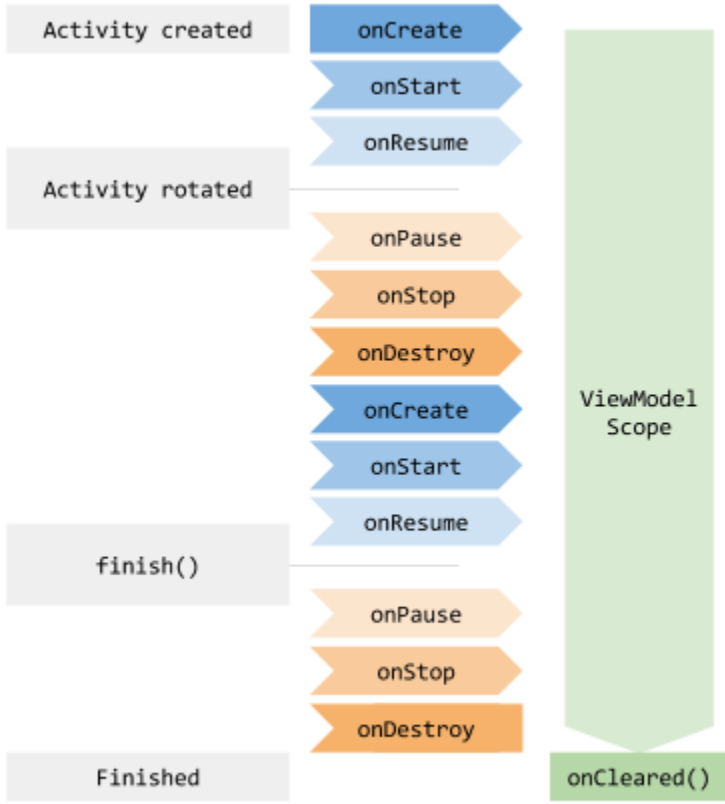
- Le framework maintient le **ViewModel** en vie tant que la portée de l'activité ou du Fragment est en vie. Un **ViewModel** n'est pas détruit si son propriétaire est détruit pour un changement de configuration, tel que la rotation de l'écran. La nouvelle instance du propriétaire se reconnecte à l'instance existante du **ViewModel**, comme l'illustre le schéma suivant.

- Lorsque l'activité est créée, le système appelle la fonction **onCreate(),** puis **onStart(),** puis **onResume(),** mais lorsque nous faisons pivoter l'écran, notre activité est détruite et après la rotation, le système appelle à nouveau la fonction **onCreate()** et d'autres fonctions l'une après l'autre. Comme notre activité est détruite, les données de notre activité ont également disparu.

- Pour surmonter ce problème, nous utilisons des ViewModels qui conservent les données même après des changements de configuration comme la rotation de l'écran.

- L'image montre la portée du **ViewModel**, même avec des changements de configuration, les données sont persistantes. Généralement, un **ViewModel** est appelé pour la première fois lorsque le système appelle la méthode **onCreate()** d'un objet d'activité.

- Le système peut appeler **onCreate()** plusieurs fois au cours de la vie d'une activité, par exemple lorsqu'on fait pivoter l'écran d'un appareil. Le **ViewModel** existe à partir du moment où il est appelé pour la première fois jusqu'à ce que l'activité soit terminée et détruite.



Réf

PARTIE 1

ViewModel sert les données

- **ViewModel** sert les données à l'interface utilisateur
- Les données peuvent provenir de la base de données de Room ou d'autres sources
- Le rôle du **ViewModel** est de renvoyer les données, il peut faire appel à de l'aide pour recevoir ou générer les données

Bonne pratique pour utiliser les repositories :

- Bonne pratique recommandée :
  - Utiliser un repository pour effectuer le travail d'obtention des données
  - Le **ViewModel** reste une interface propre entre l'application et les données
- Le repository sera abordé dans la compétence 14

Réf

Réf

Implémenter un ViewModel

- Architecture Components fournit une classe d'aide **ViewModel** pour le contrôleur de l'interface utilisateur qui est responsable de la préparation des données pour l'interface utilisateur. Les objets **ViewModel** sont automatiquement conservés lors des changements de configuration afin que les données qu'ils contiennent soient immédiatement disponibles pour l'activité ou l'instance de Fragment suivante. Par exemple, pour afficher une liste d'utilisateurs dans l'application, assurer d'attribuer la responsabilité d'acquérir et de conserver la liste d'utilisateurs à un **ViewModel**, plutôt qu'à une activité ou un Fragment, comme l'illustre l'exemple de code suivant :

```
public class MyViewModel extends ViewModel {
    private MutableLiveData<List<User>> users;
    public LiveData<List<User>> getUsers() {
        if (users == null) {
            users = new MutableLiveData<List<User>>();
            loadUsers();
        }
        return users;
    }

    private void loadUsers() {
        // Effectuer une opération asynchrone pour récupérer les utilisateurs.
    }
}
```

Implémenter un ViewModel

- Ensuite accéder à la liste à partir d'une activité comme suit :

```
public class MyActivity extends AppCompatActivity {
    public void onCreate(Bundle savedInstanceState) {
        // Créer un ViewModel la première fois que le système appelle la méthode onCreate() d'une activité.
        // Les activités recréées reçoivent la même instance de MyViewModel créée par la première activité.
        MyViewModel model = new ViewModelProvider(this).get(MyViewModel.class);
        model.getUsers().observe(this, users -> {
            // Mise à jour de l'interface utilisateur
        });
    }
}
```

- Si l'activité est recréée, elle reçoit la même instance de MyViewModel qui a été créée par la première activité. Lorsque l'activité propriétaire est terminée, le framework appelle la méthode **onCleared()** des objets **ViewModel** afin de pouvoir nettoyer les ressources.

Partager des données entre Fragments

Il est très fréquent que deux Fragments ou plus d'une activité aient besoin de communiquer entre eux. Imaginez un cas courant de Fragments à vue partagée (liste-détail), où un Fragment dans lequel l'utilisateur sélectionne un élément dans une liste et un autre Fragment qui affiche le contenu de l'élément sélectionné. Ce cas n'est jamais trivial car les deux Fragments doivent définir une certaine description de l'interface, et l'activité propriétaire doit lier les deux ensemble. En outre, les deux Fragments doivent gérer le scénario où l'autre Fragment n'est pas encore créé ou visible.

Ce problème commun peut être résolu en utilisant des objets **ViewModel**. Ces Fragments peuvent partager un **ViewModel** en utilisant la portée de leur activité pour gérer cette communication, comme l'illustre l'exemple de code suivant :

```
public class SharedViewModel extends ViewModel {
    private final MutableLiveData<Item> selected = new MutableLiveData<Item>();

    public void select(Item item) {
        selected.setValue(item);
    }

    public LiveData<Item> getSelected() {
        return selected;
    }
}
```

```
public class ListFragment extends Fragment {
    private SharedViewModel model;

    public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        model = new ViewModelProvider(requireActivity()).get(SharedViewModel.class);
        itemSelector.setOnClickListener(item -> {
            model.select(item);
        });
    }
}

public class DetailFragment extends Fragment {

    public void onViewCreated(@NonNull View view, Bundle savedInstanceState) {
        super.onViewCreated(view, savedInstanceState);
        SharedViewModel model = new ViewModelProvider(requireActivity()).
                                                        get(SharedViewModel.class);
        model.getSelected().observe(getViewLifecycleOwner(), item -> {
            // Update the UI.
        });
    }
}
```

Partager des données entre Fragments

- On remarque que les deux Fragments récupèrent l'activité qui les contient. De cette façon, lorsque les Fragments obtiennent chacun le **ViewModelProvider**, ils reçoivent la même instance de **SharedViewModel**, dont la portée est limitée à cette activité
- Cette approche offre les avantages suivants :
    - L'activité n'a pas besoin de faire quoi que ce soit, ou de savoir quoi que ce soit sur cette communication.
    - Les Fragments n'ont pas besoin de se connaître les uns les autres, à part le contrat **SharedViewModel**. Si l'un des Fragments disparaît, l'autre continue à fonctionner comme d'habitude.
    - Chaque Fragment a son propre cycle de vie, et n'est pas affecté par le cycle de vie de l'autre. Si un Fragment remplace l'autre, l'interface utilisateur continue à fonctionner sans problème.

PARTIE 2

Créer des tâches asynchrones et tâches de fond

Dans ce module, vous allez :

- Connaissance juste des différentes solutions de gestions des tâches asynchrones
- Implémentation facile des tâches asynchrones

6 heures

CHAPITRE 1

Utiliser le Work Manager

Ce que vous allez apprendre dans ce chapitre :

- Utilisation du Workmanager
- Manipulation des Threads

3 heures

CHAPITRE 1

Utiliser le Work Manager

Définition

WorkManager est la solution recommandée pour les travaux persistants. Un travail est persistant lorsqu'il reste programmé malgré les redémarrages de l'application et les redémarrages du système. Étant donné que la plupart des traitements en arrière-plan sont mieux réalisés grâce à des travaux persistants, **WorkManager** est la principale API recommandée pour les traitements en arrière-plan.

Par exemple :

- L'envoi de journaux ou d'analyses aux services backend
- Synchronisation périodique des données de l'application avec un serveur

Types de travaux persistants

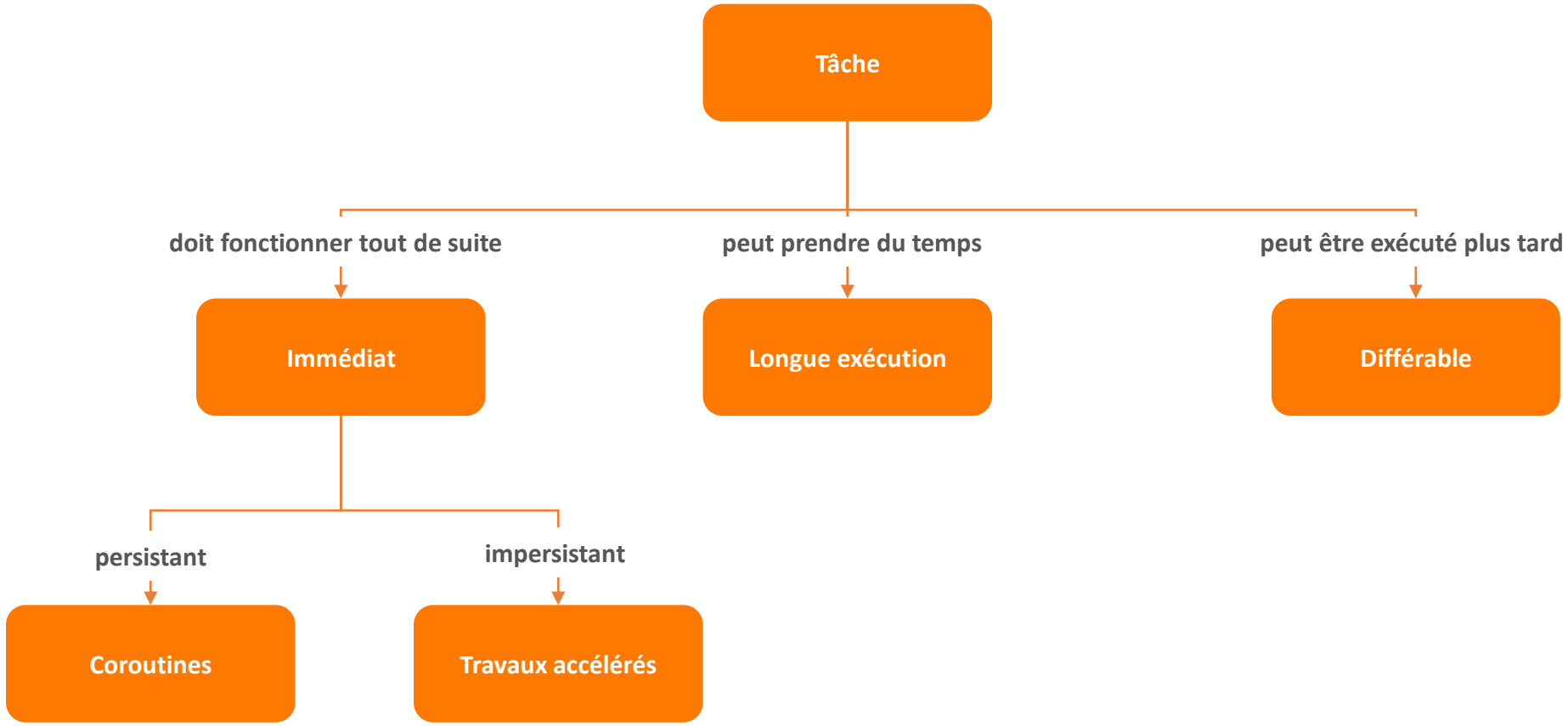
WorkManager gère trois types de travaux persistants :

- **Immédiat** : tâches qui doivent commencer immédiatement et se terminer bientôt. Elles peuvent être accélérées
- **Longue exécution** : tâches qui peuvent durer plus longtemps, potentiellement plus de 10 minutes
- **Différable** : tâches programmées qui commencent à un moment ultérieur et peuvent s'exécuter périodiquement

De même, le travail de fond dans chacune de ces trois catégories peut être soit persistant, soit impersistant :

- **Travail persistant** : il reste programmé même si l'application est redémarrée ou si l'appareil est redémarré
- **Travail impersistant** : il n'est plus programmé après la fin du processus

Types des travaux en arrière plan



Approches du travail en arrière-plan

Aborder différemment le travail persistant et impersistant :

- **Tout travail persistant** : utiliser **WorkManager** pour toutes les formes de travail persistant.
- **Travail impersistant immédiat** : utiliser les coroutines Kotlin pour les tâches persistantes immédiates. Pour les utilisateurs du langage de programmation Java, lisez le guide sur le threading pour connaître les options recommandées.
- **Travail persistant à long terme et différé** : ne pas utiliser les tâches impersistantes à longue durée d'exécution et pouvant être reportées. Plutôt effectuer de telles tâches par le biais de travaux persistants en utilisant **WorkManager**.

Le tableau suivant indique l'approche à adopter pour chaque type de travail en arrière-plan.

Type	Périodicité	Comment accéder	Persistent	Impersistent
Immédiat	Une seule fois	OneTimeWorkRequest et Worker. Pour un travail accéléré, appeler setExpedited() sur OneTimeWorkRequest.	WorkManager	Coroutines
Longue exécution	Une fois ou périodique	Tout WorkRequest ou Worker. Appeler setForeground() dans le Worker pour gérer la notification.	WorkManager	Non recommandé. Au lieu de cela, effectuer le travail de manière persistante en utilisant WorkManager.
Différable	Une fois ou périodique	PeriodicWorkRequest et Worker.	WorkManager	Non recommandé. Au lieu de cela, effectuer le travail de manière persistante en utilisant WorkManager.

Démarrer avec WorkManager

Pour commencer à utiliser **WorkManager**, importer d'abord la bibliothèque dans le projet Android.

Ajouter les dépendances suivantes au fichier **build.gradle** de l'application :

```
dependencies {
    def work_version = "2.7.1"

    // (Java only)
    implementation "androidx.work:work-runtime:$work_version"

    // Kotlin + coroutines
    implementation "androidx.work:work-runtime-ktx:$work_version"

    // optional - RxJava2 support
    implementation "androidx.work:work-rxjava2:$work_version"

    // optional - GCMNetworkManager support
    implementation "androidx.work:work-gcm:$work_version"

    // optional - Test helpers
    androidTestImplementation "androidx.work:work-testing:$work_version"

    // optional - Multiprocess support
    implementation "androidx.work:work-multiprocess:$work_version"
}
```

Une fois que les dépendances sont ajoutées, synchroniser le Gradle de projet, l'étape suivante consiste à définir un **work** à exécuter.

Définir le work

Le **work** est défini à l'aide de la classe **Worker**. La méthode **doWork()** s'exécute de manière asynchrone sur un Thread d'arrière-plan fourni par **WorkManager**.

Pour créer un **work** à exécuter par **WorkManager**, étendre la classe **Worker** et surcharger la méthode **doWork().** Par exemple, pour créer un **Worker** qui télécharge des images, on procède comme suit :

```
public class UploadWorker extends Worker {
  public UploadWorker(
     @NonNull Context context,
     @NonNull WorkerParameters params) {
     super(context, params);
  }

  @Override
  public Result doWork() {

    // Faire le travail ici - dans ce cas, télécharger les images.
    uploadImages();

    // Indiquer si le travail s'est terminé avec succès avec le Résultat.
    return Result.success();
  }
}
```

Le résultat renvoyé par **doWork()** indique au service **WorkManager** si le travail a réussi et, en cas d'échec, si le travail doit être relancé ou non.

- **Resultat.success()** : le travail s'est terminé avec succès
- **Result.failure()** : le travail a échoué
- **Result.retry()** : le travail a échoué et doit être réessayé à un autre moment conformément à la politique de relance

PARTIE 2

Créer une demande de travail (WorkRequest)

Une fois que le **work** est défini, il doit être planifié avec le service **WorkManager** pour pouvoir être exécuté. **WorkManager** offre une grande souplesse dans la programmation de le **work**. On peut le programmer pour qu'il s'exécute périodiquement sur un intervalle de temps, ou le programmer pour qu'il s'exécute une seule fois.

Quelle que soit la façon de planifier le **work**, on utilise toujours un **WorkRequest**. Alors qu'un **Worker** définit l'unité de travail, une **WorkRequest** (et ses sous-classes) définit comment et quand elle doit être exécutée. Dans le cas le plus simple, on peut utiliser un **OneTimeWorkRequest**, comme le montre l'exemple suivant.

```
WorkRequest uploadWorkRequest =
  new OneTimeWorkRequest.Builder(UploadWorker.class)
    .build();
```

Soumettre la demande de work au système

Enfin, soumettre le **WorkRequest** au **WorkManager** à l'aide de la méthode **enqueue().**

```
WorkManager
    .getInstance(myContext)
    .enqueue(uploadWorkRequest);
```

L'heure exacte à laquelle le **worker** va être exécuté dépend des contraintes utilisées dans le **WorkRequest** et des optimisations du système. **WorkManager** est conçu pour donner le meilleur comportement sous ces restrictions.

Programmer des travaux accélérés

WorkManager 2.7.0 a introduit le concept de Work accéléré. Cela permet à **WorkManager** d'exécuter des travaux importants tout en donnant au système un meilleur contrôle sur l'accès aux ressources.

Le Work accéléré se distingue par les caractéristiques suivantes :

- **Importance** : le **Work** accéléré convient aux tâches qui sont importantes pour l'utilisateur ou qui sont initiées par l'utilisateur.
- **Vitesse** : le **Work** accéléré convient mieux aux tâches courtes qui commencent immédiatement et se terminent en quelques minutes.
- **Quotas** : un quota au niveau du système qui limite le temps d'exécution en avant-plan détermine si un Work accéléré peut démarrer.
- **Gestion de l'alimentation** : les restrictions de gestion de l'alimentation, telles que l'économiseur de batterie et la mise en veille, sont moins susceptibles d'affecter le Work accéléré.
- **Temps de latence** : le système exécute immédiatement les Works accélérés, à condition que la charge de **Work** actuelle du système lui permette de le faire. Cela signifie qu'ils sont sensibles à la latence et ne peuvent pas être programmés pour une exécution ultérieure.

Un cas d'utilisation potentiel pour le **Work** accéléré pourrait être dans une application de chat lorsque l'utilisateur veut envoyer un message ou une image jointe. De même, une application qui gère un flux de paiement ou d'abonnement pourrait également vouloir utiliser le Work accéléré. En effet, ces tâches sont importantes pour l'utilisateur, s'exécutent rapidement en arrière-plan et doivent être lancées immédiatement.

Quotas

Le système doit allouer le temps d'exécution de l'application à un **Work** accéléré avant qu'il ne puisse être exécuté. Le temps d'exécution n'est pas illimité. Il est plutôt limité par un quota. Lorsque l'application utilise son temps d'exécution et atteint son quota alloué, elle ne peut plus exécuter de Work accéléré jusqu'à ce que le quota soit actualisé. Cela permet à Android d'équilibrer plus efficacement les ressources entre les applications.

Chaque application dispose de son propre quota de temps d'exécution en avant-plan. La quantité de temps d'exécution disponible est basée sur le standby bucket et l'importance du processus.

Remarques

- Lorsque l'application est au premier plan, les quotas ne limitent pas l'exécution du **Work** accéléré. Un quota de temps d'exécution ne s'applique que lorsque l'application est en arrière-plan, ou lorsque l'application passe en arrière-plan. Ainsi, il est possible d'accélérer le Work en arrière-plan. En outre, il est possible d'utiliser **setForeground()** lorsque l'application est au premier plan.

Exécution des travaux accélérés

À partir de **WorkManager** 2.7, l'application peut appeler **setExpedited()** pour déclarer qu'une **WorkRequest** doit être exécutée le plus rapidement possible en utilisant un job accéléré. L'extrait de code suivant fournit un exemple d'utilisation de **setExpedited()** :

```
OneTimeWorkRequest request = new OneTimeWorkRequestBuilder<T>()
    .setInputData(inputData)
    .setExpedited(OutOfQuotaPolicy.RUN_AS_NON_EXPEDITED_WORK_REQUEST)
    .build();
```

Remarques

- Dans cet exemple, nous initialisons une instance de **OneTimeWorkRequest** et appelons **setExpedited()** sur celle-ci. Cette demande devient alors un **Work** accéléré. Si le quota le permet, elle commencera à s'exécuter immédiatement en arrière-plan.

Rétrocompatibilité et services d'avant-plan

Afin de maintenir la rétrocompatibilité pour les travaux accélérés, **WorkManager** peut exécuter un service d'avant-plan sur les versions de plateforme antérieures à Android 12. Les services d'avant-plan peuvent afficher une notification à l'utilisateur.

Les méthodes **getForegroundInfoAsync()** et **getForegroundInfo()** de **Worker** permettent à **WorkManager** d'afficher une notification lors de l'appel de **setExpedited()** avant Android 12.

Tout **ListenableWorker** doit implémenter la méthode **getForegroundInfo** si l'on souhaite demander que la tâche soit exécutée en tant que tâche accélérée.

En ciblant Android 12 ou plus, les services de premier plan restent à la disposition par le biais de la méthode **setForeground** correspondante.

Attention

- L'absence d'implémentation de la méthode correspondante **getForegroundInfo** peut entraîner des pannes d'exécution lors de l'appel de **setExpedited** sur les anciennes versions de la plateforme.
- **setForeground()** peut lancer des exceptions d'exécution sur Android 12, et pourrait lancer une exception si le lancement était restreint.

Planifier les travaux périodiques

L'application peut parfois exiger que certaines tâches soient exécutées périodiquement. Par exemple, sauvegarder périodiquement les données, télécharger du contenu récent dans l'application ou charger les journaux sur un serveur.

Voici comment utiliser le **PeriodicWorkRequest** pour créer un objet **WorkRequest** qui s'exécute périodiquement :

```
PeriodicWorkRequest saveRequest =
    new PeriodicWorkRequest.Builder(SaveImageToFileWorker.class, 1, TimeUnit.HOURS)
      // Constraints
      .build();
```

Dans cet exemple, le **Work** est planifié avec un intervalle d'une heure.

Remarque

- L'intervalle minimal de répétition qui peut être défini est de 15 minutes (comme dans l'API **JobScheduler**).

La période d'intervalle est définie comme le temps minimum entre les répétitions. L'heure exacte à laquelle le **Worker** va être exécuté dépend des contraintes utilisées dans le objet **WorkRequest** et des optimisations effectuées par le système.

PARTIE 2

Intervalles d'exécution flexibles

Si la nature de Work le rend sensible au moment de l'exécution, il est possible de configurer le **PeriodicWorkRequest** pour qu'il s'exécute dans une période flexible à l'intérieur de chaque période d'intervalle, comme le montre la figure 1.

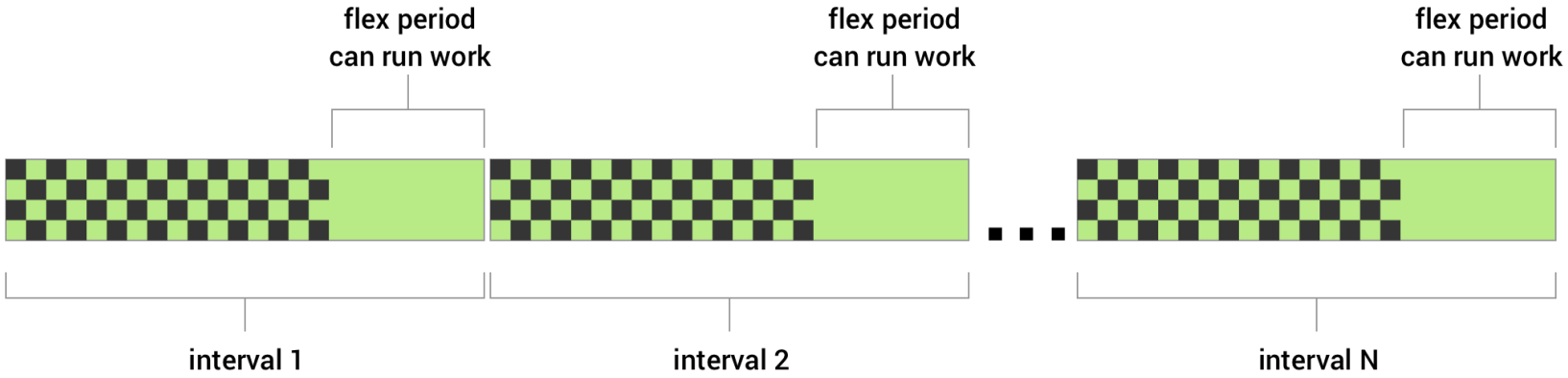


Figure 1. Le diagramme montre les intervalles répétitifs avec la période flexible dans laquelle le Work peut se dérouler (Réf).

Pour définir un Work périodique avec une période flexible, il suffit de transmettre un intervalle flexible (**flexInterval**) avec l'intervalle de répétition (**repeatInterval**) lors de la création de la demande de Work périodique (**PeriodicWorkRequest**). La période de flexion commence à **repeatInterval** - **flexInterval**, et va jusqu'à la fin de l'intervalle.

PARTIE 2

Exemple

Voici un exemple de **Work** périodique qui peut être exécuté pendant les 15 dernières minutes de chaque période d'une heure.

```
WorkRequest saveRequest =
    new PeriodicWorkRequest.Builder(SaveImageToFileWorker.class,
        1, TimeUnit.HOURS,
        15, TimeUnit.MINUTES)
    .build();
```

L'intervalle de répétition doit être supérieur ou égal à **PeriodicWorkRequest.MIN_PERIODIC_INTERVAL_MILLIS** et l'intervalle de flexion doit être supérieur ou égal à **PeriodicWorkRequest.MIN_PERIODIC_FLEX_MILLIS**.

Effet des contraintes sur le Work périodique

Il est possible d'appliquer des contraintes aux travaux périodiques. Par exemple, ajouter une contrainte à la demande de **Work** de sorte que le Work ne s'exécute que lorsque l'appareil de l'utilisateur est en charge. Dans ce cas, même si l'intervalle de répétition défini est dépassé, la demande de Work périodique ne sera pas exécutée tant que cette condition ne sera pas remplie. Ainsi, une exécution particulière de Work pourrait être retardée, voire ignorée, si les conditions ne sont pas remplies dans l'intervalle d'exécution.

Contraintes Work

Les contraintes garantissent que le **Work** est reporté jusqu'à ce que les conditions optimales soient réunies. Les contraintes suivantes sont disponibles pour **WorkManager**.

NetworkType	Permet de limiter le type de réseau nécessaire à l'exécution de Work. Par exemple, Wi-Fi (UNMETERED).
BatteryNotLow	Lorsqu'il a la valeur true, le **Work** ne sera pas exécuté si l'appareil est en mode batterie faible.
RequiresCharging	Lorsqu'il est défini sur true, le **Work** ne sera exécuté que lorsque l'appareil est en charge.
DeviceIdle	Lorsqu'elle est définie sur true, l'appareil de l'utilisateur doit être inactif avant que le **Work** ne soit exécuté. Cela peut être utile pour exécuter des opérations groupées qui pourraient avoir un impact négatif sur les performances d'autres applications fonctionnant activement sur l'appareil de l'utilisateur.
StorageNotLow	Lorsque cette option est définie sur true, le **Work** ne sera pas exécuté si l'espace de stockage de l'utilisateur sur l'appareil est trop faible.

Exemple

Pour créer un ensemble de contraintes et l'associer à un **Work**, créer une instance de contrainte à l'aide de **Contraints.Builder()** et l'affecter à **WorkRequest.Builder()**.

Par exemple, le code suivant construit une demande de **Work** qui ne s'exécute que lorsque l'appareil de l'utilisateur est à la fois en charge et en Wi-Fi :

```
Constraints constraints = new Constraints.Builder()
    .setRequiredNetworkType(NetworkType.UNMETERED)
    .setRequiresCharging(true)
    .build();

WorkRequest myWorkRequest =
    new OneTimeWorkRequest.Builder(MyWork.class)
        .setConstraints(constraints)
        .build();
```

Lorsque plusieurs contraintes sont spécifiées, le **Work** ne sera exécuté que si toutes les contraintes sont respectées.

Si une contrainte n'est pas respectée pendant l'exécution de **Work**, **WorkManager** arrêtera **Worker**. Le **Work** sera alors relancé lorsque toutes les contraintes seront satisfaites.

Work retardé

Si un **Work** n'a pas de contraintes ou si toutes les contraintes sont respectées lorsque le Work est mis en file d'attente, le système peut décider d'exécuter le Work immédiatement. Si le Work ne doit pas être exécuté immédiatement, il est possible de spécifier qu'il doit être lancé après un délai initial minimum.

Voici un exemple de la façon de configurer le Work pour qu'il soit exécuté au moins 10 minutes après avoir été mis en file d'attente.

```
WorkRequest myWorkRequest =
    new OneTimeWorkRequest.Builder(MyWork.class)
        .setInitialDelay(10, TimeUnit.MINUTES)
        .build();
```

Bien que l'exemple illustre comment définir un délai initial pour une demande de Work unique (**OneTimeWorkRequest**), il est aussi possible de définir un délai initial pour une demande de Work périodique (**PeriodicWorkRequest**). Dans ce cas, seule la première exécution de l'opération périodique sera retardée.

Remarque

L'heure exacte à laquelle le **worker** va être exécuté dépend également des contraintes utilisées dans la demande de Work et des optimisations du système. **WorkManager** est conçu pour donner le meilleur comportement possible sous ces restrictions.

Politique de relance et de backoff

Si le **Workmanager** doit relancer le Work, il peut renvoyer **Result.retry()** à partir du **Worker**. Le Work est alors reprogrammé en fonction d'un délai de relance et d'une politique de relance.

- Le délai de relance (**Backoff delay**) spécifie le temps minimum à attendre avant de réessayer le Work après la première tentative. Cette valeur ne peut être inférieure à 10 secondes (ou MIN_BACKOFF_MILLIS) ;
- La politique de **backoff** (Backoff policy) définit comment le délai de **backoff** doit augmenter au fil du temps pour les tentatives ultérieures. **WorkManager** supporte 2 politiques de backoff, **LINEAR** et **EXPONENTIEL**.

Chaque demande de **Work** a une politique de **backoff** et un délai de **backoff**. La politique par défaut est EXPONENTIELLE avec un délai de 10 secondes, mais il est possible de la modifier dans la configuration de la demande de Work.

Voici un exemple de personnalisation du délai et de la politique de **backoff**.

```
WorkRequest myWorkRequest =
    new OneTimeWorkRequest.Builder(MyWork.class)
        .setBackoffCriteria(
            BackoffPolicy.LINEAR,
            OneTimeWorkRequest.MIN_BACKOFF_MILLIS,
            TimeUnit.MILLISECONDS)
        .build();
```

Dans cet exemple, le délai de récupération minimum est fixé à la valeur minimale autorisée, soit 10 secondes. Comme la politique est LINEAIRE, l'intervalle de tentative augmentera d'environ 10 secondes à chaque nouvelle tentative. Par exemple, la première exécution se terminant par **Result.retry()** sera tentée à nouveau après 10 secondes, puis 20, 30, 40, et ainsi de suite, si le **Work** continue à renvoyer **Result.retry()** après les tentatives suivantes. Si la politique de **backoff** était définie sur **EXPONENTIEL**, la séquence de durée des tentatives serait plus proche de 20, 40, 80, et ainsi de suite.

Les délais de temporisation sont inexacts et peuvent varier de plusieurs secondes entre les tentatives, mais ne seront jamais inférieurs au délai de temporisation initial spécifié dans la configuration.

Tag work

Chaque demande de **Work** possède un identifiant unique, qui peut être utilisé pour identifier ce Work ultérieurement afin d'annuler le **Work** ou d'observer son avancement.

Si un groupe de travaux est logiquement lié, il peut aussi être utile de baliser ces éléments de Work. Le balisage permet d'opérer avec un groupe de demandes de Work ensemble.

Par exemple :

WorkManager.cancelAllWorkByTag(String) annule toutes les demandes de Work avec une étiquette particulière, et **WorkManager.getWorkInfosByTag(String)** renvoie une liste des objets **WorkInfo** qui peuvent être utilisés pour déterminer l'état actuel du Work.

Le code suivant montre comment ajouter une balise **"cleanup"** à un Work :

```
WorkRequest myWorkRequest =
    new OneTimeWorkRequest.Builder(MyWork.class)
    .addTag("cleanup")
    .build();
```

Enfin, plusieurs étiquettes peuvent être ajoutées à une seule demande de Work. En interne, ces balises sont stockées sous la forme d'un ensemble de chaînes de caractères. Pour obtenir l'ensemble des balises associées à la demande de Work, il suffit d'utiliser **WorkInfo.getTags().**

À partir de la classe **Worker**, il est possible de récupérer son ensemble de balises via **ListenableWorker.getTags().**

Affecter les données d'entrée

Le **Work** peut nécessiter des données d'entrée pour pouvoir être exécuté. Par exemple, un Work qui gère le téléchargement d'une image peut nécessiter l'URI de l'image à télécharger en entrée.

Les valeurs d'entrée sont stockées sous forme de paires clé-valeur dans un objet Data et peuvent être définies dans la demande de Work. **WorkManager** fournira les données d'entrée au Work lorsqu'il l'exécutera. La classe **Worker** peut accéder aux arguments d'entrée en appelant **Worker.getInputData().** Le code ci-dessous montre comment créer une instance de **Worker** qui nécessite des données d'entrée et comment les envoyer dans la demande de **Work**.

```
// Définir le Worker nécessitant des données
public class UploadWork extends Worker {

  public UploadWork(Context appContext, WorkerParameters workerParams) {
    super(appContext, workerParams);
  }

  @NonNull
  @Override
  public Result doWork() {
    String imageUriInput = getInputData().getString("IMAGE_URI");
    if(imageUriInput == null) {
      return Result.failure();
    }

    uploadFile(imageUriInput);
    return Result.success();
  }
  ...
}

// Créer une WorkRequest pour le Worker et l'envoyer en entrée
WorkRequest myUploadWork =
    new OneTimeWorkRequest.Builder(UploadWork.class)
        .setInputData(
            new Data.Builder()
                .putString("IMAGE_URI", "http://...")
                .build()
        )
        .build();
```

PARTIE 2

CHAPITRE 1

Utiliser le Work Manager

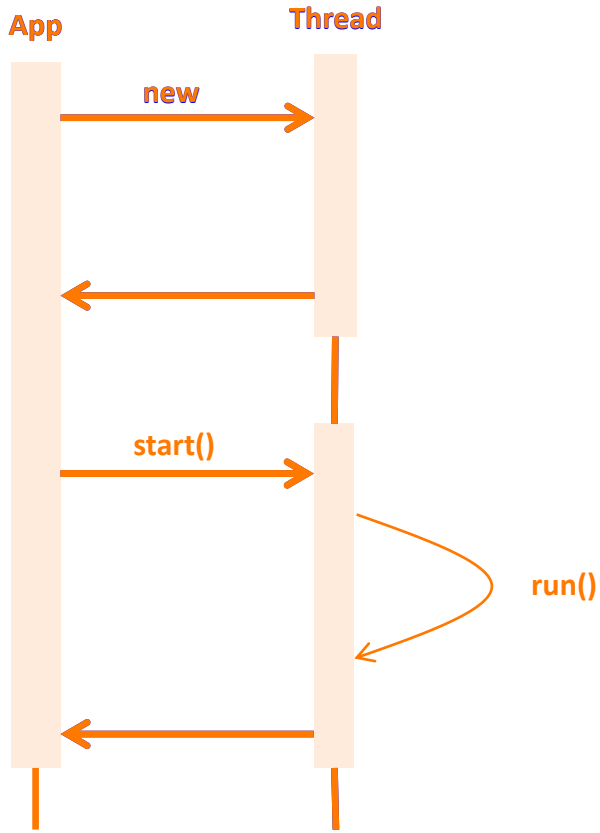
1. Utilisation du Workmanager
2. Manipulation des Threads

Thread

- **Définition** : un Thread désigne un point d'exécution dans le programme. En fait, le langage Java est multi-Thread, c'est à dire qu'il peut exécuter du code à plusieurs endroits de façon indépendante.

  Exemple : un client FTP peut télécharger plusieurs fichiers, naviguer sur plusieurs serveurs en même temps, chaque connexion étant gérée par un Thread différent.

- **Création** :
  - **Dériver de la class Thread :** il est possible de créer une nouvelle classe qui dérive de la classe **java.lang.Thread**. Il suffit ensuite de redéfinir la méthode **run()**. C'est cette méthode que le Thread va exécuter
  - **Implémenter java.lang.Runnable :** Il est possible de créer une classe qui implémente l'interface **java.lang.Runnable** et définir la méthode **run().** Ensuite il suffit de créer un objet **java.lang.Thread** en lui passant la classe en paramètre
- **Méthodes** :
  - Pour lancer l'exécution d'un Thread, exécuter la méthode **start()**, elle qui lance la méthode **run()**
  - On peut simuler une "pause" dans l'exécution d'une application en utilisant la méthode **sleep()** de la classe **Thread**. Cette méthode force le Thread courant à cesser son exécution pendant le temps passé en paramètres



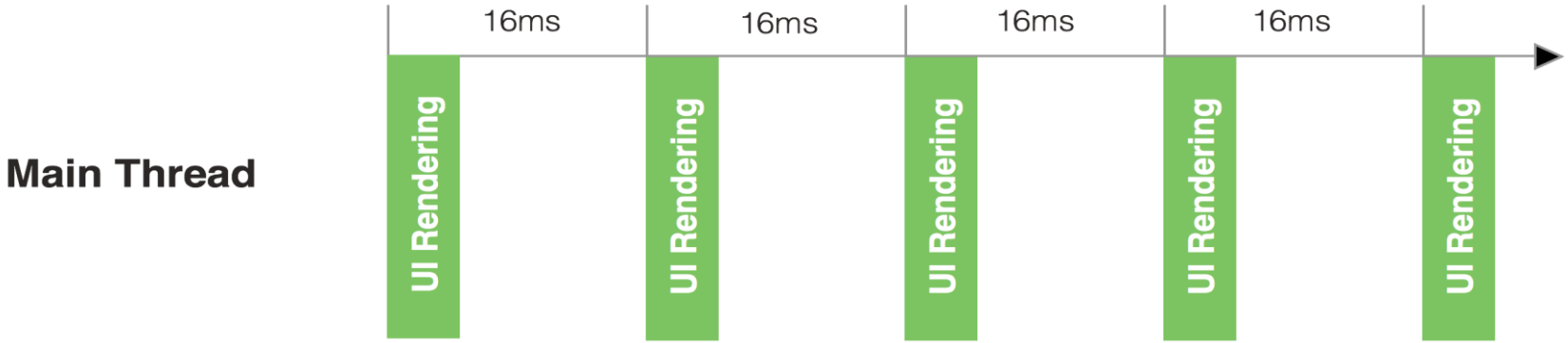
PARTIE 2

Main Thread

- Chemin d'exécution indépendant dans un programme en cours d'exécution ;
- Le code est exécuté ligne par ligne ;
- L'application fonctionne sur un Thread Java appelé "main" ou "UI Thread" ;
- Dessine l'interface utilisateur sur l'écran ;
- Répond aux actions de l'utilisateur en traitant les événements de l'interface utilisateur.

Main Thread doit être rapide

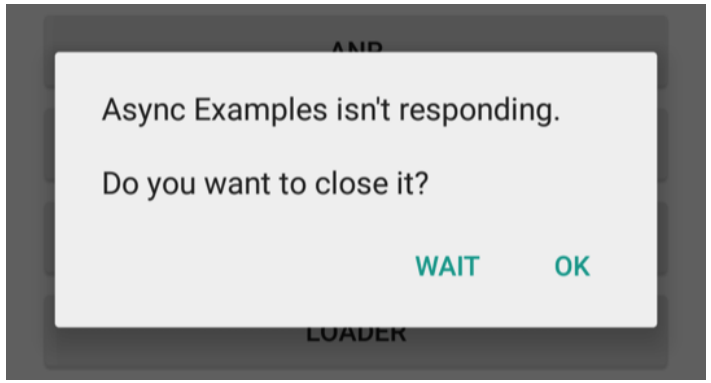
- Le matériel met à jour l'écran toutes les 16 millisecondes ;
- Le Thread de l'interface utilisateur a 16 ms pour faire tout son travail ;
- Si cela prend trop de temps, l'application bégaie ou se bloque.



Réf

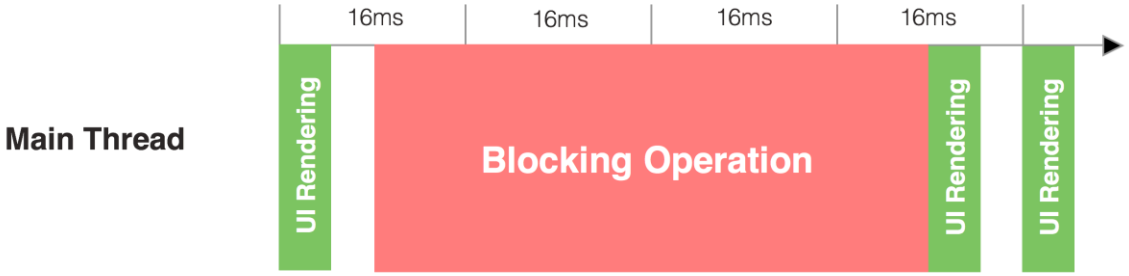
Les utilisateurs désinstallent les applications qui ne répondent pas à leurs besoins

- Si l'interface utilisateur attend trop longtemps la fin d'une opération, elle ne répond plus
- Le cadre affiche une boîte de dialogue Application Not Responding (ANR)



Qu'est-ce qu'une tâche de longue durée ?

- Fonctionnement du réseau
- Calculs longs
- Téléchargement/téléchargement de fichiers
- Traitement des images
- Chargement des données



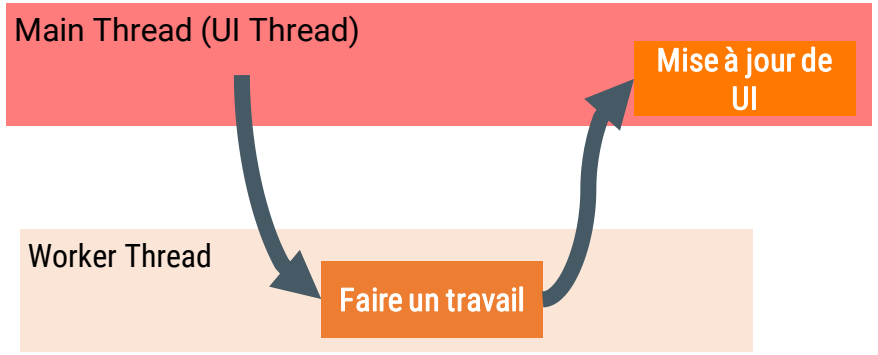
Réf

PARTIE 2

Threads d'arrière plan

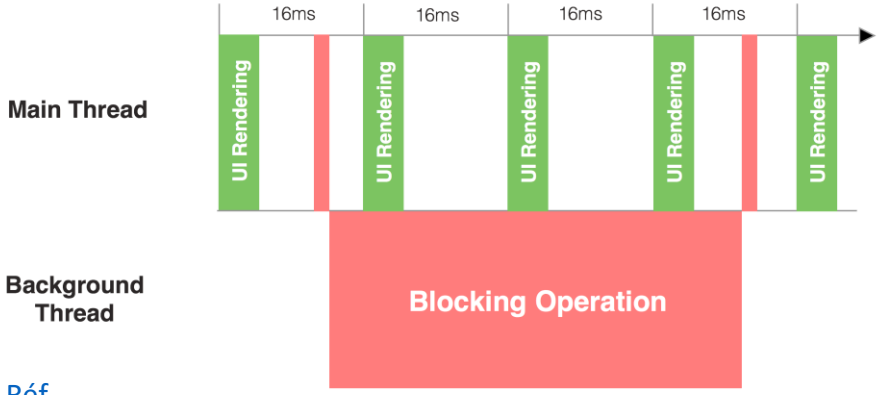
Exécution de tâches longues sur un Thread d'arrière-plan :

- AsyncTask
- Le framework Loader
- Services



Deux règles pour les Threads Android

- Ne pas bloquer le Thread de l'interface utilisateur :
  - Effectuer tout le travail en moins de 16 ms pour chaque écran
  - Exécuter les travaux lents non UI sur un Thread non UI (User Interface)
- Ne pas accéder à la boîte à outils Android UI en dehors du Thread UI :
  - Effectuer le travail de l'interface utilisateur uniquement sur le Thread de l'interface utilisateur

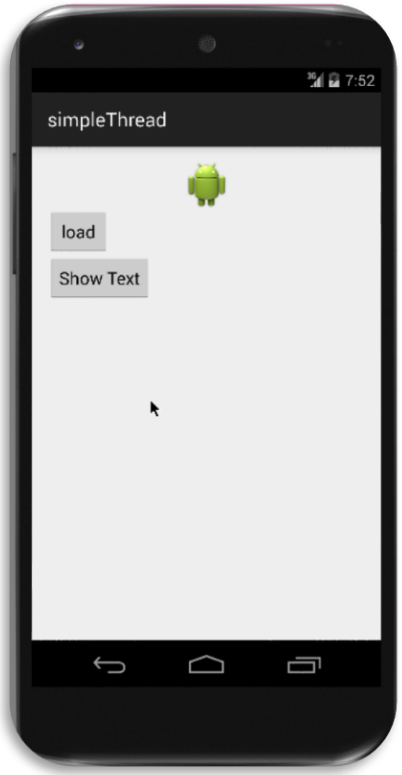


Réf

PARTIE 2

Thread : Exemple

- Le premier bouton permet de charger une image, le deuxième permet d'afficher un message texte. On souhaite que le traitement se fasse d'une façon indépendante, sans que le chargement de l'image ne bloque l'affiche du message texte.




```
private ImageView mView;
private Bitmap mBitmap;
private int delay = 500;

@Override
protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   mView = (ImageView)findViewById(R.id.imageView);
   final Button loadIcon = (Button)findViewById(R.id.loadIcon);
   loadIcon.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
         loadIcon();
      }
   });

   final Button showText = (Button)findViewById(R.id.show);
   showText.setOnClickListener(new View.OnClickListener() {
      @Override
      public void onClick(View v) {
         Toast.makeText(getApplicationContext(), "Je travaille", Toast.LENGTH_SHORT).show();
      }
   });
}

private void loadIcon(){
      new Thread(new Runnable() {
         @Override
         public void run() {
            try {
               Thread.sleep(delay);
            } catch (InterruptedException e) {
               e.printStackTrace();
            }
            mBitmap = BitmapFactory.decodeResource(getResources(), R.drawable.icon);
            mView.setImageBitmap(mBitmap);
         }
      }).start();
}
```

PARTIE 2

Thread : Thread UI

- Ne jamais bloquer le Thread UI.
- Ne pas manipuler les vues standards en dehors du Thread UI.
- Sur un **View**, on peut faire **boolean post(Runnable action)** pour ajouter le **Runnable** à la pile des messages du Thread UI. Le boolean retourné vaut **true** s'il a été correctement placé dans la pile des messages.


```
private void loadIcon(){
        new Thread(new Runnable() {
          @Override
          public void run() {
            try {
              Thread.sleep(delay);
            } catch (InterruptedException e) {
              e.printStackTrace();
            }
            mBitmap = BitmapFactory.decodeResource(getResources(),
R.drawable.icon);
            mView.post(new Runnable() {
              @Override
              public void run() {
                mView.setImageBitmap(mBitmap);
              }
            });
          }
        }).start();
    }
```

PARTIE 2

Thread : Problème

- La méthode d'Activity **void runOnUiThread(Runnable action)** spécifie qu'une action doit s'exécuter dans le Thread UI. Si le Thread actuel est le Thread UI, alors l'action est exécutée immédiatement. Sinon, l'action est ajoutée à la pile des évènements du Thread UI.


```
private void loadIcon(){
        new Thread(new Runnable() {
          @Override
          public void run() {
            try {
              Thread.sleep(delay);
            } catch (InterruptedException e) {
              e.printStackTrace();
            }
            mBitmap = BitmapFactory.decodeResource(getResources(), R.drawable.icon);
            runOnUiThread.post(new Runnable() {
              @Override
              public void run() {
                mView.setImageBitmap(mBitmap);
              }
            });
          }
        }).start();
  }
```

Bonne pratique

Si l'activité crée sa hiérarchie de vues lors d'un appel à sa méthode **onCreate(),** comment savoir dans quel **Thread onCreate()** a été appelé ?

- Utiliser **LogCat** pour imprimer l'ID du Thread
- Regarder le code pour voir si l'appel provient d'un de vos Threads ou d'un des Threads du système
- **onCreate()** est toujours appelé dans un Thread spécial créé spécifiquement pour l'activité lorsqu'elle démarre
- **onCreate()** est exécuté dans le Thread de l'interface utilisateur

CHAPITRE 2

Utiliser le Job scheduler

Ce que vous allez apprendre dans ce chapitre :

- Implémentation de Job scheduler
- Implémentation de l'AsyncTask

3 heures

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE 2

Utiliser le Job scheduler

1. **Implémentation de Job scheduler**
2. Implémentation de l'AsyncTask

Qu'est-ce que le Job Scheduler ?

- Utilisé pour la programmation intelligente des tâches d'arrière-plan
- Basé sur des conditions, et non sur un calendrier
- Beaucoup plus efficace que **AlarmManager**
- Regroupe les tâches pour minimiser la consommation de la batterie
- API 21+ (pas dans la bibliothèque de support)

Composants du Job Scheduler

- **JobService** : Classe de service où la tâche est initiée
- **JobInfo** : Modèle de construction permettant de définir les conditions de la tâche
- **JobScheduler** : planifie et annule les tâches, lance le service

JobService

- Sous-classe de **JobService**, implémente la tâche dans cette classe
- Surcharge de :
    - onStartJob()
    - onStopJob()
- S'exécute sur le Thread principal

onStartJob()

- Implémente le travail à réaliser dans cette méthode
- Appelé par le système lorsque les conditions sont remplies
- S'exécute sur le Thread principal
- Décharge le travail lourd sur un autre Thread

onStartJob() retourne un boolean

FALSE - Travail terminé.

TRUE

- Le travail a été déchargé
- Doit appeler **jobFinished()** depuis le **worker thread**
- Passer l'objet **JobParams** depuis **onStartJob().**

onStopJob()

- Appelé si le système a déterminé que l'exécution du job doit s'arrêter
- ... parce que les conditions spécifiées ne sont plus remplies
- Par exemple, il n'y a plus de Wi-Fi, l'appareil n'est plus inactif
- Avant **jobFinished(JobParameters, boolean)**
- Renvoie TRUE pour replanifier

Code basique d'un Job

```
public class MyJobService extends JobService {
    private UpdateAppsAsyncTask updateTask = new UpdateAppsAsyncTask();
    @Override
    public boolean onStartJob(JobParameters params) {
        updateTask.execute(params);
        return true; // le travail a été déchargé
    }
     @Override
     public boolean onStopJob(JobParameters jobParameters) {
        return true;
     }
}
```

Enregistrer le JobService

```
<service

   android:name=".NotificationJobService"

   android:permission=

        "android.permission.BIND_JOB_SERVICE"/>
```

JobInfo

- Définir les conditions d'exécution
- Objet **JobInfo.Builder**

Objet de construction JobInfo

- Arg 1 : ID du job
- Arg 2 : composant du service
- Arg 3 : JobService à lancer

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

Conditions de mise en place

setRequiredNetworkType(int networkType)

setBackoffCriteria(long initialBackoffMillis, int backoffPolicy)

setMinimumLatency(long minLatencyMillis)

setOverrideDeadline(long maxExecutionDelayMillis)

setPeriodic(long intervalMillis)

setPersisted(boolean isPersisted)

setRequiresCharging(boolean requiresCharging)

setRequiresDeviceIdle(boolean requiresDeviceIdle)

setRequiredNetworkType()

`setRequiredNetworkType(int networkType)`

- NETWORK_TYPE_NONE : par défaut, aucun réseau n'est requis
- NETWORK_TYPE_ANY : requiert une connectivité réseau
- NETWORK_TYPE_NOT_ROAMING : demande une connectivité réseau non itinérante
- NETWORK_TYPE_UNMETERED : demande une connectivité réseau non mesurée

setMinimumLatency()

`setMinimumLatency(long minLatencyMillis)`

- Nombre minimal de millisecondes à attendre avant de terminer la tâche.

setPeriodic()

`setPeriodic(long intervalMillis)`

- Répète la tâche après un certain temps
- Passe dans l'intervalle de répétition
- Mutuellement exclusif avec les conditions de latence minimale et de dépassement de délai
- L'exécution de la tâche n'est pas garantie dans la période donnée

setOverrideDeadline()

`setOverrideDeadline(long maxExecutionDelayMillis)`

- Nombre maximal de millisecondes à attendre avant d'exécuter la tâche, même si les autres conditions ne sont pas remplies

setPersisted()

`setPersisted(boolean isPersisted)`

- Indique si le travail est conservé lors des redémarrages du système
- Passer en True ou False
- Requiert l'autorisation RECEIVE_BOOT_COMPLETED

setRequiresCharging()

`setRequiresCharging(boolean requiresCharging)`

- Si le dispositif doit être branché
- Passer en True ou False
- La valeur par défaut est False

setRequiresDeviceIdle()

`setRequiresDeviceIdle(boolean requiresDeviceIdle)`

- Si le dispositif doit être en mode inactif
- Le mode inactif est une définition libre du système, lorsque le périphérique n'est pas utilisé et ne l'a pas été depuis un certain temps
- A utiliser pour les travaux nécessitant beaucoup de ressources
- Passer en True ou False. La valeur par défaut est False

Code JobInfo

```
JobInfo.Builder builder = new JobInfo.Builder(
        JOB_ID, new ComponentName(getPackageName(),
        NotificationJobService.class.getName()))
          .setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED)
          .setRequiresDeviceIdle(true)
          .setRequiresCharging(true);
JobInfo myJobInfo = builder.build();
```

Planification du job

1. Obtenir un objet **JobScheduler** du système ;
2. Appeler **schedule()** sur **JobScheduler**, avec l'objet **JobInfo**.

```
mScheduler =

    (JobScheduler)getSystemService(JOB_SCHEDULER_SERVICE);

mScheduler.schedule(myJobInfo);
```

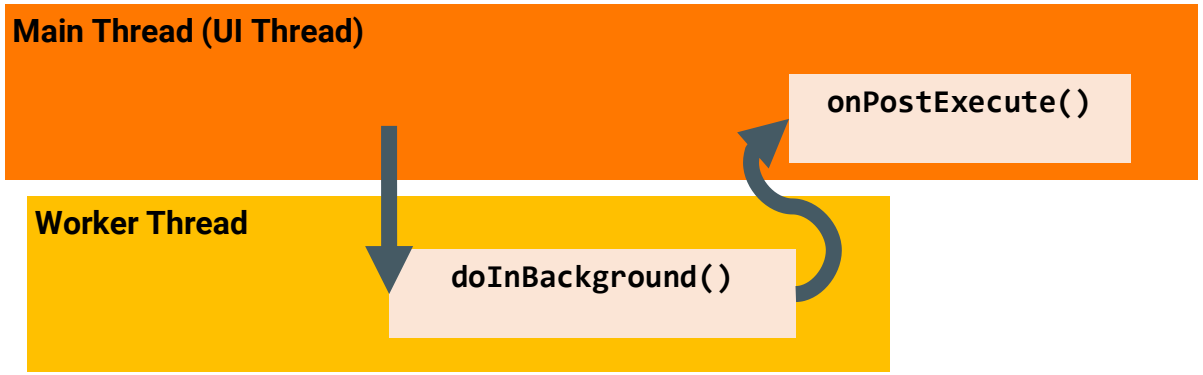
CHAPITRE 2

Utiliser le Job scheduler

1. Implémentation de Job scheduler
2. **Implémentation de l'AsyncTask**

Qu'est-ce qu'AsyncTask ?

Utiliser **AsyncTask** pour mettre en œuvre des tâches d'arrière-plan de base



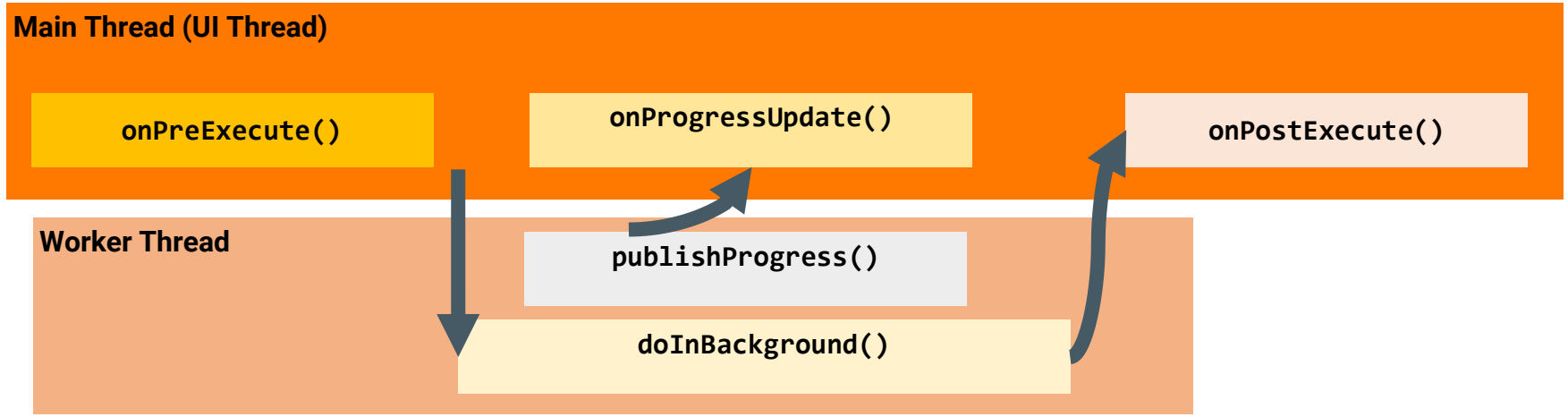
Redéfinir deux méthodes

- **doInBackground()** : exécution sur un Thread d'arrière-plan :
  - Tout le travail doit être effectué en arrière-plan
- **onPostExecute()** : exécution sur le Thread principal lorsque le travail est terminé :
  - Traitement des résultats
  - Publier les résultats sur l'interface utilisateur

Méthodes auxiliaires d'AsyncTask

- onPreExecute() :
  - S'exécute sur le Thread principal ;
  - Met en place la tâche.

- onProgressUpdate() :
  - S'exécute sur le Thread principal ;
  - Reçoit les appels de **publishProgress()** du Thread d'arrière-plan.



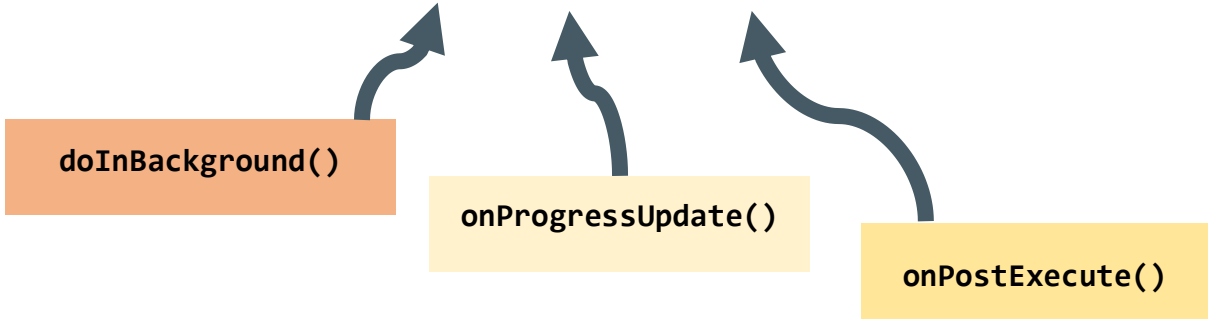
Création d'AsyncTask

- Sous-classe **AsyncTask**
- Fournir le type de données envoyé à **doInBackground()**
- Fournir le type de données des unités de progression pour **onProgressUpdate()**
- Fournir le type de données du résultat pour **onPostExecute()**

```
private class MyAsyncTask
                extends AsyncTask<URL, Integer, Bitmap> {...}
```

Définition de la classe MyAsyncTask

```
private class MyAsyncTask
                        extends AsyncTask<String, Integer, Bitmap> {...}
```



- **String** : peut être une requête, URI pour le nom de fichier ;
- **Integer** : un pourcentage complété, étapes effectuées ;
- **Bitmap** : une image à afficher ;
- Utiliser **Void** si aucune donnée n'est transmise.

PARTIE 2

onPreExecute()

```
protected void onPreExecute() {

    // afficher une barre de progression

    // afficher un toast

}
```

doInBackground()

```
protected Bitmap doInBackground(String... query) {

    // Obtenir le bitmap

    return bitmap;

}
```

onProgressUpdate()

```
protected void onProgressUpdate(Integer... progress) {

        setProgressPercent(progress[0]);

}
```

onPostExecute()

```
protected void onPostExecute(Bitmap result) {

   // Faites quelque chose avec le bitmap

}
```

Démarrer le travail d'arrière-plan

```
public void loadImage (View view) {

    String query = mEditText.getText().toString();

    new MyAsyncTask(query).execute();

}
```

Limitations d'AsyncTask

- Lorsque la configuration du dispositif change, l'activité est détruite
- **AsyncTask** ne peut plus se connecter à **Activity**
- Nouvelle **AsyncTask** créée à chaque changement de configuration
- Les anciennes **AsyncTasks** restent en place
- L'application peut manquer de mémoire ou planter

Quand utiliser AsyncTask ?

- Tâches courtes ou interruptibles ;
- Tâches qui n'ont pas besoin de faire un rapport à l'IU ou à l'utilisateur ;
- Tâches de moindre priorité qui peuvent être laissées inachevées ;
- Sinon utiliser **AsyncTaskLoader**.

Qu'est-ce qu'un Loader ?

- Permet le chargement asynchrone des données ;
- **Se reconnecte à l'activité après un changement de configuration ;**
- Peut surveiller les changements dans la source de données et fournir de nouvelles données ;
- Rappels implémentés dans l'activité ;
- Plusieurs types de chargeurs disponibles :
    - AsyncTaskLoader, CursorLoader

Pourquoi utiliser des loaders ?

- Exécution des tâches hors du Thread de l'interface utilisateur ;
- **LoaderManager** gère les changements de configuration pour les utilisateurs ;
- Mis en œuvre efficacement par le framework ;
- Les utilisateurs n'ont pas à attendre le chargement des données.

Qu'est-ce qu'un LoaderManager ?

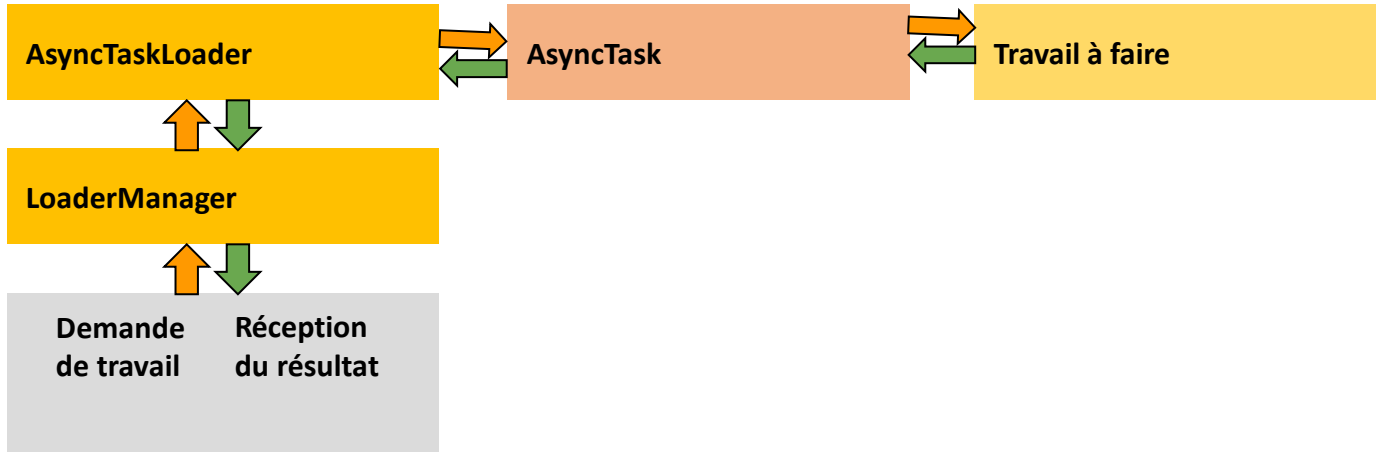
- Gère les fonctions du loader via des callbacks
- Peut gérer plusieurs loaders
    - Loader pour les données de base de données, pour les données **AsyncTask**, pour les données internet...

Obtenir un loader avec initLoader()

- Crée et démarre un loader, ou réutilise un loader existant, y compris ses données ;
- Utiliser **restartLoader()** pour effacer les données du loader existant :

```
getLoaderManager().initLoader(Id, args, callback);
getLoaderManager().initLoader(0, null, this);
getSupportLoaderManager().initLoader(0, null, this);
```

Aperçu d'AsyncTaskLoader



AsyncTask → AsyncTaskLoader

`doInBackground()`	→	`loadInBackground()`
`onPostExecute()`	→	`onLoadFinished()`

Etapes pour la sous-classe AsyncTaskLoader

1. Créer une sous-classe **AsyncTaskLoader**
2. Implémenter le constructeur
3. **loadInBackground()**
4. **onStartLoading()**

Créer une sous-classe AsyncTaskLoader

```
public static class StringListLoader
        extends AsyncTaskLoader<List<String>> {

     public StringListLoader(Context context, String queryString) {
        super(context);
        mQueryString = queryString;
     }
}
```

loadInBackground()

```
public List<String> loadInBackground() {
    List<String> data = new ArrayList<String>;
    //TODO: Charger les données à partir du réseau ou d'une base de données.
    return data;
}
```

onStartLoading()

Lorsque **restartLoader()** ou **initLoader()** est appelé, le **LoaderManager** invoque la fonction de rappel **onStartLoading()**

- Vérifier les données mises en cache
- Commence à observer la source de données (si nécessaire)
- Appeler **forceLoad()** pour charger les données s'il y a des changements ou s'il n'y a pas de données mises en cache :

```
protected void onStartLoading() {  forceLoad();  }
```

Implémenter les callbacks du loader dans l'activité

- **onCreateLoader()** : crée et renvoie un nouveau loader pour l'ID donné
- **onLoadFinished()** : appelé quand un loader précédemment créé a terminé son chargement
- **onLoaderReset()** : appelé lorsqu'un loader précédemment créé est réinitialisé et que ses données sont indisponibles

onCreateLoader()

```
@Override
public Loader<List<String>> onCreateLoader(int id, Bundle args) {
    return new StringListLoader(this,args.getString("queryString"));
}
```

onLoadFinished()

Les résultats de **loadInBackground()** sont transmis à **onLoadFinished()** où il est possible de les afficher :

```
public void onLoadFinished(Loader<List<String>> loader, List<String> data) {
    mAdapter.setData(data);
}
```

onLoaderReset()

- Appelé seulement quand le loader est détruit
- Laisser vide la plupart du temps

```
@Override
public void onLoaderReset(final LoaderList<String>> loader) { }
```

Obtenir un loader avec initLoader()

- Dans l'activité
- Utiliser la bibliothèque de support pour être compatible avec plus de dispositifs :

```
getSupportLoaderManager().initLoader(0, null, this);
```

PARTIE 3

Manipuler les permissions

Dans ce module, vous allez :

- Identification correcte des contraintes de chaque version d'Android
- Identification juste des différentes permissions
- Manipulation facile des permissions

7 heures

CHAPITRE 1

Déclarer une permission

Ce que vous allez apprendre dans ce chapitre :

- Présentation du fichier Manifest.xml
- Utilisation des Install-time permissions
- Utilisation des Runtime permissions

1 heure

CHAPITRE 1

Déclarer une permission

01 – Déclarer une permission

Présentation du fichier Manifest.xml

Avant-propos

Manifest.xml :

- Est un Fichier XML
- Précise l´architecture de l'application
- Chaque application doit en avoir un
- AndroidManifest.xml est dans la racine du projet

Contenu

- Précise le nom du package java utilisant l'application. Cela sert d'identifiant unique !
- Décrit les composants de l'application
  - Liste des **activités**, **services**, **broadcast receivers** ;
  - Précise les classes qui les implémentent
  - Précise leurs **capacités** (à quels Intents ils réagissent)

➢ Ceci permet au système de savoir comment lancer chaque partie de l'application afin de satisfaire au principe de réutilisabilité.

- Définit les **permissions** de l'application :
  - Droit de passer des appels
  - Droit d'accéder à Internet
  - Droit d'accéder au GPS
- Précise la **version d'Android** minimum nécessaire
- Déclare les **librairies** utilisées
- Déclare des outils **d'Instrumentation** (uniquement pour le développement)

PARTIE 3

01 – Déclarer une permission

Présentation du fichier Manifest.xml

Conventions

Seuls deux éléments sont obligatoires :

- <manifest>: contient le **package**, la **version**... Englobe tout le fichier
- <application> : **décrit l'application** et contiendra la liste de ses composants
- Les données sont passées en tant qu'attribut et non en tant que contenu
- Tous les **attributs commencent** par android: (sauf quelques uns dans <manifest>)

Les ressources

- Au lieu de contenir les données en tant que tel le fichier **manifest** peut faire appel à des ressources
- <activityandroid: icon=" @drawable=smallPic":: :>
- Ces ressources sont définies dans le répertoire **res** de l'application

PARTIE 3

01 – Déclarer une permission

Présentation du fichier Manifest.xml

Permissions

- Une application ne peux pas utiliser certaines fonctionnalités **sauf** si précisé dans le fichier manifest
- Il faut donc préciser les **permissions** nécessaires grâce à : <uses-permission>
- Il existe des **permissions standards** :
    - android.permission.CALL_EMERGENCY_NUMBERS
    - android.permission.READ_OWNER_DATA
    - android.permission.SET_WALLPAPER
    - android.permission.DEVICE_POWER
- Il est possible de définir ses propres permissions

Intent Filter

- Ils informent le système à quels Intent les composants **peuvent réagir**
- Un composant peut avoir plusieurs filtres
- **Exemple** : Cas d'un éditeur de texte :
    - Filtre pour éditer un document existant
    - Filtre pour initier un nouveau document
- Un filtre **doit posséder** une "action" qui définit à quoi il correspond

PARTIE 3

Exemple

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="ma.projet.android.myapplication">
    <uses-permission android:name="android.permission.INTERNET"/>
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MyApplication">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

CHAPITRE 1

Déclarer une permission

1. Présentation du fichier Manifest.xml
2. **Utilisation des Install-time permissions et Runtime permissions**

Permissions sur Android

Les permissions d'application aident à préserver la vie privée des utilisateurs en protégeant l'accès aux éléments suivants :

- **Données restreintes**, telles que l'état du système et les informations de contact d'un utilisateur
- **Actions restreintes**, telles que la connexion à un appareil jumelé et l'enregistrement audio

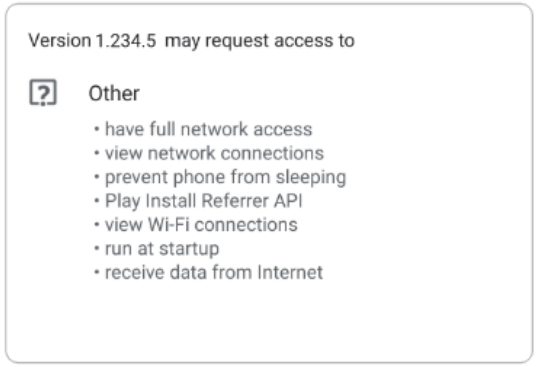
Types de permissions

Android classe les permissions en différents types, notamment les permissions d'installation, les **permissions d'exécution** et les **permissions spéciales**. Le type de chaque permission indique l'étendue des **données restreintes** auxquelles l'application peut accéder, et l'étendue des **actions restreintes** que l'application peut effectuer, lorsque le système accorde cette permission à l'application.

PARTIE 3

Install-time permissions

Les permissions d'installation donnent à l'application un accès limité à des données restreintes et lui permettent d'effectuer des actions restreintes qui n'affectent que très peu le système ou les autres applications. Les permissions d'installation sont accordées automatiquement par le système lorsque l'utilisateur installe l'application. L'app store présente un avis d'autorisation d'installation à l'utilisateur lorsqu'il consulte la page de détails d'une application, comme le montre la figure suivante.



Android comprend plusieurs sous-types de permissions d'installation, notamment les permissions normales et les permissions de signature.

Figure. La liste des autorisations d'installation d'une application, qui apparaît dans une App Store.

Types de permissions

- Permissions normales :
  - Ces permissions permettent d'accéder à des données et à des actions qui vont au-delà de la sandbox de l'application. Toutefois, ces données et actions présentent très peu de risques pour la vie privée de l'utilisateur et le fonctionnement des autres applications ;
  - Le système attribue le niveau de protection "normal" aux permissions normales, comme indiqué sur la page de référence de l'API des permissions.
- Permissions de signature :
  - Si l'application déclare une autorisation de signature qu'une autre application a définie, et si les deux applications sont signées par le même certificat, le système accorde l'autorisation à la première application au moment de l'installation. Dans le cas contraire, la permission ne peut être accordée à la première application.
  - Le système attribue le niveau de protection "signature" aux permissions de signature, comme indiqué sur la page de référence de l'API des permissions.

Permissions d'exécution

Les permissions d'exécution, également appelées permissions dangereuses, donnent à l'application un accès supplémentaire à des données restreintes et lui permettent d'effectuer des actions restreintes qui ont un impact plus important sur le système et les autres applications. Par conséquent, il est nécessaire de demander des autorisations d'exécution dans l'application avant de pouvoir accéder aux données restreintes ou d'effectuer des actions restreintes. Lorsque l'application demande une autorisation d'exécution, le système affiche une invite d'autorisation d'exécution, comme le montre la Figure.

De nombreuses autorisations d'exécution permettent d'accéder à des données utilisateur privées, un type particulier de données restreintes comprenant des informations potentiellement sensibles. Les informations de localisation et de contact sont des exemples de données utilisateur privées.

Le microphone et la caméra donnent accès à des informations particulièrement sensibles. Par conséquent, le système aide à expliquer pourquoi l'application accède à ces informations.

Le système attribue le niveau de protection "dangereux" aux permissions d'exécution, comme indiqué sur la page de référence de l'API des permissions.

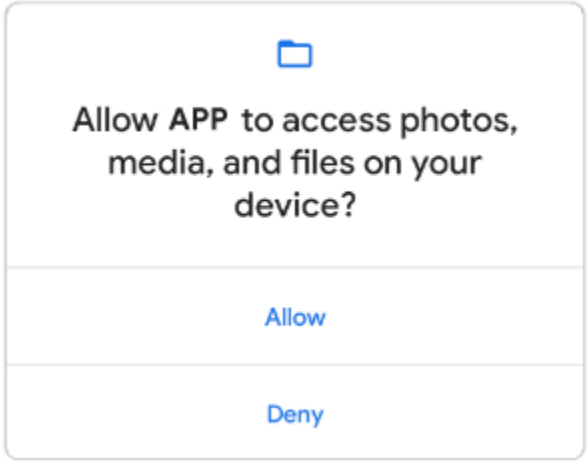


Figure : L'invite de permission du système qui apparaît lorsque l'application demande une permission d'exécution.

PARTIE 3

Permissions spéciales

Les permissions spéciales correspondent à des opérations particulières de l'application. Seuls la plateforme et les OEMs (fabricant d'équipements originaux) peuvent définir des permissions spéciales. En outre, la plateforme et les OEM définissent généralement des autorisations spéciales lorsqu'ils souhaitent protéger l'accès à des actions particulièrement puissantes, telles que le dessin sur d'autres apps.

La page Accès spécial aux applications dans les paramètres du système contient un ensemble d'opérations que l'utilisateur peut activer ou désactiver. Un grand nombre de ces opérations sont mises en œuvre en tant que permissions spéciales.

Chaque autorisation spéciale a ses propres détails de mise en œuvre. Les instructions d'utilisation de chaque permission spéciale figurent sur la page de référence de l'API des permissions. Le système attribue le niveau de protection "**appop**" aux permissions spéciales.

Bonnes pratiques

Les permissions d'applications s'appuient sur les fonctions de sécurité du système et aident Android à atteindre les objectifs suivants en matière de confidentialité des utilisateurs :

- **Le contrôle** : l'utilisateur a le contrôle sur les données qu'il partage avec les apps
- **Transparence** : l'utilisateur comprend quelles sont les données utilisées par une application et pourquoi l'application accède à ces données
- **Minimisation des données** : une application n'accède et n'utilise que les données nécessaires à une tâche ou une action spécifique que l'utilisateur invoque

1. Demander un nombre minimal de permissions :

Lorsque l'utilisateur demande une action particulière dans l'application, celle-ci ne doit demander que les autorisations dont elle a besoin pour réaliser cette action. En fonction de l'utilisation des permissions, il peut exister un autre moyen de répondre au cas d'utilisation de l'application sans avoir à accéder à des informations sensibles.

Bonnes pratiques

2. Associer les autorisations d'exécution à des actions spécifiques

Demander des autorisations aussi tard que possible dans le flux des cas d'utilisation de l'application. Par exemple, si l'application permet aux utilisateurs d'envoyer des messages audio à d'autres personnes, il convient d'attendre que l'utilisateur se rende sur l'écran de messagerie et appuie sur le bouton Envoyer un message audio. Une fois que l'utilisateur a appuyé sur le bouton, l'application peut alors demander l'accès au microphone.

3. Tenir compte des dépendances de l'application

Lors de l'inclusion d'une bibliothèque, les permissions requises pour celle-ci sont également héritées. Vérifier les autorisations requises par chaque dépendance et l'usage qui en est fait.

4. Être transparent

Lors d'une demande d'autorisation, expliquer clairement à quoi vous accédez et pourquoi, afin que les utilisateurs puissent prendre des décisions éclairées.

5. Rendre les accès au système explicites

Si le système ne fournit pas déjà des indications sur l'accès aux données ou au matériel sensibles, tels que la caméra ou le microphone, l'application doit fournir une indication continue. Ce rappel aide les utilisateurs à comprendre exactement quand l'application accède à des données restreintes ou effectue des actions restreintes.

CHAPITRE 2

Demander une permission

Ce que vous allez apprendre dans ce chapitre :

- Utilisation des Workflow pour la demande de permissions
- Utilisation des permissions de localisation
- Présentation de la permission d’accès au réseau
- Présentation de la permission d’accès aux contacts du téléphone
- Présentation de la permission d’accès au Bluetooth

6 heures

CHAPITRE 2

Demander une permission

1. **Utilisation des Workflow pour la demande de permissions**
2. Utilisation des permissions de localisation
3. Présentation de la permission d’accès au réseau
4. Présentation de la permission d’accès aux contacts du téléphone
5. Présentation de la permission d’accès au Bluetooth

Principes de base

Les principes de base pour demander des permissions au moment de l'exécution sont les suivants :

- Demander des permissions dans le contexte, lorsque l'utilisateur commence à interagir avec la fonctionnalité qui le nécessite.
- Ne pas bloquer l'utilisateur. Donner toujours la possibilité d'annuler un flux d'interface utilisateur conviviale lié aux permissions.
- Si l'utilisateur refuse ou révoque une autorisation dont une fonction a besoin, faites en sorte que l'utilisateur puisse continuer à utiliser l'application, éventuellement en désactivant la fonction qui nécessite l'autorisation.
- Ne faites pas d'hypothèses sur le comportement du système. Par exemple, ne présupposer pas que les permissions apparaissent dans le même groupe de permissions, Un groupe d'autorisations aide simplement le système à minimiser le nombre de boîtes de dialogue présentées à l'utilisateur lorsqu'une application demande des autorisations étroitement liées.

Workflow pour la demande de permissions (1/2)

Avant de déclarer et de demander des permissions d'exécution dans l'application, il convient d'évaluer si celle-ci en a besoin. Il est possible de réaliser de nombreux cas d'utilisation dans l'application, tels que la prise de photos, la mise en pause de la lecture multimédia et l'affichage de publicités pertinentes, sans avoir à déclarer de permissions.

Si l'application doit déclarer et demander des autorisations d'exécution, procéder comme suit :

1. Dans le fichier manifeste de l'application, déclarer les autorisations que l'application pourrait devoir demander.
2. Concevoir l'interface utilisateur de l'application de manière à ce que des actions spécifiques soient associées à des autorisations d'exécution spécifiques. Les utilisateurs doivent savoir quelles actions peuvent nécessiter qu'ils accordent l'autorisation à l'application d'accéder aux données privées des utilisateurs.
3. Attendre que l'utilisateur invoque la tâche ou l'action de l'application qui nécessite l'accès à des données privées spécifiques. À ce moment-là, l'application peut demander la permission d'exécution nécessaire pour accéder à ces données.
4. Vérifier si l'utilisateur a déjà accordé l'autorisation d'exécution requise par l'application. Si c'est le cas, l'application peut accéder aux données privées de l'utilisateur. Dans le cas contraire, passer à l'étape suivante.

Il est nécessaire de vérifier si l'on dispose de cette permission chaque fois que l'on effectue une opération qui requiert cette permission.

Workflow pour la demande de permissions (2/2)

5. Vérifier si l'application doit montrer une justification à l'utilisateur, expliquant pourquoi l'application a besoin que l'utilisateur accorde une autorisation d'exécution particulière. Si le système détermine que l'application ne doit pas afficher de justification, passer directement à l'étape suivante, sans afficher d'élément d'interface utilisateur.

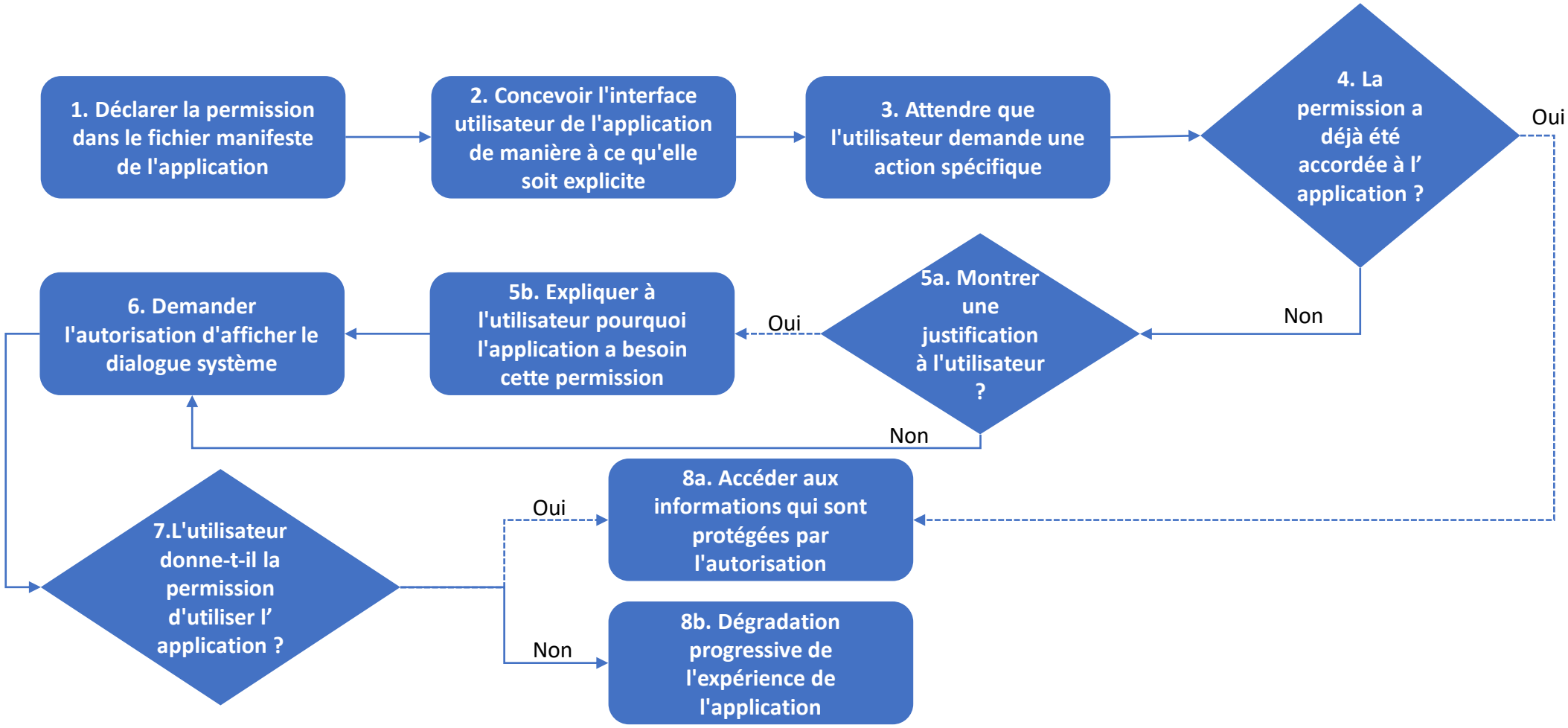
   Si le système détermine que l'application doit afficher une justification, présenter-la à l'utilisateur dans un élément d'interface utilisateur. Cette justification doit expliquer clairement les données auxquelles l'application tente d'accéder et les avantages que l'application peut apporter à l'utilisateur s'il accorde l'autorisation d'exécution. Une fois que l'utilisateur a pris connaissance du raisonnement, passer à l'étape suivante.

6. Demander la permission d'exécution dont l'application a besoin pour accéder aux données privées de l'utilisateur. Le système affiche une demande d'autorisation d'exécution, comme celle qui figure sur la page de présentation des autorisations.;
7. Vérifier la réponse de l'utilisateur, qu'il ait choisi d'accorder ou de refuser l'autorisation d'exécution ;
8. Si l'utilisateur a accordé l'autorisation à l'application, celle-ci peut accéder aux données privées de l'utilisateur. Si, au contraire, l'utilisateur a refusé l'autorisation, il faut procéder à une transition progressive de l'expérience de l'application afin qu'elle offre des fonctionnalités à l'utilisateur, même sans les informations protégées par cette permission.

La figure présentée dans la diapositive suivante illustre le workflow et l'ensemble des décisions associées à ce processus.

PARTIE 3

Workflow



Déterminer si l'utilisateur a déjà accordé la permission à l'application

Pour vérifier si l'utilisateur a déjà accordé à l'application une permission particulière, il suffit de passer cette permission à la méthode **ContextCompat.checkSelfPermission()**. Cette méthode renvoie soit **PERMISSION_GRANTED**, soit **PERMISSION_DENIED**, selon que l'application dispose ou non de cette permission.

Si la méthode **ContextCompat.checkSelfPermission()** renvoie **PERMISSION_DENIED**, appeler **shouldShowRequestPermissionRationale().** Si cette méthode renvoie **true**, l'utilisateur reçoit une interface utilisateur conviviale. Dans cette interface, l'utilisateur doit expliquer pourquoi la fonction qu'il souhaite activer nécessite une autorisation particulière.

De plus, si l'application demande une autorisation liée à la localisation, au microphone ou à l'appareil photo, penser à expliquer la raison pour laquelle l'application doit avoir accès à ces informations.

Expliquer pourquoi l'application a besoin de la permission

La boîte de dialogue des permissions affichée par le système lors de l'appel de **requestPermissions()** indique la permission que l'application souhaite obtenir, mais ne dit pas pourquoi. Dans certains cas, l'utilisateur peut trouver cela déroutant. C'est une bonne idée d'expliquer à l'utilisateur pourquoi l'application veut les permissions avant d'appeler **requestPermissions().**

Les recherches montrent que les utilisateurs sont beaucoup plus à l'aise avec les demandes d'autorisation s'ils savent pourquoi l'application en a besoin.

Demander des permissions

Les étapes suivantes montrent comment utiliser le contrat **RequestPermission**. Le processus est pratiquement le même pour le contrat **RequestMultiplePermissions**.

1. Dans la logique d'initialisation de l'activité ou du Fragment, passer une implémentation de **ActivityResultCallback** dans un appel à **registerForActivityResult(). L'ActivityResultCallback** définit la façon dont l'application traite la réponse de l'utilisateur à la demande de permission.

   Conserver une référence à la valeur de retour de **registerForActivityResult(),** qui est de type **ActivityResultLauncher**.

2. Pour afficher la boîte de dialogue des autorisations du système lorsque cela est nécessaire, appeler la méthode **launch()** sur l'instance de **ActivityResultLauncher** déjà enregistrée dans l'étape précédente.

   Après l'appel de la méthode **launch(),** la boîte de dialogue des autorisations du système apparaît. Lorsque l'utilisateur fait un choix, le système invoque de manière asynchrone l'implémentation de **ActivityResultCallback** déjà définie à l'étape précédente.

Demander des permissions

L'extrait de code suivant montre comment traiter la réponse relative aux permissions :

```
// Enregistrer le callback des permissions, qui gère la réponse de l'utilisateur fournie dans la boite de dialogue des permissions du système.
// Enregistrer la valeur de retour, une instance de
// ActivityResultLauncher, comme une variable d'instance.
private ActivityResultLauncher<String> requestPermissionLauncher =
    registerForActivityResult(new RequestPermission(), isGranted -> {
        if (isGranted) {
            // La permission est accordée. Poursuivre l'action ou le flux de travail dans l'application.
        } else {
            // Expliquer à l'utilisateur que la fonction n'est pas disponible car elle nécessite une autorisation qu'il a refusée.
            // La fonctionnalité nécessite une autorisation que l'utilisateur a refusée. En même temps, respecter la décision de l'utilisateur.
            // Ne créer pas de lien vers les paramètres du système dans le but de convaincre l'utilisateur de changer sa décision.
        }
    }) ;
```

Demander des permissions

Cet extrait de code illustre le processus recommandé pour vérifier l'existence d'une autorisation et demander une autorisation à l'utilisateur si nécessaire :

```
si (ContextCompat.checkSelfPermission(
        CONTEXT, Manifest.permission.REQUESTED_PERMISSION) ==
        PackageManager.PERMISSION_GRANTED) {
    // Utiliser l'API qui requiert la permission.
    performAction(...) ;
} else if (shouldShowRequestPermissionRationale(...)) {
    // Dans une interface utilisateur conviviale, expliquer à l'utilisateur pourquoi l’application requiert cette
    // permission pour qu'une fonction spécifique se comporte comme prévu. Dans cette interface utilisateur,
    // inclure un bouton "annuler" ou "non merci" qui permet à l'utilisateur de continuer à utiliser l’application sans accorder la permission.
    showInContextUI(...) ;
} else {
    // Demander directement l'autorisation.
    // ActivityResultCallback enregistré reçoit le résultat de cette demande.
    requestPermissionLauncher.launch(
          Manifest.permission.REQUESTED_PERMISSION) ;
}
```

Gérer soi-même le code de demande d'autorisation

Au lieu de laisser le système gérer le code de demande de permission, il est possible de gérer le code de demande de permission soi-même. Pour ce faire, il suffit d'inclure le code de demande dans un appel à **requestPermissions().** L'extrait de code suivant montre comment demander une autorisation à l'aide d'un code de demande :

```
si (ContextCompat.checkSelfPermission(
        CONTEXT, Manifest.permission.REQUESTED_PERMISSION) ==
        PackageManager.PERMISSION_GRANTED) {
    // Utiliser l'API qui requiert la permission.
    performAction(...) ;
} else if (shouldShowRequestPermissionRationale(...)) {
    // Dans une interface utilisateur conviviale, expliquer à l'utilisateur pourquoi l’application requiert cette
    // permission pour qu'une fonction spécifique se comporte comme prévu. Dans cette interface utilisateur,
    // inclure un bouton "annuler" ou "non merci" qui permet à l'utilisateur de continuer à utiliser l’application sans accorder la permission.
    showInContextUI(...) ;
} else {
    // Demander directement l'autorisation.
    requestPermissions(CONTEXT,
            new String[] { Manifest.permission.REQUESTED_PERMISSION },
            REQUEST_CODE) ;
}
```

Gérer soi-même le code de demande d'autorisation

Une fois que l'utilisateur a répondu à la boîte de dialogue d'autorisation du système, ce dernier invoque l'implémentation de la fonction **onRequestPermissionsResult()** de l'application. Le système transmet la réponse de l'utilisateur à la boîte de dialogue d'autorisation, ainsi que le code de demande défini, comme le montre l'extrait de code suivant :

```
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions,
        int[] grantResults) {
    switch (requestCode) {
        case PERMISSION_REQUEST_CODE :
            // Si la demande est annulée, les tableaux de résultats sont vides.
            si (grantResults.length > 0 &&
                    grantResults[0] == PackageManager.PERMISSION_GRANTED) {
                // L'autorisation est accordée. Continuer l'action ou le flux de travail
                // dans l'application.
            } else {
                // Expliquer à l'utilisateur que la fonction est indisponible parce que
                // la fonction requiert une autorisation que l'utilisateur a refusée.
                // Dans le même temps, respecter la décision de l'utilisateur.
                // Ne créer pas de lien vers les paramètres du système dans
                // le but de convaincre l'utilisateur de changer sa décision.
            }
            retour ;
        }
        // D'autres lignes 'case' pour vérifier d'autres
        // autorisations que cette application pourrait demander.
    }
}
```

Demande de permissions multiples

Lors de la demande d'une permission de localisation, il faut suivre les mêmes bonnes pratiques que pour toute autre permission d'exécution. Une différence importante en ce qui concerne les permissions de localisation est que le système comprend plusieurs permissions liées à la localisation. Les permissions requises et la manière dont elles sont demandées dépendent des exigences de localisation pour le cas d'utilisation de l'application.

Localisation au premier plan

Si l'application contient une fonction qui ne partage ou ne reçoit des informations de localisation qu'une seule fois, ou pour une durée déterminée, cette fonction nécessite un accès à la localisation au premier plan. Voici quelques exemples :

- Dans une application de navigation, une fonctionnalité permet aux utilisateurs d'obtenir un itinéraire virage par virage.
- Dans une application de messagerie, une fonction permet aux utilisateurs de partager leur position actuelle avec un autre utilisateur.

Le système considère que l'application utilise la localisation au premier plan si une fonction de l'application accède à la localisation actuelle de l'appareil dans l'une des situations suivantes :

- Une activité appartenant à l'application est visible.
- L'application exécute un service de premier plan. Lorsqu'un service de premier plan est en cours d'exécution, le système sensibilise l'utilisateur en affichant une notification persistante. L'accès à l'application est maintenu lorsque celle-ci est placée en arrière-plan, par exemple lorsque l'utilisateur appuie sur le bouton Home de son appareil ou éteint l'écran de son appareil.

En outre, il est recommandé de déclarer un type de service d'avant-plan pour la localisation, comme le montre l'extrait de code suivant. Sur Android 10 (niveau 29 de l'API) et les versions ultérieures, ce type de service d'avant-plan doit être déclaré.

```
<!-- Recommandé pour Android 9 (API niveau 28) et inférieur. -->
<!-- Requis pour Android 10 (niveau 29 de l'API) et supérieur. -->
<service
    android:name="MonNavigationService"
    android:foregroundServiceType="location" ... >
<!-- Tout élément interne devrait être placé ici. -->
</service>
```

PARTIE 3

Demande de permissions multiples

L'application déclare un besoin de localisation au premier plan lorsqu'elle demande l'autorisation **ACCESS_COARSE_LOCATION** ou **ACCESS_FINE_LOCATION**, comme le montre l'extrait suivant :

```
<manifeste ... >
  <!-- Toujours inclure cette permission -->
  <uses-permission android:name="android.permission.ACCESS_COARSE_LOCATION" />

  <!-- N'inclure que si l'application bénéficie d'un accès à une localisation précise. -->
  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />
</manifest>
```

Demande de permissions multiples

Localisation en arrière-plan

Une application nécessite un accès à la localisation en arrière-plan si une fonctionnalité de l'application partage constamment la localisation avec d'autres utilisateurs ou utilise l'API Geofencing. Voici quelques exemples :

- Dans une application de partage de localisation familiale, une fonctionnalité permet aux utilisateurs de partager en permanence la localisation avec les membres de la famille.
- Dans une application IoT, une fonctionnalité permet aux utilisateurs de configurer leurs appareils domestiques de manière à ce qu'ils s'éteignent lorsque l'utilisateur quitte son domicile et se rallument lorsque l'utilisateur rentre chez lui.

Le système considère que l'application utilise la localisation en arrière-plan si elle accède à la localisation actuelle de l'appareil dans toute autre situation que celles décrites dans la section sur la localisation en avant-plan. La précision de la localisation en arrière-plan est la même que celle de la localisation au premier plan, qui dépend des autorisations de localisation que l'application déclare.

Sur Android 10 (niveau 29 de l'API) et les versions ultérieures, il est nécessaire de déclarer l'autorisation **ACCESS_BACKGROUND_LOCATION** dans le manifeste de l'application afin de demander l'accès à l'emplacement en arrière-plan au moment de l'exécution. Sur les versions antérieures d'Android, lorsque l'application reçoit un accès à l'emplacement en avant-plan, elle reçoit automatiquement un accès à l'emplacement en arrière-plan.

```
<manifeste ... >
  <!-- Requis uniquement lors de la demande d'accès à la localisation en arrière-plan sur
       Android 10 (niveau 29 de l'API) et plus. -->
  <uses-permission
android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
</manifest>
```

CHAPITRE 2

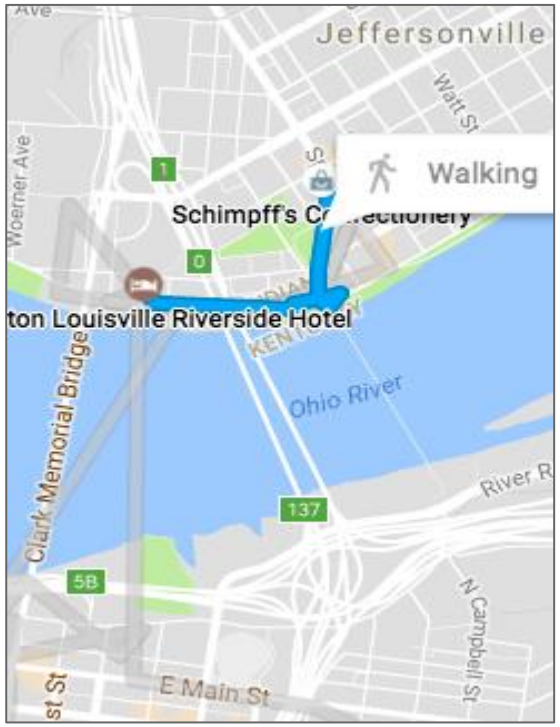
Demander une permission

1. Utilisation des Workflow pour la demande de permissions
2. **Utilisation des permissions de localisation**
3. Présentation de la permission d’accès au réseau
4. Présentation de la permission d’accès aux contacts du téléphone
5. Présentation de la permission d’accès au Bluetooth

PARTIE 3

Introduction

- Téléphones mobiles -> le mot clé est **MOBILE**
- Les utilisateurs se déplacent et vont partout
- L'application peut détecter et utiliser l'emplacement de l'appareil pour personnaliser l'expérience de l'utilisateur



Utiliser la localisation dans l'application

- Vérifier si la permission de localisation a été accordée
- Demander l'autorisation si nécessaire
- Demander la localisation la plus récente
- Demander les mises à jour de la localisation

Obtenir l'emplacement de l'appareil

- Utilise **FusedLocationProviderClient** ;
- Effectue des demandes de localisation combinant GPS, Wi-Fi et réseau cellulaire
- Équilibre entre des résultats rapides et précis et une consommation minimale de la batterie
- Renvoie un objet de localisation avec la latitude et la longitude

- Utilise le géocodeur pour convertir l'emplacement lat/long en adresse physique.

Latitude 33.5219
Longitude 7.6350

50, Rue Caporal Driss Chbakou Ain Borja, 20 300, Casablanca

Configuration des services Google Play

Les services de localisation sont fournis par Google Play Services.

Installer le dépôt Google dans Android Studio :

1. Sélectionner **Outils > Android > SDK Manager ;**
2. Sélectionner l'onglet **SDK Tools ;**
3. Développer **Support Repository ;**
4. Sélectionner **Google Repository** et cliquer sur **OK.**

Ajout de Google Play au projet

Ajouter aux dépendances dans build.gradle (Module : app) :

```
implementation 'com.google.android.gms:play-services-location:x.x.x’
```

xx.x.x est le numéro de version, tel que 11.0.2.

Remplacer par le nouveau numéro de version, si Android Studio le propose.

Demande de permission de localisation

Les applications doivent demander l'autorisation de localisation

- **ACCESS_COARSE_LOCATION** pour une localisation précise à un pâté de maisons près
- **ACCESS_FINE_LOCATION** pour obtenir une localisation précise

Demande de permission dans le manifeste :

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

Demande de permission au moment de l'exécution :

- L'utilisateur peut révoquer l'autorisation à tout moment
- Vérifier la permission chaque fois que l'application utilise la localisation
- Détails et exemples : demande de permission au moment de l'exécution

Étapes pour vérifier/demander une permission

- Utiliser **checkSelfPermission()** pour voir si la permission est accordée
- Utiliser **requestPermissions()** pour demander l'autorisation
- Vérifier la réponse de l'utilisateur pour savoir si la demande a été accordée

Vérifier/demander la permission :

```
 if (ActivityCompat.checkSelfPermission(this,
             Manifest.permission.ACCESS_FINE_LOCATION) !=
                       PackageManager.PERMISSION_GRANTED) {
      ActivityCompat.requestPermissions(this,
         new
String[]{Manifest.permission.ACCESS_FINE_LOCATION},
         REQUEST_LOCATION_PERMISSION);

   } else {

      Log.d(TAG, "getLocation: permissions accordée");

   }
```

Obtenir la réponse de l'utilisateur

Obtenir **onRequestPermissionsResult()** pour vérifier si l'utilisateur a accordé la permission :

```
@Override
public void onRequestPermissionsResult(int requestCode,
          String permissions[], int[] grantResults) {
             switch (requestCode) {
                case REQUEST_LOCATION_PERMISSION: {
                 // Vérifier si la permission est accordée.
```

Vérifier si la demande a été accordée

La réponse est renvoyée dans le tableau des permissions

Comparer le paramètre **grantResults** à PackageManager.PERMISSION_GRANTED.

```
// Vérifier si l'autorisation est accordée.
if (grantResults.length > 0) && grantResults[0] ==
                    PackageManager.PERMISSION_GRANTED) {
          // Permission est accordée.
          ...
    } else {// Permission est refusée...
```

02 – Demander une permission

Utilisation des permissions de localisation

PARTIE 3

Obtenir la position de l'appareil

- Utiliser **FusedLocationProviderClient** pour demander le dernier emplacement connu
- En général, la dernière position connue est la même que la position actuelle

Obtenir FusedLocationProviderClient :

Pour obtenir FusedLocationProviderClient :

```
FusedLocationProviderClient flpClient =
        LocationServices.getFusedLocationProviderClient(
            context);
```

Demande de la dernière localisation connue:

- Appeler le client FusedLocationProviderClient getLastLocation()
- Retourne l'objet **Task** représentant la tâche asynchrone de récupération de l'objet Location
- La **Task** fournit des méthodes pour ajouter des écouteurs de succès et d'échec
- Récupération de la latitude et de la longitude à partir de l'objet Location

getLastLocation() écouteur de succès

```
mFusedLocationClient.getLastLocation().addOnSuccessListener
(
        new OnSuccessListener<Location>() {
            @Override
            public void onSuccess(Location location) {
                if (location != null) {
                    mLastLocation = location;
                    // Obtenir la lat et long.
                } else { // Show "no location" }
        }
});
```

getLastLocation() écouteur d'échec

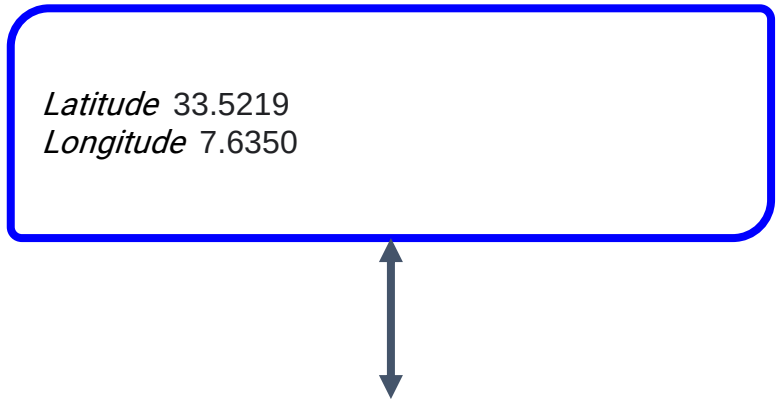
```
mFusedLocationClient.getLastLocation().addOnFailureListener
(
        new OnFailureListener() {
            @Override
            public void onFailure(@NonNull Exception e) {
             Log.e(TAG, "onFailure: ", e.printStackTrace());
            }
        }
);
```

Obtenir la latitude et la longitude

```
public void onSuccess(Location location) {

      if (location != null) {

            // Obtenir la lat et long.

            lat = location.getLatitude(),

            long = location.getLongitude(),

            time = location.getTime()));

       } else { // aucune localisation}
```

Géocodage et géocodage inverse

- Géocoder : convertir une adresse de rue lisible par l'homme en latitude/longitude
- Géocodage inverse : convertir la latitude/longitude en adresse routière lisible par l'homme



Utiliser la classe Geocoder

- Utiliser **Geocoder** pour le géocodage et le géocodage inverse :

Geocoder geocoder = new Geocoder(context,

Locale.getDefault()) ;

- Les méthodes font une requête réseau - ne pas appeler sur le Thread principal

Service backend Geocoder

- Le **Geocoder** nécessite un service de backend qui n'est pas inclus dans le framework Android de base
- Utiliser **isPresent()** pour vérifier si l'implémentation existe
- Les méthodes d'interrogation du **Geocoder** renvoient une liste vide si aucun service de base n'existe

Inversion des coordonnées de géocodage

getFromLocation(
double latitude, double longitude, int maxResults)

Renvoie la liste des objets Address :

```
List<Address> addresses = geocoder.getFromLocation(
    location.getLatitude(), location.getLongitude(), 1);
```

Géocodage de l'adresse en coordonnées :

getFromLocationName(
String locationName, int maxResults)

Renvoie la liste des objets Address avec les coordonnées de latitude et de longitude :

```
List<Address> addresses = geocoder.getFromLocationName(
    "50, Rue Caporal Driss Chbakou Ain Borja, 20 300,
Casablanca", 1)
Address firstAddress = addresses.get(0);
double latitude = firstAddress.getLatitude();
double longitude = firstAddress.getLongitude();
```

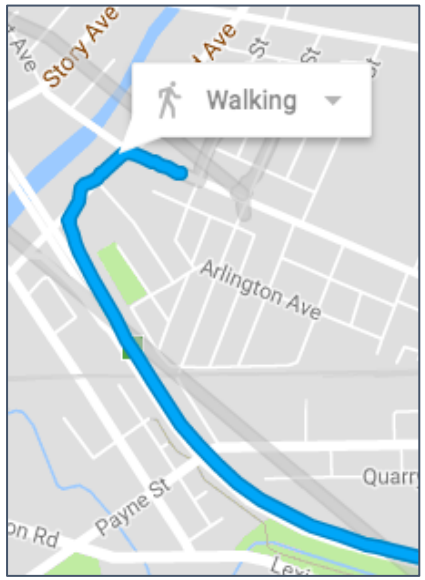
PARTIE 3

PARTIE 3

Création d'un objet LocationRequest

Obtenir des mises à jour de la localisation :

- L'application peut obtenir le dernier emplacement connu
- Elle peut également demander des mises à jour régulières pour suivre l'emplacement
- Utiliser **LocationRequest** pour définir les paramètres des demandes de mise à jour de la position



Paramètres de LocationRequest

Définir les paramètres de **LocationRequest** pour contrôler les demandes de localisation

- **setInterval()** : définit la fréquence à laquelle l'application doit être mise à jour
- **setFastestInterval()** : fixe une limite au taux de mise à jour pour éviter le scintillement et le débordement des données
- **setPriority()** : définit la priorité et les sources des requêtes

Demander des valeurs de priorité

PRIORITY_BALANCED_POWER_ACCURACY	Précise à un bloc de ville près (100 mètres) ; utilise uniquement le Wi-Fi et le réseau cellulaire, pour consommer moins d'énergie.
PRIORITY_HIGH_ACCURACY	Utilise le GPS s'il est disponible
PRIORITY_LOW_POWER	Précision à l'échelle d'une ville (10 km)
PRIORITY_NO_POWER	Mise à jour lorsqu'elle est déclenchée par d'autres applications (aucune consommation supplémentaire).

Créer un exemple de LocationRequest

```
private LocationRequest getLocationRequest() {

    LocationRequest locationRequest = new
LocationRequest();

    locationRequest.setInterval(10000);

    locationRequest.setFastestInterval(5000);

    locationRequest.setPriority(

        LocationRequest.PRIORITY_HIGH_ACCURACY);

   return locationRequest;

}
```

Requesting location updates

- Utiliser **LocationRequest** avec **FusedLocationProviderClient** ;
- Précision de la localisation déterminée par :
  - Les fournisseurs de localisation disponibles (réseau et GPS) ;
  - La permission de localisation demandée ;
  - Options définies dans la demande de localisation.

Étapes pour lancer les mises à jour de localisation

1. Créer l'objet **LocationRequest**
2. Surcharger **LocationCallback.onLocationResult()**
3. Utiliser **requestLocationUpdates**() sur **FusedLocationProviderClient** pour lancer des mises à jour régulières

Utiliser **requestLocationUpdates()** pour lancer des mises à jour régulières :

- Transmettre **LocationRequest** et **LocationCallback**
- Les mises à jour de la localisation sont transmises à **onLocationResult()**

Utiliser LocationCallback

```
mLocationCallback = new LocationCallback() {
  @Override
  public void onLocationResult(
                        LocationResult locationResult) {
  for (Location location : locationResult.getLocations()) {
        // Mise à jour des IU avec les données de
        // localisation
        // ...
    }
};
```

Travailler avec les paramètres de l'utilisateur

Étapes à suivre pour vérifier les paramètres de l'appareil :

1. Créer un objet **LocationSettingsRequest.**
2. Créer un objet **SettingsClient** .
3. Utiliser **checkLocationSettings()** pour vérifier si les paramètres du dispositif correspondent à **LocationRequest.**
4. Utiliser **OnFailureListener** pour savoir si les paramètres ne correspondent pas à **LocationRequest**.

1. Créer un LocationSettingsRequest

Créer un objet **LocationSettingsRequest** et ajouter une ou plusieurs demandes de localisation :

```
LocationSettingsRequest settingsRequest =

  nouvel objet LocationSettingsRequest.Builder()

   .addLocationRequest(mLocationRequest).build() ;
```

2. Créer un SettingsClient

Créer un objet **SettingsClient** en utilisant **LocationServices.getSettingsClient()** et transmettre le contexte :

```
SettingsClient client =

   LocationServices.getSettingsClient(this) ;
```

3. Vérifier si les paramètres de l'appareil correspondent à la requête

- Utiliser **checkLocationSettings()** pour voir si les paramètres du dispositif correspondent à **LocationRequest** :

  Task<LocationSettingsResponse> task =

  client.checkLocationSettings(settingsRequest) ;

- Renvoie un objet **Task**.

4. Intercepter lorsque les paramètres ne correspondent pas

- Utiliser le générateur **OnFailureListener** pour détecter les cas où les paramètres ne correspondent pas à **LocationRequest**
- L'exception passée dans **onFailure**() contient le statut
- Si les paramètres ne correspondent pas, le statut est **LocationSettingsStatusCodes.RESOLUTION_REQUIRED**
- Afficher une boîte de dialogue pour modifier les paramètres

Exemple de code pour le dialogue utilisateur

```
task.addOnFailureListener(this, new OnFailureListener() {
   @Override
   public void onFailure(@NonNull Exception e) {
      int statusCode = ((ApiException) e).getStatusCode();
      if (statusCode ==
           LocationSettingsStatusCodes.RESOLUTION_REQUIRED)
      {
         // Afficher un dialogue à l'utilisateur.
         // ...
      }
   }
});

// Afficher un dialogue à l'utilisateur.
try {
     // Appeler startResolutionForResult(), vérifier
     // résultat dans onActivityResult()
     ResolvableApiException resolvable =
                                (ResolvableApiException) e;
     resolvable.startResolutionForResult
               (MainActivity.this,REQUEST_CHECK_SETTINGS);
     } catch (IntentSender.SendIntentException sendEx) {
               // Ignorer l'erreur
     }
```

Traiter la décision de l'utilisateur

- Surcharger **onActivityResult()** dans **Activity** ;
- S'assurer que le requestCode correspond à la constante dans **`startResolutionForResult()`.**

Exemple de méthode **onActivityResult** :

```
@Override
protected void onActivityResult(int requestCode,
                                int resultCode, Intent data) {
   if (requestCode == REQUEST_CHECK_SETTINGS) {
      if (resultCode == RESULT_CANCELED) {
         stopTrackingLocation();
       } else if (resultCode == RESULT_OK) {
         startTrackingLocation();
       }
   }
   super.onActivityResult(requestCode, resultCode, data);
}
```

PARTIE 3

APIs de localisation

On trouve deux API qui sont liées au concept de localisation :

- **Une API qui permet de localiser l'appareil :**

Le **GPS** est la solution la plus efficace pour localiser un appareil, cependant il s'agit aussi de la plus coûteuse en batterie. Une autre solution courante est de se localiser à l'aide des points d'accès WiFi à proximité et de la distance mesurée avec les antennes relais du réseau mobile les plus proches

- **Une API qui permet d'afficher des cartes :**

l'API pour **Google Maps** n'est pas intégrée à Android, mais appartient à une extension appelée « **Google APIs** », les librairies de cette API doivent être intégrées au code source de l'application pour utiliser les cartes Google Maps.

Localiser l'appareil

- **Permission :**

Tout d'abord, il faut demander la permission dans le Manifest pour utiliser les fonctionnalités de localisation.

- Pour utilisation du **GPS**, il faut inclure la permission suivante :

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

- Pour une localisation plus imprécise par **WiFi** et **antennes relais** :

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

- Ensuite, faire appel à un nouveau service système pour accéder à ces fonctionnalités : **LocationManager**, que l'on récupère de cette manière :

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

Obtenir des notifications du fournisseur

Pour obtenir la dernière position connue de l'appareil, utilisez Location **getLastKnownLocation(String provider)**.

Si on veut faire en sorte que le fournisseur se mette à jour automatiquement à une certaine période ou tous les x mètres, on peut utiliser la méthode suivante :

```
locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 600
00, 150, new LocationListener() {
  @Override
  public void onStatusChanged(String provider, int status, Bundle extras) {
  }
  @Override
  public void onProviderEnabled(String provider) {
  }
  @Override
  public void onProviderDisabled(String provider) {
  }
  @Override
  public void onLocationChanged(Location location) {
    Log.d("GPS", "Latitude " + location.getLatitude() + " et longitude " + location.ge
tLongitude());
  }
});
```

requestLocationUpdates

Cette méthode possède 4 arguments :

- Le provider utilisé pour recevoir les mises à jour des coordonnées utilisateurs (GPS / NETWORK …) ;
- Un intervale minimum entre deux notifications (en millisecondes) ;
- Un intervale minimum entre deux notifications (en mètres) ;
- L'instance de votre **LocationListener**.

- **OnProviderEnabled(String provider)** : cette méthode est appelée quand une source de localisation est activée (GPS, 3G..etc).

```
String msg =
String.format(getResources().getString(R.string.provider_enabled), provider);
Toast.makeText(getApplicationContext(), msg, Toast.LENGTH_SHORT).show();
```

```
<string name="provider_enabled">The provider %s is now enabled</string>
```

- **OnProviderDisabled(String provider)** : cette méthode est appelée quand une source de localisation est désactivée (GPS, 3G..etc). L'argument est le nom de la source désactivée.

```
String msg = String.format(getResources().getString(R.string.provider_enabled),
provider);
Toast.makeText(getApplicationContext(), msg, Toast.LENGTH_SHORT).show();
```

```
<string name="provider_disabled">The provider %s is now disabled</string>
```

requestLocationUpdates

- **OnStatusChanged(String provider, int status, Bundle extras)** : appeler quand le statut d'une source change. Il existe 3 statuts (**OUT_OF_SERVICE**, **TEMPORARILY_UNAVAILABLE, AVAILABLE**).

```
String newStatus = "";
 switch (status) {
    case LocationProvider.OUT_OF_SERVICE:
       newStatus = "OUT_OF_SERVICE";
       break;
    case LocationProvider.TEMPORARILY_UNAVAILABLE:
       newStatus = "TEMPORARILY_UNAVAILABLE";
       break;
    case LocationProvider.AVAILABLE:
       newStatus = "AVAILABLE";
       break;
 }
String msg =
String.format(getResources().getString(R.string.provider_new_status),    provider,
newStatus);

Toast.makeText(getApplicationContext(), msg, Toast.LENGTH_SHORT).show();
```

<string name="provider_new_status">The provider %1$s has now a new status %2$s</string>

- **OnLocationChanged(Location location)** : cette méthode est appelée quand la localisation de l'utilisateur est mise à jour. L'argument représente la nouvelle position. Vous pouvez récupérer plusieurs informations comme la latitude, longitude, altitude et précision (en mètre).

```
latitude = location.getLatitude();
longitude = location.getLongitude();
altitude = location.getAltitude();
accuracy = location.getAccuracy();

String msg = String.format(
      getResources().getString(R.string.new_location), latitude,
      longitude, altitude, accuracy);
Toast.makeText(getApplicationContext(), msg,
            Toast.LENGTH_LONG).show();
```

<string name="new_location">New Location : Latitude = %1$s, Longitude = %2$s, Altitude = %3$s avec une précision de %4$s mètres </string>

Afficher des cartes

- Les éléments de l'API :
  - La fonction **MapFragment** qui vient de faciliter l'affichage d'une carte sur toutes les tailles d'écrans des appareils Android
  - Les données vectorisées qui offrent deux avantages : un chargement et un téléchargement plus rapides
  - De nouvelles vues 3D, indoor, trafic, qui offrent un affichage de la vue en 3D (légèrement inclinée)
  - Des marqueurs beaucoup plus simples à créer
  - Un cache amélioré et beaucoup d'autres fonctions

Afficher des cartes

- **Marker :**

```
public void onLocationChanged(Location location) {
        latitude = location.getLatitude();
        longitude = location.getLongitude();
        altitude = location.getAltitude();
        accuracy = location.getAccuracy();

        String msg = String.format(
              getResources().getString(R.string.new_location), latitude,
              longitude, altitude, accuracy);
              mMap.addMarker(new MarkerOptions().position(new LatLng(latitude, longitude)).title("Marker")
                                    .icon(BitmapDescriptorFactory.fromResource(R.drawable.ic_launcher)));
            Toast.makeText(getApplicationContext(), msg, Toast.LENGTH_LONG).show();
}
```

.snippet("Population: 4,137,400")

- **Type :**

```
if (mMap != null) {
        mMap.setMapType(GoogleMap.MAP_TYPE_HYBRID);
        setUpMap();
}
```

```
mMap.setOnMarkerClickListener(new GoogleMap.OnMarkerClickListener() {
        @Override
        public boolean onMarkerClick(Marker marker) {
          Toast.makeText(getApplicationContext(), marker.getTitle(),
                Toast.LENGTH_SHORT).show();
          return false;
        }
});
```

PARTIE 3

PARTIE 3

Afficher des cartes

- **PolyLine :**

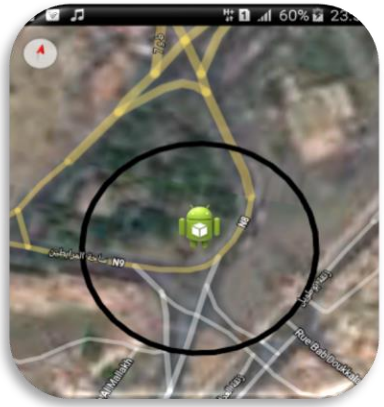
```
PolylineOptions rectOptions = new PolylineOptions()
            .add(new LatLng(31.6341600, -7.9999400))
            .add(new LatLng(34.0132500, -6.8325500));
      mMap.addPolyline(rectOptions);
```

- **Polygons :**

```
mMap.addPolygon(new PolygonOptions()
            .add( new LatLng(31.6341600, -7.9999400), new LatLng(34.0132500, -6.8325500),
new LatLng(34.0371500, -4.9998000)).fillColor(Color.BLUE));
```

- **Cercle :**

```
CircleOptions circleOptions = new CircleOptions()
            .center(new LatLng(31.6341600, -7.9999400))
            .radius(100); // en mètre
mMap.addCircle(circleOptions);
```



CHAPITRE 2

Demander une permission

1. Utilisation des Workflow pour la demande de permissions
2. Utilisation des permissions de localisation
3. **Présentation de la permission d’accès au réseau**
4. Présentation de la permission d’accès aux contacts du téléphone
5. Présentation de la permission d’accès au Bluetooth

Permissions dans AndroidManifest

Internet :

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

Vérification de l'état du réseau :

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

Obtenir des informations sur le réseau

- **ConnectivityManager :**
  - Répond aux questions sur l'état de la connectivité du réseau
  - Notifie les applications lorsque la connectivité du réseau change
- **NetworkInfo :**
  - Décrit l'état d'une interface réseau d'un type donné
  - Mobile ou Wi-Fi

Vérifier si le réseau est disponible

```
ConnectivityManager connMgr = (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
NetworkInfo networkInfo = connMgr.getActiveNetworkInfo();

if (networkInfo != null && networkInfo.isConnected()) {
    // Créer un thread d'arrière-plan pour se connecter et obtenir des données
    new DownloadWebpageTask().execute(stringUrl);
} else {
    textView.setText("No network connection available.");
}
```

Vérifier le WiFi et le Mobile

```
NetworkInfo networkInfo =
    connMgr.getNetworkInfo(ConnectivityManager.TYPE_WIFI);
boolean isWifiConn = networkInfo.isConnected();


networkInfo =
    connMgr.getNetworkInfo(ConnectivityManager.TYPE_MOBILE);
boolean isMobileConn = networkInfo.isConnected();
```

CHAPITRE 2

Demander une permission

1. Utilisation des Workflow pour la demande de permissions
2. Utilisation des permissions de localisation
3. Présentation de la permission d’accès au réseau
4. **Présentation de la permission d’accès aux contacts du téléphone**
5. Présentation de la permission d’accès au Bluetooth

Demander des permissions pour récupérer les contacts

Pour effectuer tout type de recherche dans le fournisseur des contacts, l'application doit avoir la permission **READ_CONTACTS**. Pour ce faire, ajouter cet élément **<uses-permission>** dans le fichier du manifeste en tant qu'élément enfant de **<manifest>** :

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

Récupérer les contacts

Afin de récupérer la liste des contacts, nous allons devoir utiliser la classe **ContentResolver** et plus précisément la méthode **query**.

Cette classe permet, notamment, via des requêtes de demander des informations aux systèmes : par exemple la liste des SMS ou la liste des contacts.

Le code suivant permet de lister les noms et les numéros de téléphone enregistrés dans le répertoire téléphonique au niveau de logcat.

```
Cursor phones = cr.query(ContactsContract.CommonDataKinds.Phone.CONTENT_URI, null, null, null, null);
while (phones.moveToNext()) {
      String name = phones.getString(phones.getColumnIndex(ContactsContract.CommonDataKinds.Phone.DISPLAY_NAME));
      String phoneNumber = phones.getString(phones.getColumnIndex(ContactsContract.CommonDataKinds.Phone.NUMBER));
      Log.d (" nom : " + name);
      Log.d (" numéro : " + phoneNumber);
}
phones.close();// close cursor
```

Intent implicite : composer un numéro ou appeler

- **ACTION_DIAL** : utiliser l'application Téléphone pour composer l'appel :
  - Avantage : action la plus simple sans avoir à demander la permission de l'utilisateur
  - Inconvénient : l'utilisateur doit retourner à l'application depuis l'application Téléphone
- **ACTION_CALL** : passer l'appel depuis l'application :
  - Avantage : l'utilisateur reste dans l'application
  - Inconvénient : il faut demander l'autorisation de l'utilisateur

URI d'un numéro de téléphone

Pour rappel : URI est l'acronyme de Uniform Resource Identifier.

Préparer un identificateur de ressources uniformes (URI) sous forme de chaîne de caractères :

- editText : EditText pour saisir un numéro de téléphone (21255551212) ;
- phoneNumber : chaîne préfixée par "tel :", par exemple (tel:21255551212) ;

```
String phoneNumber =
    String.format("tel : %s", editText.getText().toString()) ;
```

La classe PhoneNumberUtils

La classe **PhoneNumberUtils** fournit des méthodes utilitaires pour les chaînes de numéros de téléphone :

- **normalizeNumber()** supprime les caractères autres que les chiffres
- **formatNumber()** formate un numéro de téléphone pour un pays spécifique si le numéro n'inclut pas déjà un code de pays

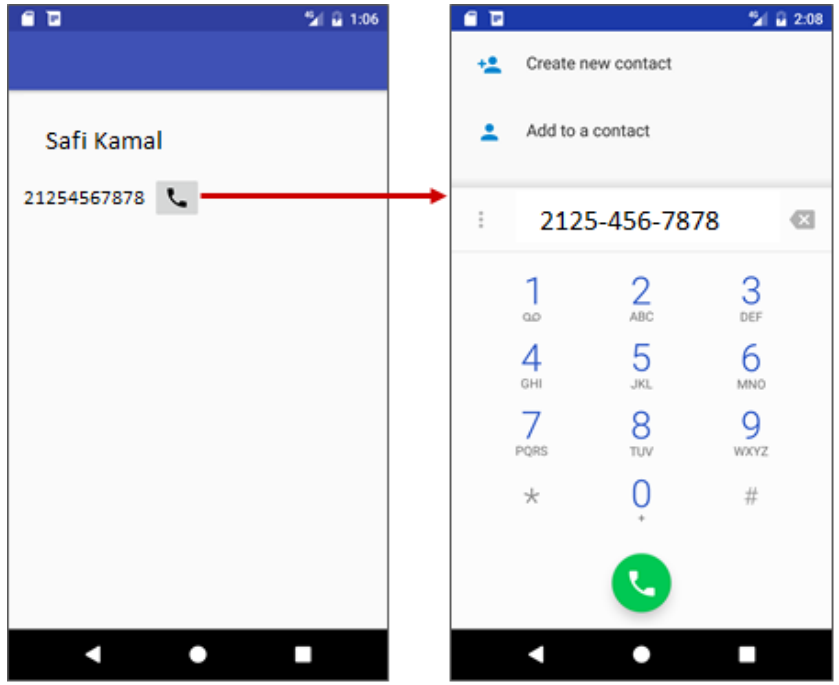
02 – Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

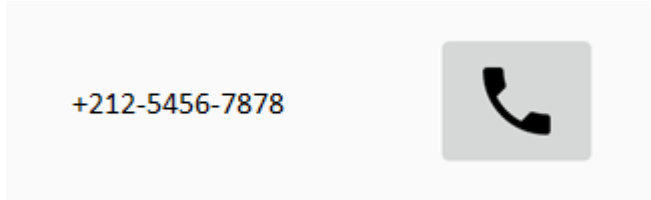
Intent avec ACTION_DIAL

Intent avec ACTION_DIAL utilise l'application téléphonique pour composer un appel

- L'utilisateur peut modifier le numéro de téléphone avant de composer l'appel ;
- L'utilisateur retourne à l'application en appuyant sur Retour.



Ajouter un handler onClick pour le bouton d'appel



- Ajouter un bouton d'appel et un numéro de téléphone
- Ajouter l'attribut **android:onClick** au bouton
- Créer un handler pour **onClick** :

```
public void callNumber(View view) {
        ...
}
```

PARTIE 3

Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

Créer un Intent avec un numéro de téléphone

1. Préparer un URI :

```
String phoneNumber =
        String.format("tel : %s", editText.getText().toString()) ;
```

2. Créer un **Intent** implicite :

```
Intent dialIntent = new Intent(Intent.ACTION_DIAL) ;
```

3. **setData()** avec le numéro de téléphone :

```
dialIntent.setData(Uri.parse(phoneNumber)) ;
```

Envoyer l'Intent à l'application Téléphone

- Utiliser **resolveActivity()** avec **getPackageManager()** pour déterminer l'Intent d'une application installée ;
- Si le résultat n'est pas nul, appeler **startActivity() :**

```
si (dialIntent.resolveActivity(getPackageManager()) != null) {
        startActivity(dialIntent) ;
} else {
.       .. // Consignation et affichage des explications
```

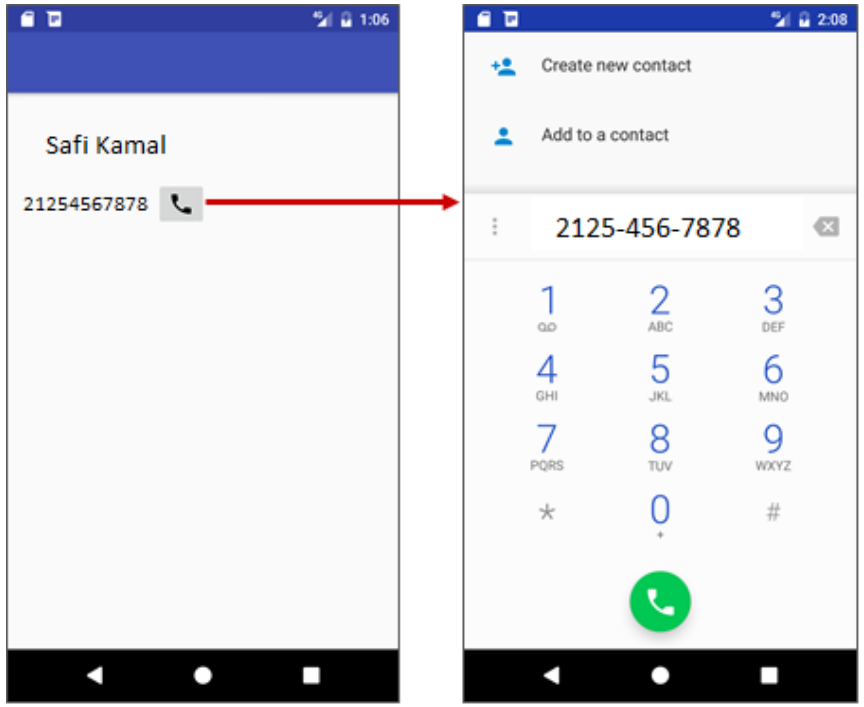
PARTIE 3

02 – Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

L'application téléphone compose et connecte l'appel

- Appuyer sur le bouton pour passer un appel
- Le numéro de téléphone apparaît dans l'application Téléphone
- Appuyer pour composer le numéro et passer l'appel



Intent avec ACTION_CALL

Utilisation de **ACTION_CALL** :

- Ajouter des permissions à **AndroidManifest.xml** :

<uses-permission android:name="android.permission.CALL_PHONE" />

<uses-permission android:name="android.permission.READ_PHONE_STATE" />

- Vérifier si la téléphonie est activée ; sinon, désactiver la fonction téléphone
- Vérifier si l'utilisateur accorde toujours la permission ; demander-la si nécessaire
- Étendre **PhoneStateListener**
- Enregistrer l'écouteur en utilisant **TelephonyManager**
- Créer un **Intent** implicite.

PARTIE 3

02 – Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

Vérification de TelephonyManager

- La fonctionnalité de téléphonie est fournie par **TelephonyManager** ;
- Certains appareils ne la possèdent pas, il faut donc vérifier :

```
private Boolean isTelephonyEnabled() {
    TelephonyManager mTelephonyManager =
     (TelephonyManager)getSystemService(TELEPHONY_SERVICE);
    if (mTelephonyManager != null) {
        if (mTelephonyManager.getSimState() ==
            TelephonyManager.SIM_STATE_READY) {
            return true;
        } else {
           return false;
        }
     }
```

La téléphonie est-elle activée ?

```
if (isTelephonyEnabled()) {
    // OK, la téléphonie est activée
} else {
  Toast.makeText(this,
  R.string.telephony_not_enabled,Toast.LENGTH_LONG).show();
  Log.d(TAG, getString(R.string.telephony_not_enabled));
  // Désactiver le bouton d'appel
}
```

PARTIE 3

02 – Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

Avez-vous l'autorisation de téléphoner ?

Si l'utilisateur a désactivé l'autorisation de téléphoner pour l'application, l'application peut demander l'autorisation de téléphoner.

Vérifier les étapes de permission

- Définir le nombre entier à utiliser pour le paramètre **requestCode** :

  private static final int MY_PERMISSIONS_REQUEST_CALL_PHONE = 1 ;

- Utiliser **checkSelfPermission() ;**
- Utiliser **requestPermissions()** :
  - **Contexte** (**this**), et **CALL_PHONE** dans le tableau de chaînes de permissions ;
  - La constante entière de requête : MY_PERMISSIONS_REQUEST_CALL_PHONE

Vérifier le code de permission

```
if (ActivityCompat.checkSelfPermission(this,
    Manifest.permission.CALL_PHONE) !=
    PackageManager.PERMISSION_GRANTED) {


        ActivityCompat.requestPermissions(this,
                new
                  String[]{Manifest.permission.CALL_PHONE},
                      MY_PERMISSIONS_REQUEST_CALL_PHONE);
```

PARTIE 3

02 – Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

Obtenir la réponse de l'utilisateur

Surcharger **onRequestPermissionsResult()** pour vérifier si le **requestCode** renvoyé est MY_PERMISSIONS_REQUEST_CALL_PHONE.

```
@Override
public void onRequestPermissionsResult(int requestCode,
                String permissions[], int[] grantResults) {
        switch (requestCode) {
            case MY_PERMISSIONS_REQUEST_CALL_PHONE : {
                ...
```

Vérifier si la demande a été acceptée

- Réponse retournée dans un tableau de permissions (index 0 si une seule demande a été envoyée)
- Comparer au résultat de l'octroi : **PERMISSION_GRANTED** ou **PERMISSION_DENIED** :

```
if (permissions[0].equalsIgnoreCase(Manifest.permission.CALL_PHONE)
        && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
    // La permission a été accordée.
} else {
    ...
```

PARTIE 3

02 – Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

Étendre de PhoneStateListener

- **PhoneStateListener** surveille les changements d'états de téléphonie :
  - Sonnerie, décrochage, inactivité.
- Étendre PhoneStateListener et surcharger **onCallStateChanged() :**

```
private class MyPhoneCallListener extends PhoneStateListener {
   @Override
   public void onCallStateChanged
                  (int state, String incomingNumber) {
   switch (state) {
   ...
```

Prévoir des actions pour les états du téléphone

Ajouter les actions à effectuer en fonction de l'état du téléphone :

- L'état est **CALL_STATE_IDLE** jusqu'à ce qu'un appel soit lancé
- L'état passe à **CALL_STATE_OFFHOOK** pour établir la connexion et reste dans cet état pendant toute la durée de l'appel
- L'état revient à **CALL_STATE_IDLE** après l'appel
- L'application reprend lorsque l'état redevient **CALL_STATE_IDLE**
- L'état est **CALL_STATE_RING** lorsqu'un appel entrant est détecté

PARTIE 3

Exemple de CALL_STATE_RINGING

```
switch (state) {
   case  TelephonyManager.CALL_STATE_RING :
      // Un appel entrant sonne, afficher le numéro.
      TextView incomingView = (TextView)
                        findViewById(R.id.incoming) ;
      incomingView.setText(incomingNumber) ;
      incomingView.setVisibility(View.VISIBLE) ;
      break ;
   case TelephonyManager.CALL_STATE_OFFHOOK :
      // L'appel téléphonique est actif (le téléphone est décroché).
      ...
```

Enregistrer un écouteur

Enregistrer l'écouteur dans la méthode **onCreate()** en utilisant **telephonyManager.listen()** avec le **PhoneStateListener** défini sur **LISTEN_CALL_STATE**.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    MyPhoneCallListener mListener = new
                        MyPhoneCallListener();
    telephonyManager.listen(mListener,

                    PhoneStateListener.LISTEN_CALL_STATE);
}
```

02 – Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

Envoyer un Intent avec un numéro de téléphone

- Préparer un **URI** ;
- Créer l'Intent implicite et inclure **setData()** avec le numéro de téléphone ;
- Utiliser **resolveActivity()** avec **getPackageManager()** pour résoudre l'intention implicite vers une application installée ;
- Si le résultat est non nul, appeler **startActivity()**.

Code pour envoyer l'Intent

```
String phoneNumber = String.format("tel: %s",

editText.getText().toString());
Intent callIntent = new Intent(Intent.ACTION_CALL);
callIntent.setData(Uri.parse(phoneNumber));
if (callIntent.resolveActivity(getPackageManager())
                != null) {
    startActivity(callIntent);
} else {
    … // Log et explication de la démonstration
```

PARTIE 3

02 – Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

Intent implicite et SmsManager

- **ACTION_SENDTO** avec intention implicite : utiliser l'application de messagerie installée :
  - **Avantage** : action la plus simple sans avoir à demander l'autorisation de l'utilisateur
  - **Inconvénient** : l'utilisateur retourne à l'application à partir de l'application de messagerie
- **SmsManager** : Envoyer un message à partir de l'application :
  - **Avantage** : l'utilisateur reste dans l'application
  - **Inconvénient** : il faut demander la permission à l'utilisateur

URI du numéro de téléphone pour l'envoi de SMS

Préparer un identificateur de ressources uniformes (URI) sous forme de chaîne de caractères

- editText : EditText pour saisir un numéro de téléphone (21255551212) ;
- smsNumber : Chaîne préfixée par "smsto :", par exemple (smsto:21255551212)

Exemple :

```
String smsNumber =
        String.format("smsto: %s",
    editText.getText().toString());
```

PARTIE 3

02 – Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

Intent avec ACTION_SENDTO

ACTION_SENDTO : utiliser une application de messagerie installée pour envoyer le message.

- L'utilisateur peut modifier le numéro de téléphone et le message avant l'envoi
- L'utilisateur retourne à l'application en appuyant sur le bouton Retour

Créer un Intent avec le téléphone et le message

1. Préparer un URI :

   String smsNumber =

   String.format("smsto : %s", editText.getText().toString()) ;

2. Créer un Intent implicite :

   Intent smsIntent = new Intent(Intent.ACTION_SENDTO) ;

3. Appeler **setData()** avec le numéro de téléphone :

   smsIntent.setData(Uri.parse(smsNumber)) ;

4. Appeler **putExtra()** avec la chaîne de messages sms et la clé "sms_body" :

   smsIntent.putExtra("sms_body", sms) ;

Envoyer l'Intent à une application de messagerie

- Utiliser **resolveActivity()** avec **getPackageManager()** pour résoudre l'intention implicite d'une application installée
- Si le résultat est non nul, appeler **startActivity()**

```
if (smsIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(smsIntent);
} else {
    … // Log
```

PARTIE 3

02 – Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

Utilisation de SmsManager

1. Ajouter la permission à **AndroidManifest.xml** :

   <uses-permission android:name="android.permission.SEND_SMS" />

2. Vérifier si l'utilisateur accorde toujours la permission SMS, ou la demander si nécessaire
3. Utiliser la méthode **sendTextMessage()**

Vérifier les étapes de permission

- Définir le nombre entier à utiliser pour le paramètre **requestCode** :

  private static final int MY_PERMISSIONS_REQUEST_SEND_SMS = 1 ;

- Utiliser **checkSelfPermission() ;**
- Utiliser **requestPermissions() ;**
  - **Contexte (this),** et **SEND_SMS** dans le tableau de permissions de la chaîne
  - La constante entière de demande **MY_PERMISSIONS_REQUEST_SEND_SMS**

PARTIE 3

02 – Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

Vérifier le code d'autorisation

```
if (ActivityCompat.checkSelfPermission(this,
                    Manifest.permission.SEND_SMS) !=
                    PackageManager.PERMISSION_GRANTED) {
    // Permission non encore accordée. Utiliser requestPermissions()
    ActivityCompat.requestPermissions(this,
                    new String[]{Manifest.permission.SEND_SMS},
                    MY_PERMISSIONS_REQUEST_SEND_SMS);
```

Obtenir la réponse de l'utilisateur

Surcharger **onRequestPermissionsResult()** pour vérifier si le requestCode renvoyé est MY_PERMISSIONS_REQUEST_SEND_SMS.

```
@Override
public void onRequestPermissionsResult(int requestCode,
                    String permissions[], int[] grantResults) {
    switch (requestCode) {
        case MY_PERMISSIONS_REQUEST_SEND_SMS: {
            ...
```

Vérifier si la demande a été accordée

- Réponse retournée dans un tableau de permissions (index 0 si une seule demande a été reçue) ;
- Comparer avec le résultat de l'octroi : **PERMISSION_GRANTED** ou **PERMISSION_DENIED**.

```
if (permissions[0].equalsIgnoreCase(Manifest.permission.SEND_SMS)
            && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
        // La permission a été accordée.
    } else {
        ...
```

Utilisation de SmsManager pour envoyer

Utiliser **sendTextMessage()** avec les paramètres suivants

- **destinationAddress** : chaîne pour le numéro de téléphone ;
- **scAddress** : chaîne pour l'adresse du SMSC (centre de service), ou null pour la valeur par défaut ; généralement prédéfinie dans la carte SIM ;
- **smsMessage** : chaîne pour le corps du message ;
- **sentIntent** : PendingIntent diffusé lorsque le message est envoyé avec succès ou si le message a échoué, ou utiliser null ;
- **deliveryIntent** : PendingIntent diffusé lorsque le message est remis au destinataire, ou utiliser null.

Étapes pour envoyer un message SMS (1)

1. Déclarer les paramètres string et PendingIntent :

...

// Définir l'adresse du centre de service si nécessaire, sinon null.

String scAddress = null ;

// Définir les intents en attente de diffusion

// lorsque le message est envoyé et lorsqu'il est livré, ou définir à null.

PendingIntent sentIntent = null, deliveryIntent = null ;

...

Étapes pour envoyer un message SMS (2)

2. Créer un smsManager ;
3. Utiliser sendTextMessage().

```
SmsManager smsManager = SmsManager.getDefault();
smsManager.sendTextMessage
          (destinationAddress, scAddress, smsMessage,
                        sentIntent, deliveryIntent);
```

Réception des messages SMS

Utilisation de broadcast receiver.

1. Ajouter la permission à **AndroidManifest.xml :**

   <uses-permission android:name="android.permission.RECEIVE_SMS" />

2. Ajouter **BroadcastReceiver ;**
3. Enregistrer le récepteur de diffusion pour l'intention **SMS_RECEIVED ;**
4. Surcharger la méthode **onReceive()** de **broadcast receiver.**

Ajouter le broadcast receiver

1. Sélectionner le nom du paquet ;
2. Choisir **File > New > Other > Broadcast Receiver :**
   - Vérifier que les cases "**Exported**" et "**Enabled**" sont cochées ;
   - Android Studio génère des balises dans **AndroidManifest.xml.**

```
<receiver
android:name="com.example.android.smsmessaging.MySmsReceiver"
    android:enabled="true"
    android:exported="true">
</receiver>
```

02 – Demander une permission

Présentation de la permission d'accès aux contacts du téléphone

Enregistrer le broadcast receiver

Enregistrer le receiver pour l'Intent **android.provider.Telephony.SMS_RECEIVED** :

```
<receiver
  android:name="com.example.android.smsmessaging.MySmsReceiver"
    android:enabled="true"
    android:exported="true">
    <intent-filter>
        <action
           android:name="android.provider.Telephony.SMS_RECEIVED"/>
    </intent-filter>
</receiver>
```

Surcharger receivers onReceive()

1. Obtenir un message SMS : récupérer les extras de l'Intent, stocker dans le bundle et définir le tableau des msgs
2. Obtenir le format à utiliser avec **createFromPdu()** pour créer un **SmsMessage**
3. Utiliser **createFromPdu()** pour remplir le tableau de msgs
4. Récupérer l'adresse d'origine à l'aide de **getOriginatingAddress()**
5. Récupérer le corps du message avec **getMessageBody()**

PARTIE 3

1. Obtenir un message SMS

Récupérer les extras de l'Intent, les stocker dans un bundle, et définir le tableau des msgs.

```
@Override
public void onReceive(Context context, Intent intent) {
    // Obtenir le message SMS.
    Bundle bundle = intent.getExtras();
    SmsMessage[] msgs;
    String strMessage = "";
    String format = bundle.getString("format");
```

2. Obtenir le format

- Utiliser **createFromPdu()** pour créer le **SmsMessage**, en utilisant le format fourni dans le bundle.
- Le format est issu d'un broadcast **SMS_RECEIVED_ACTION** :
  - "3gpp" pour les messages GSM/UMTS/LTE au format 3GPP, ou
  - "3gpp2" pour les messages CDMA/LTE au format 3GPP2

String format = bundle.getString("format") ;

3. Utiliser createFromPdu()

Utiliser **createFromPdu()** pour remplir le tableau des msgs

- Android version 6.0 (Marshmallow) et plus récente :
  - createFromPdu(byte[] pdu, String format) ;
  - msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i], format).
- Versions antérieures d'Android :
  - createFromPdu(byte[] pdu) ;
  - msgs[i] = SmsMessage.createFromPdu((byte[]) pdus[i]).

4. Obtenir l'adresse d'origine

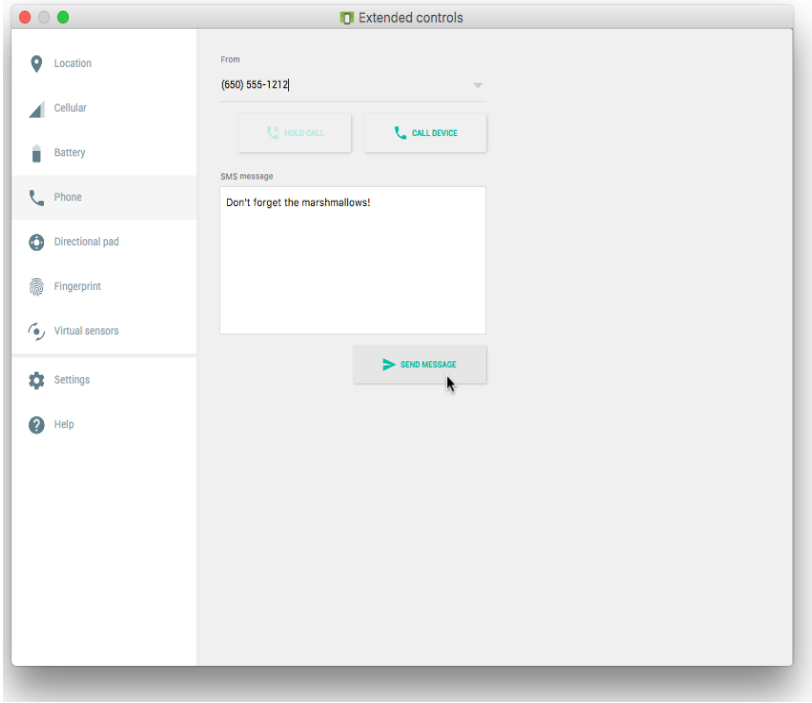
Obtenir l'adresse d'origine en utilisant **getOriginatingAddress()** :

```
strMessage += "SMS from " + msgs[i].getOriginatingAddress();
```

5. Obtenir le corps du message

Obtenir le corps du message en utilisant **getMessageBody()** :

```
strMessage += " :" + msgs[i].getMessageBody() + "\n";
```



Réception des messages dans l'émulateur

1. Lancer l'application sur l'émulateur
2. Cliquer sur l'icône ... (Plus)
3. Cliquer sur Téléphone dans la colonne de gauche
4. Entrer un message
5. Cliquer sur Envoyer le message

CHAPITRE 2

Demander une permission

1. Utilisation des Workflow pour la demande de permissions
2. Utilisation des permissions de localisation
3. Présentation de la permission d’accès au réseau
4. Présentation de la permission d’accès aux contacts du téléphone
5. **Présentation de la permission d’accès au Bluetooth**

02 – Demander une permission

Présentation de la permission d’accès au Bluetooth

Aperçu

La plateforme Android prend en charge la pile réseau Bluetooth, qui permet à un appareil d'échanger des données sans fil avec d'autres appareils Bluetooth. Le cadre d'application permet d'accéder à la fonctionnalité Bluetooth par le biais des API Bluetooth. Ces API permettent aux applications de se connecter à d'autres périphériques Bluetooth, ce qui permet des fonctions sans fil point à point et multipoint.

En utilisant les API Bluetooth, une application peut effectuer les opérations suivantes :

- Rechercher d'autres périphériques Bluetooth
- Interroger l'adaptateur Bluetooth local pour les périphériques Bluetooth appariés
- Établir des canaux RFCOMM (Radio frequency communication)
- Se connecter à d'autres appareils par la découverte de services
- Transférer des données vers et depuis d'autres appareils
- Gérer des connexions multiples

Configurer le Bluetooth

Avant que l'application ne puisse communiquer par Bluetooth ou Bluetooth Low Energy, il faut vérifier que le Bluetooth est bien pris en charge par l'appareil et, si c'est le cas, il faut s'assurer qu'il est activé. À noter que cette vérification n'est nécessaire que si l'attribut **android:required** de l'entrée du fichier manifeste **<uses-feature.../>** est défini sur false.

Si le Bluetooth n'est pas pris en charge, il convient de désactiver gracieusement toutes les fonctionnalités Bluetooth. Si le Bluetooth est pris en charge, mais désactivé, il est alors possible de demander à l'utilisateur d'activer le Bluetooth sans quitter l'application.

La première étape consiste à ajouter les permissions Bluetooth dans le fichier **manifest** afin d'utiliser les API Bluetooth.

Une fois les autorisations en place, la configuration du Bluetooth s'effectue en deux étapes à l'aide du **BluetoothAdapter** :

1. **Obtenir le BluetoothAdapter.**

Le **BluetoothAdapter :** est nécessaire pour toute activité Bluetooth. Le **BluetoothAdapter** représente l'adaptateur Bluetooth propre à l'appareil (la radio Bluetooth). Pour obtenir un **BluetoothAdapter**, il faut d'abord disposer d'un contexte. Utiliser le contexte pour obtenir une instance du service système **BluetoothManager**. L'appel de **BluteoothManager#getAdapter** fournira un objet **BluetoothAdapter**. Si **getAdapter**() renvoie **null**, alors le périphérique ne supporte pas le Bluetooth.

Par exemple :

```
BluetoothManager bluetoothManager =
                                    getSystemService(BluetoothManager.class);
BluetoothAdapter bluetoothAdapter = bluetoothManager.getAdapter();
if (bluetoothAdapter == null) {
  // L'appareil ne prend pas en charge Bluetooth
}
```

2. **Activer le Bluetooth :**

Ensuite, il faut s'assurer que le Bluetooth est activé. Appeler **isEnabled**() pour vérifier si le Bluetooth est actuellement activé. Si cette méthode renvoie false, alors le Bluetooth est désactivé. Pour demander l'activation du Bluetooth, appeler **startActivityForResult()**, en passant une action d'intention **ACTION_REQUEST_ENABLE**. Cet appel émet une demande d'activation du Bluetooth via les paramètres du système (sans arrêter l'application).

Par exemple :

```
if (!bluetoothAdapter.isEnabled()) {
  Intent enableBtIntent = new
                              Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
  startActivityForResult(enableBtIntent, REQUEST_ENABLE_BT);
}
```

Rechercher des périphériques Bluetooth

Avant d'effectuer la recherche de périphériques, il est utile d'interroger l'ensemble des périphériques appariés pour voir si le périphérique souhaité est déjà connu. Pour ce faire, appeler **getBondedDevices().** Cette fonction renvoie un ensemble d'objets **BluetoothDevice** représentant les périphériques appariés. Par exemple, il est possible d'interroger tous les appareils appariés et d'obtenir le nom et l'adresse MAC de chaque appareil, comme le montre l'extrait de code suivant :

```
Set<BluetoothDevice> pairedDevices = bluetoothAdapter.getBondedDevices();
if (pairedDevices.size() > 0) {
  // Il y a des dispositifs jumelés. Récupérer le nom et l'adresse de chaque
  // appareil apparié.
   for (BluetoothDevice device : pairedDevices) {
     String deviceName = device.getName();
     String deviceHardwareAddress = device.getAddress(); // Adresse MAC
  }
}
```

Découvrir les appareils Bluetooth à proximité

Pour commencer à découvrir des périphériques, appeler **startDiscovery().**
Ce processus est asynchrone et renvoie une valeur booléenne indiquant si la découverte a été lancée avec succès. Le processus de découverte implique généralement un balayage de recherche d'environ 12 secondes, suivi d'un balayage de page de chaque périphérique trouvé pour récupérer son nom Bluetooth.

Pour recevoir des informations sur chaque périphérique découvert, l'application doit enregistrer un **BroadcastReceiver** pour l'intention **ACTION_FOUND**. Le système diffuse cette intention pour chaque périphérique. **L'intent** contient les champs supplémentaires **EXTRA_DEVICE** et **EXTRA_CLASS**, qui contiennent à leur tour un **BluetoothDevice** et une **BluetoothClass**, respectivement. L'extrait de code suivant montre comment enregistrer le traitement de la diffusion lorsque des périphériques sont découverts :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
  ...

 // Enregistrer les diffusions lorsqu'un périphérique est détecté.
  IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
  registerReceiver(receiver, filter);
}

// Créer un BroadcastReceiver pour ACTION_FOUND.
private final BroadcastReceiver receiver = new BroadcastReceiver() {
  public void onReceive(Context context, Intent intent) {
    String action = intent.getAction();
    if (BluetoothDevice.ACTION_FOUND.equals(action)) {
      // La découverte a trouvé un appareil. Obtenir l'objet BluetoothDevice
      // et ses informations à partir de l'Intent.
      BluetoothDevice device = intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
      String deviceName = device.getName();
      String deviceHardwareAddress = device.getAddress(); // Adresse MAC
    }
  }
};

@Override
protected void onDestroy() {
  super.onDestroy();
  ...

   // Ne pas oublier de désenregistrer le récepteur ACTION_FOUND.
  unregisterReceiver(receiver);
}
```

Connecter des appareils Bluetooth

Connecter en tant que serveur

Pour configurer un socket serveur et accepter une connexion, effectuer la séquence d'étapes suivante :

1. Obtenir un **BluetoothServerSocket** en appelant :

   listenUsingRfcommWithServiceRecord(String, UUID).

2. Démarrer l'écoute des requêtes de connexion en appelant **accept().**
3. Sauf s'il s'agit d'accepter des connexions supplémentaires, appeler **close().**

L'identifiant unique universel (**UUID**) est inclus dans l'entrée de protocole de découverte de services (SDP) et constitue la base de l'accord de connexion avec le dispositif client.

Voici un Thread simplifié pour le composant serveur qui accepte les connexions entrantes :

```
private class AcceptThread extends Thread {
  private final BluetoothServerSocket mmServerSocket;

  public AcceptThread() {
     // Utiliser un objet temporaire qui sera plus tard assigné à
     // car mmServerSocket est final.
    BluetoothServerSocket tmp = null;
    try {
       // MY_UUID est la chaîne UUID de l'application, également utilisée par le client.
      tmp = bluetoothAdapter.listenUsingRfcommWithServiceRecord(NAME, MY_UUID);
    } catch (IOException e) {
      Log.e(TAG, "Socket's listen() method failed", e);
    }
    mmServerSocket = tmp;
  }
  public void run() {
    BluetoothSocket socket = null;
     // Continue d'écouter jusqu'à ce qu'une exception se produise ou qu'un socket soit renvoyé.
    while (true) {
      try {
        socket = mmServerSocket.accept();
      } catch (IOException e) {
        Log.e(TAG, "Socket's accept() method failed", e);
        break;
      }

      if (socket != null) {
        // Une connexion a été acceptée. Effectuez le travail associé à
        // la connexion dans un Thread séparé.
        manageMyConnectedSocket(socket);
        mmServerSocket.close();
        break;
      }
    }
  }
   // Ferme le socket de connexion et provoque la fin du Thread.
   public void cancel() {
    try {
      mmServerSocket.close();
    } catch (IOException e) {
      Log.e(TAG, " Impossible de fermer le socket de connexion ", e);
    }
  }
}
```

Connecter en tant que serveur

Dans cet exemple, une seule connexion entrante est souhaitée, donc dès qu'une connexion est acceptée et que le **BluetoothSocket** est acquis, l'application passe le **BluetoothSocket** acquis à un Thread séparé, ferme le **BluetoothServerSocket**, et sort de la boucle.

Noter que lorsque **accept()** renvoie le **BluetoothSocket**, le socket est déjà connecté. Par conséquent, il convient de ne pas appeler **connect(),** comme dans le cas du côté client.

La méthode **manageMyConnectedSocket()** propre à l'application est conçue pour lancer le Thread de transfert des données, ce qui est abordé dans la rubrique consacrée au transfert des données Bluetooth.

En général, il faut fermer l'interface **BluetoothServerSocket** dès que les connexions entrantes ne sont plus acceptées. Dans cet exemple, **close()** est appelé dès que le **BluetoothSocket** est acquis. Il est également possible de fournir une méthode publique dans votre Thread qui peut fermer le **BluetoothSocket** privé dans le cas où il faudrait arrêter d'écouter sur ce socket serveur.

Connecter en tant que client

Afin d'initier une connexion avec un périphérique distant qui accepte les connexions sur un socket serveur ouvert, il faut d'abord obtenir un objet **BluetoothDevice** qui représente le périphérique distant. Pour savoir comment créer un **BluetoothDevice**, consulter la section "Trouver des périphériques Bluetooth". Ensuite, il faut utiliser l'objet **BluetoothDevice** pour acquérir un **BluetoothSocket** et lancer la connexion.

La procédure de base est la suivante :

1. **En utilisant le BluetoothDevice, obtenir un BluetoothSocket en appelant createRfcommSocketToServiceRecord(UUID) :**

*Cette méthode initialise un objet **BluetoothSocket** qui permet au client de se connecter à un périphérique **BluetoothDevice**. **L'UUID** transmis ici doit correspondre à l'UUID utilisé par le périphérique serveur lorsqu'il a appelé **listenUsingRfcommWithServiceRecord(String, UUID)** pour ouvrir son **BluetoothServerSocket**. Pour utiliser un **UUID** correspondant, coder en dur la chaîne UUID dans l'application, puis la référencer à partir du code du serveur et du client.*

2. **Initier la connexion en appelant connect(). Noter que la méthode est un appel bloquant :**

*Lorsqu'un client appelle cette méthode, le système effectue une recherche SDP pour trouver le dispositif distant dont l'UUID correspond. Si la recherche aboutit et que le dispositif distant accepte la connexion, il partage le canal RFCOMM à utiliser pendant la connexion, et la méthode **connect()** revient. Si la connexion échoue ou si la méthode **connect()** s'arrête (après environ 12 secondes), la méthode lève une **IOException**.*

Comme **connect()** est un appel bloquant, il faut toujours exécuter cette procédure de connexion dans un **Thread distinct** du **Thread** de l'activité principale (IU).

Exemple

La classe suivante illustre un exemple de base d'un Thread client qui initie une connexion Bluetooth :

```
private class ConnectThread extends Thread {
  private final BluetoothSocket mmSocket;
  private final BluetoothDevice mmDevice;

  public ConnectThread(BluetoothDevice device) {
    // Utilisation d'un objet temporaire qui est ensuite assigné à mmSocket
    // car mmSocket est final.
    BluetoothSocket tmp = null;
    mmDevice = device;
    try {
      // Obtenir un BluetoothSocket pour se connecter au BluetoothDevice donné.
      // MY_UUID est la chaîne UUID de l'application, également utilisée dans le code du serveur.
      tmp = device.createRfcommSocketToServiceRecord(MY_UUID);
    } catch (IOException e) {
      Log.e(TAG, "Socket's create() method failed", e);
    }
    mmSocket = tmp;
  }

  public void run() {
    // Annuler la découverte car cela ralentit la connexion.
    bluetoothAdapter.cancelDiscovery();
    try {
      // Connecter l'appareil distant via le socket. Cet appel se bloque
      // jusqu'à ce qu'il réussisse ou qu'il lève une exception.
      mmSocket.connect();
    } catch (IOException connectException) {
      // Impossible de se connecter ; fermer le socket et retourner.
      try {
        mmSocket.close();
      } catch (IOException closeException) {
        Log.e(TAG, "Could not close the client socket", closeException);
      }
      return;
    }
    // La tentative de connexion a réussi. Effectuer le travail associé à
    // la connexion dans un Thread séparé.
    manageMyConnectedSocket(mmSocket);
  }
  // Ferme le socket client et provoque la fin du Thread.
  public void cancel() {
    try {
      mmSocket.close();
    } catch (IOException e) {
      Log.e(TAG, " Impossible de fermer le socket client ", e);
    }
  }
}
```

Exemple

Remarquer que dans cet exemple, **cancelDiscovery()** est appelé avant la tentative de connexion. Il faut toujours appeler **cancelDiscovery()** avant **connect(),** parce que **cancelDiscovery()** réussit, que la découverte du périphérique soit en cours ou non. Si l'application doit déterminer si la découverte du périphérique est en cours, elle peut le faire en utilisant **isDiscovering().**

La méthode **manageMyConnectedSocket(),** propre à l'application, est conçue pour lancer le Thread de transfert des données, qui est abordé dans la section consacrée au transfert des données Bluetooth.

Une fois terminé avec **BluetoothSocket**, appeler **close().** Cela permet de fermer immédiatement le socket connecté et de libérer toutes les ressources internes associées.

Transfert de données Bluetooth

Une fois que la connexion à un périphérique Bluetooth a été établie avec succès, chacun d'entre eux possède un **BluetoothSocket** connecté.

Il est désormais possible de partager des informations entre les appareils.

En utilisant le **BluetoothSocket**, la procédure générale pour transférer des données est la suivante :

1. Obtenir **l'InputStream** et **l'OutputStream** qui gèrent les transmissions par le socket en utilisant respectivement **getInputStream()** et **getOutputStream().**
2. Lire et écrire des données dans les flux en utilisant **read(byte[])** et **write(byte[])**.

Il y a, bien sûr, des détails d'implémentation à prendre en compte. En particulier, il convient d'utiliser un Thread dédié pour la lecture et l'écriture dans le flux.

Ceci est important car les méthodes **read(byte[])** et **write(byte[])** sont des appels bloquants. La méthode **read(byte[])** bloque jusqu'à ce qu'il y ait quelque chose à lire dans le flux.

La méthode **write**(**byte[])** ne bloque généralement pas, mais elle peut le faire pour contrôler le flux si le périphérique distant n'appelle pas **read(byte[])** assez rapidement et que les tampons intermédiaires se remplissent en conséquence.

Par conséquent, il est préférable de consacrer la boucle principale du Thread à la lecture de **l'InputStream**.

Une méthode publique distincte peut être utilisée dans le Thread pour lancer les écritures dans le **OutputStream**.

Déclarer les permissions

Si l'application est destinée à Android 12 (niveau API 31) ou à une version ultérieure, déclarer les permissions suivantes dans le fichier manifeste de l'application :

- Si l'application recherche des périphériques Bluetooth, tels que des périphériques BLE, déclarer l'autorisation **BLUETOOTH_SCAN ;**
- Si l'application permet à d'autres périphériques Bluetooth de découvrir le périphérique actuel, déclarer l'autorisation **BLUETOOTH_ADVERTISE ;**
- Si l'application communique avec des périphériques Bluetooth déjà appariés, déclarer l'autorisation **BLUETOOTH_CONNECT**.

Pour les anciennes déclarations d'autorisations liées à Bluetooth, définir **android:maxSdkVersion** sur **30**.

Cette étape de compatibilité des applications permet au système d'accorder à l'application uniquement les autorisations Bluetooth dont elle a besoin lorsqu'elle est installée sur des appareils fonctionnant sous Android 12 ou une version ultérieure.

Si l'application utilise les résultats de l'analyse Bluetooth pour déterminer l'emplacement physique, déclarer l'autorisation **ACCESS_FINE_LOCATION**. Sinon, il est possible d'affirmer fermement que l'application ne dérive pas la localisation physique.

Les permissions **BLUETOOTH_ADVERTISE**, **BLUETOOTH_CONNECT**, et **BLUETOOTH_SCAN** sont des permissions d'exécution. Par conséquent, il faut demander explicitement l'approbation de l'utilisateur dans l'application avant de pouvoir rechercher des périphériques Bluetooth, rendre un périphérique détectable par d'autres périphériques ou communiquer avec des périphériques Bluetooth déjà appariés.

Lorsque l'application demande au moins une de ces autorisations, le système invite l'utilisateur à autoriser l'application à accéder aux périphériques proches.

L'extrait de code suivant montre comment déclarer les autorisations liées à Bluetooth dans l'application si celle-ci est destinée à Android 12 ou à une version ultérieure :

Déclarer les permissions

```
<manifest>
  <!-- Demander les autorisations Bluetooth héritées sur les anciens appareils. -->
   <uses-permission android:name="android.permission.BLUETOOTH"
                android:maxSdkVersion="30" />
   <uses-permission android:name="android.permission.BLUETOOTH_ADMIN"
                android:maxSdkVersion="30" />
<!-- Nécessaire uniquement si votre application recherche des périphériques Bluetooth. Si l'application n'utilise pas les résultats du balayage Bluetooth pour obtenir des informations de localisation physique, il est possible d'affirmer fermement que l'application ne dérive pas la localisation physique. -->
   <uses-permission android:name="android.permission.BLUETOOTH_SCAN" />
```

```
<!-- Nécessaire uniquement si l'application rend l'appareil détectable par les appareils Bluetooth. -->
   <uses-permission
android:name="android.permission.BLUETOOTH_ADVERTISE" />
   <!-- Nécessaire uniquement si votre application communique avec des périphériques Bluetooth déjà appariés. -->
   <uses-permission  android:name="android.permission.BLUETOOTH_CONNECT"
/>
<!-- Nécessaire uniquement si l'application utilise les résultats du balayage Bluetooth pour déterminer l'emplacement physique. -->
   <uses-permission
android:name="android.permission.ACCESS_FINE_LOCATION" />>
   ...
</manifest>
```

PARTIE 3

Cibler Android 11 ou inférieur

Si l'application est destinée à Android 11 (niveau API 30) ou à une version inférieure, déclarer les permissions suivantes dans le fichier manifeste de l'application :

- **BLUETOOTH** est nécessaire pour effectuer toute communication Bluetooth classique ou BLE, comme demander une connexion, accepter une connexion et transférer des données
- **ACCESS_FINE_LOCATION** est nécessaire car, sur Android 11 et inférieur, un balayage Bluetooth pourrait potentiellement être utilisé pour collecter des informations sur la localisation de l'utilisateur

Si l'application est destinée à Android 9 (niveau 28 de l'API) ou à une version inférieure, il convient de déclarer la permission **ACCESS_COARSE_LOCATION** au lieu de la permission **ACCESS_FINE_LOCATION**.

Découvrir les dispositifs Bluetooth locaux

Pour que l'application puisse lancer la découverte des périphériques ou manipuler les paramètres Bluetooth, il est nécessaire de déclarer l'autorisation **BLUETOOTH_ADMIN**.

La plupart des applications ont besoin de cette autorisation uniquement pour pouvoir découvrir les périphériques Bluetooth locaux.

Il convient de ne pas utiliser les autres capacités accordées par cette autorisation, sauf si l'application est un " power manager " qui modifie les paramètres Bluetooth à la demande de l'utilisateur.

Déclarer la permission dans le fichier manifeste de votre application.

Par exemple :

```
<manifest ... >
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />
...
</manifest>
```

Découvrir les dispositifs Bluetooth locaux

Lorsque l'application prend en charge un service et peut fonctionner sous Android 10 (niveau 29 de l'API) ou Android 11, il convient d'accorder la permission **ACCESS_BACKGROUND_LOCATION** pour découvrir les périphériques Bluetooth.

L'extrait de code suivant montre comment déclarer l'autorisation **ACCESS_BACKGROUND_LOCATION** :

```
<manifest ... >
<uses-permission
android:name="android.permission.ACCESS_BACKGROUND_LOCATION" />
...
</manifest>
```

Spécifier l'utilisation de la fonctionnalité Bluetooth

Lorsque Bluetooth est un élément essentiel de l'application, il suffit d'ajouter des drapeaux dans le fichier du manifeste pour indiquer cette exigence. L'élément **<uses-feature>** permet de préciser le type de matériel que l'application utilise et s'il est nécessaire ou non.

Cet exemple montre comment indiquer que le Bluetooth classique est requis pour l'application.

```
<uses-feature
android:name="android.hardware.bluetooth" android:required="true"/>
```

Si l'application repose sur la technologie Bluetooth Low Energy, il est recommandé d'utiliser les éléments suivants :

```
<uses-feature
android:name="android.hardware.bluetooth_le" android:required="true"/>
```

Vérifier la disponibilité de la fonctionnalité au moment de l'exécution

Pour rendre l'application disponible aux appareils qui ne prennent pas en charge Bluetooth classic ou BLE, il convient d'inclure l'élément **<uses-feature>** dans le manifeste de l'application, mais de définir **required="false"**.

Ensuite, au moment de l'exécution, il sera possible de déterminer la disponibilité de la fonctionnalité en utilisant **PackageManager.hasSystemFeature()** :

```
// Utiliser cette vérification pour déterminer si Bluetooth classic est pris en charge par l'appareil.
// Il est ensuite possible de désactiver de manière sélective les fonctions liées à BLE.
if (!getPackageManager().hasSystemFeature(PackageManager.FEATURE_BLUETOOTH)) {
    Toast.makeText(this, R.string.bluetooth_not_supported, Toast.LENGTH_SHORT).show();
    finish();
}
// Utiliser ce contrôle pour déterminer si BLE est pris en charge par l'appareil. Ensuite,
// il est possible de désactiver de manière sélective les fonctions liées à BLE.
if (!getPackageManager().hasSystemFeature(PackageManager.FEATURE_BLUETOOTH_LE)) {
    Toast.makeText(this, R.string.ble_not_supported, Toast.LENGTH_SHORT).show();
    finish();
}
```

PARTIE 4

Créer des tests unitaires

Dans ce module, vous allez :

- Apprendre à utiliser Junit
- Apprendre à utiliser Mockito
- Créer des tests automatisés

6 heures

CHAPITRE 1

Tester une classe

Ce que vous allez apprendre dans ce chapitre :

- Utilisation de Junit
- Utilisation de Mockito

3 heures

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

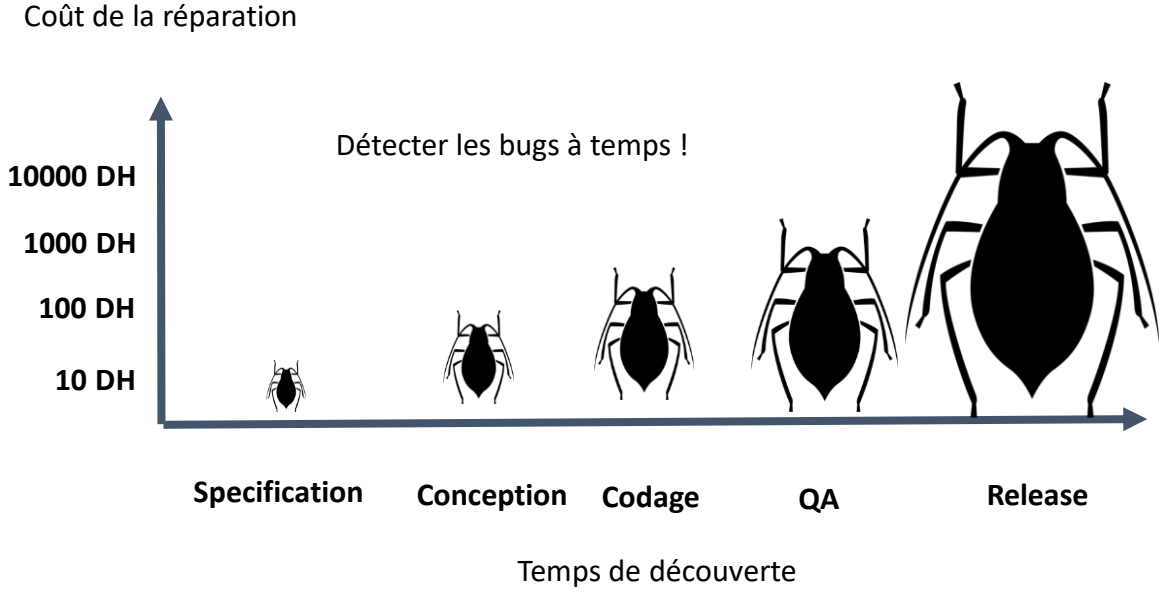
CHAPITRE 1

Tester une classe

1. **Utilisation de Junit**
2. Utilisation de Mockito

Pourquoi faut-il tester l'application ?

- Trouver et résoudre les problèmes à un stade précoce
- Réduire les coûts : les coûts de correction des bugs augmentent avec le temps
- Réduire les efforts



Types de tests

- Niveaux de test :
  - Composant, intégration, protocole, système
- Types de tests :
  - Installation, compatibilité, régression, acceptation
  - Performance, évolutivité, convivialité, sécurité
- Tests d'interface utilisateur et d'interaction :
  - Outils de tests automatisés de l'interface utilisateur
  - Tests instrumentés (traités dans un autre chapitre)

Développement piloté par les tests (TDD)

- Définir un scénario de test pour une exigence
- Écrire des tests qui vérifient toutes les conditions du scénario de test
- Écrire le code en fonction du test
- Itérer et remanier le code jusqu'à ce qu'il passe le test
- Répéter jusqu'à ce que toutes les exigences aient des cas de test, que tous les tests réussissent et que toutes les fonctionnalités aient été implémentées

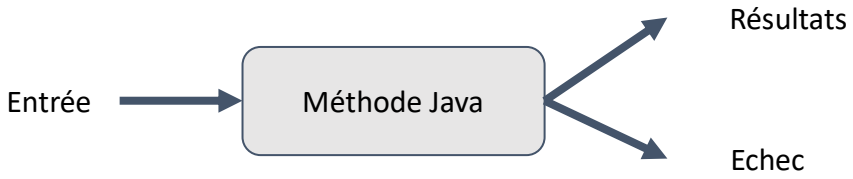
Tests dans le projet

Android Studio crée trois ensembles de sources pour le projet :

- **main** : code principal et ressources
- (**test**) : tests unitaires locaux
- (**androidTest**) : tests instrumentés

Tests unitaires

- Tester les plus petites parties du programme
- Isoler chaque composant et démontrer que les parties individuelles sont correctes
- Tester les méthodes Java



Tests unitaires locaux dans JUnit

- Compilé et exécuté entièrement sur la machine locale avec la machine virtuelle Java (**JVM**)
- À utiliser pour tester les parties de l'application (comme la logique interne) :
  - Si un accès au **framework Android** ou à l'appareil/émulateur n'est pas nécessaire
  - Si de faux objets (**mock**) peuvent être créés et se comporter comme les équivalents du framework
- Les tests unitaires sont écrits avec **JUnit**, un cadre commun de **tests unitaires** pour **Java**

Tests unitaires locaux dans le projet

- Les tests sont dans le même package que la classe d'application associée
- Seulement **org.junit** importé (pas de classes Android)
- Chemin du projet pour les classes de **test : .../nom-du-module/src/test/java/**

Importations pour JUnit

```
// Annotations
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;

// Exécutant de test JUnit4 de base
import org.junit.runners.JUnit4;

// Méthode assertThat
import static org.junit.Assert.assertThat;
```

Classe de test

```
/**
* Tests unitaires JUnit4 pour la logique de la
* calculatrice.
* Il s'agit de tests unitaires locaux ; aucun dispositif
* n'est nécessaire.
*/
@RunWith(JUnit4.class) // Spécifier le gestionnaire de test
public class CalculatorTest { // Nommer la classe à tester
}
```

ExampleTest

```
/
**
* Test pour une addition simple.
* Chaque test est identifié par une annotation @Test.
*/
@Test
public void addTwoNumbers() {
   double resultAdd = mCalculator.add(1d, 1d);
   assertThat(resultAdd, is(equalTo(2d)));
}
```

Annotation @Test

- Indique à **JUnit** que cette méthode est une méthode de test (**JUnit 4**)
- Information pour l'exécuteur de test
- Il n'est plus nécessaire de préfixer les méthodes de test par "test"

Méthode setUp()

```
/**
* Configurer l'environnement pour les tests
*/
@Before
public void setUp() {
   mCalculator = new Calculator();
}
```

- Configurer l'environnement pour les tests ;
- Initialiser des variables et des objets utilisés dans les tests multiples.

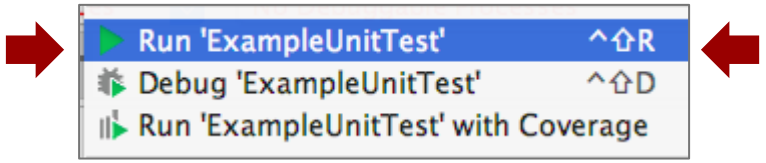
Méthode tearDown()

```
/**
* Libérer les ressources externes
*/
@After
public void tearDown() {
    ....
}
```

- Libérer les ressources.

Exécution des tests dans Android Studio

- Avec le bouton droit de la souris, cliquer sur la classe de test et sélectionner Run '**app_name**' test
- Cliquer avec le bouton droit de la souris sur le paquet de tests et sélectionner "**Run tests in 'package'**"



PARTIE 4

Test de la virgule flottante

- Attention aux tests en virgule flottante
- Rappel des notions de base en informatique : l'arithmétique à virgule flottante n'est pas précise en binaire

```
12  public class FloatingPointUnitTest {
13      @Test
14      public void addition_isCorrect() {
15          assertEquals( expected: .3d,  actual: .1d + .2d, delta: 0.d );
16      }
17  }
```

```
orrect ×
Tests failed: 1 of 1 test – 6 ms
expected:<0.3> but was:<0.30000000000000004>
Expected :0.3
Actual   :0.30000000000000004
```

Correction du test avec les nombres à virgule flottante

Ils sont identiques à 0,0005 près dans ce test.

```
12  public class FloatingPointUnitTest {
13      @Test
14      public void addition_isCorrect() {
15          assertEquals( expected: .3d,  actual: .1d + .2d, delta: 0.0005d );
16      }
17  }
```

```
_isCorrect ×
Tests passed: 1 of 1 test – 4 ms
BUILD SUCCESSFUL in 4s
```

Test Email

L'exemple suivant montre comment implémenter une classe de test unitaire locale. La méthode de test **emailValidator_correctEmailSimple_returnsTrue()** tente de vérifier **isValidEmail**(), qui est une méthode de l'application. La fonction de test retournera vrai si **isValidEmail**() retourne aussi vrai.

```
import org.junit.Test;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;

class EmailValidatorTest {
  @Test
  public void emailValidator_CorrectEmailSimple_ReturnsTrue() {
        assertTrue(EmailValidator.isValidEmail("name@email.com"));
  }
}
```

CHAPITRE 1

Tester une classe

1. Utilisation de Junit
2. **Utilisation de Mockito**

Bibliothèque Android mockable

Un problème typique est de découvrir qu'une classe utilise une ressource de type chaîne. Les ressources de type chaîne peuvent être obtenues en appelant la méthode **getString**() de la classe **Context**. Cependant, un test local ne peut pas utiliser Context ni aucune de ses méthodes car elles appartiennent au framework Android. Idéalement, l'appel à **getString**() devrait être déplacé hors de la classe, mais cela n'est pas toujours pratique. La solution consiste à créer un **mock** ou un stub de **Context** qui renvoie toujours la même valeur lorsque sa méthode **getString**() est invoquée.

Avec la bibliothèque **Mockable** Android et les frameworks de mocking tels que **Mockito** ou **MockK**, il est possible de programmer le comportement des **mocks** des classes Android dans les tests unitaires.

Pour ajouter un objet **mock** à l'aide de **Mockito** à un test unitaire local, suivre le modèle de programmation suivant :

1. Inclure la dépendance de la bibliothèque **Mockito** dans le fichier **build.gradle**.
2. Ajouter l'annotation **@RunWith(MockitoJUnitRunner.class)** au début de la définition de la classe de test unitaire. Cette annotation indique à l'exécuteur de test **Mockito** de valider que l'utilisation du framework est correcte et simplifie l'initialisation de l'objet fictif.
3. Pour générer un objet fictif pour une dépendance Android, ajouter l'annotation **@Mock** avant la déclaration du champ.
4. Pour stub le comportement de la dépendance, il est possible de spécifier une condition et une valeur de retour lorsque la condition est remplie en utilisant les méthodes **when**() et **thenReturn**().

Ajouter les dépendances de test

```
dependencies {
 // JUnit 4 framework
 testImplementation "junit:junit:$jUnitVersion"
 // Robolectric environment
 testImplementation "androidx.test:core:$androidXTestVersion"
 // Mockito framework
 testImplementation "org.mockito:mockito-core:$mockitoVersion"
 // mockito-kotlin
 testImplementation "org.mockito.kotlin:mockito-kotlin:$mockitoKotlinVersion"
 // Mockk framework
 testImplementation "io.mockk:mockk:$mockkVersion"

}
```

01 – Bibliothèque Android mockable

L'exemple suivant montre comment il est possible de créer un test unitaire qui utilise un objet Contexte fictif en **Kotlin** créé avec **Mockito-Kotlin**.

```
import android.content.Context
import org.junit.Assert.assertEquals
import org.junit.Test
import org.junit.runner.RunWith
import org.mockito.Mock
import org.mockito.junit.MockitoJUnitRunner
import org.mockito.kotlin.doReturn
import org.mockito.kotlin.mock
```

```
private const val FAKE_STRING = "HELLO WORLD"

@RunWith(MockitoJUnitRunner::class)
class MockedContextTest {

 @Mock
 private lateinit var mockContext: Context

 @Test
 fun readStringFromContext_LocalizedString() {
  // Étant donné un contexte simulé injecté dans l'objet à tester...
  val mockContext = mock<Context> {
     on { getString(R.string.name_label) } doReturn "FAKE_STRING"
  }

  val myObjectUnderTest = ClassUnderTest(mockContext)

  // ...lorsque la chaîne de caractères est renvoyée par l'objet testé...
  val result: String = myObjectUnderTest.getName()

  // ...alors le résultat devrait être celui attendu.
  assertEquals(result, FAKE_STRING)
 }
}
```

CHAPITRE 2

Créer des tests d’interface

Ce que vous allez apprendre dans ce chapitre :

- Utilisation d’Espresso
- Utilisation d’UI Automator

6 heures

CHAPITRE 2

Créer des tests d'interface

1. **Utilisation d'Espresso**
2. Utilisation d'UI Automator

Tests de l'interface utilisateur

- Effectuer toutes les actions de l'interface utilisateur avec des éléments de vue :
  - Taper sur une vue, puis saisir des données ou faire un choix
  - Examiner les valeurs des propriétés de chaque vue
- Fournir des données à tous les éléments de la vue :
  - Essayer les valeurs non valide
- Vérifier la sortie retournée :
  - Valeurs correctes ou attendues ?
  - Présentation correcte ?

Problèmes liés aux tests manuels

- Coûteux en temps, fastidieux, sujets aux erreurs
- L'interface utilisateur peut changer et nécessiter de nouveaux tests fréquents
- Certains chemins échouent avec le temps
- Au fur et à mesure que l'application devient plus complexe, les séquences d'actions possibles peuvent augmenter de façon non linéaire

Avantages des tests automatiques

- Libération de temps et de ressources pour d'autres tâches
- Plus rapides que les tests manuels
- Répétables
- Exécution des tests pour différents états et configurations de dispositifs

Espresso pour le test d'une seule application

- Vérifier que l'interface utilisateur se comporte comme prévu
- Vérifier que l'application renvoie la sortie correcte de l'interface utilisateur en réponse aux interactions de l'utilisateur
- Vérifier si la navigation et les contrôles se comportent correctement
- Vérifier si l'application répond correctement aux dépendances simulées

UI Automator pour plusieurs applications

- Vérifier que les interactions entre les différentes applications de l'utilisateur et les applications du système se comportent comme prévu
- Interagir avec les éléments visibles d'un appareil
- Surveiller les interactions entre les applications et le système
- Simuler les interactions de l'utilisateur
- Nécessite une instrumentation

Qu'est-ce que l'instrumentation ?

- Un ensemble de **hooks** dans le système Android
- Charge le paquet de test et l'application dans le même processus, permettant aux tests d'appeler les méthodes et d'examiner les champs
- Contrôle des composants indépendamment du cycle de vie de l'application
- Contrôle de la façon dont Android charge les applications

Avantages de l'instrumentation

- Les tests peuvent surveiller toutes les interactions avec le système Android
- Les tests peuvent invoquer des méthodes dans l'application
- Les tests peuvent modifier et examiner les champs de l'application indépendamment du cycle de vie de l'application

Environnement de test et configuration de l'Espresso

1. Dans Android Studio, Choisir **Tools > Android > SDK Manager**
2. Cliquer sur **SDK Tools** et rechercher **Android Support Repository**
3. Si nécessaire, mettre à jour ou installer la bibliothèque

Ajouter les dépendances à build.gradle

- Les modèles Android Studio incluent des dépendances
- Si nécessaire, ajouter les dépendances suivantes :

```
androidTestImplementation 'androidx.test.espresso:espresso-core:3.4.0'
androidTestImplementation 'androidx.test:runner:1.4.0'
androidTestImplementation 'androidx.test:rules:1.4.0'
```

Ajouter defaultConfig à build.gradle

- Les modèles Android Studio incluent le paramètre **defaultConfig**
- Si nécessaire, ajouter ce qui suit à la section **defaultConfig** :

```
testInstrumentationRunner "androidx.test.runner.AndroidJUnitRunner"
```

02 – Créer des tests d'interface
Utilisation d'Espresso

Préparer l'appareil

1. Activer le débogage USB
2. Désactiver toutes les animations dans Options du **développeur > Drawing** :
    - Échelle de l'animation de la fenêtre
    - Échelle d'animation de la transition
    - Échelle de la durée de l'animation

Créer des tests

- Stocker dans **module-name/src/androidTests/java/** :
    - Dans Android Studio : **app > java > nom-de-module (androidTest)/**
- Créer des tests en tant que classes **Junit**

PARTIE 4

Définition de la classe de test

@RunWith(AndroidJUnit4.class) : annotation obligatoire pour les tests

@LargeTest : basé sur les ressources utilisées par le test et le temps d'exécution

public class ChangeTextBehaviorTest {}

@SmallTest : s'exécute en < 60s et n'utilise aucune ressource externe

@MediumTest : s'exécute en < 300s, uniquement sur le réseau local

@LargeTest : fonctionne pendant une longue période et utilise de nombreuses ressources

@Rule spécifie le contexte du test

@Rule

publique ActivityTestRule<MainActivity> mActivityRule =

new ActivityTestRule<>(MainActivity.class) ;

@ActivityTestRule : test de la prise en charge d'une seule activité spécifiée

@ServiceTestRule : test de la prise en charge du démarrage, de la liaison et de l'arrêt d'un service

@Before et @After

```
@Before
public void initValidString() {
   mStringToBetyped = "Espresso" ;
}
```

@Before : mise en place, initialisation

@After : mise hors service, libération des ressources

Structure de la méthode @Test

```
@Test
public void changeText_sameActivity() {
   // 1. Trouver une vue
   // 2. Exécuter une action
   // 3. Vérifier que l'action a été effectuée, assert result
}
```

"Hamcrest" simplifie les tests

- "**Hamcrest**", anagramme de "**Matchers**"
- Cadre pour la création de concordances et d'assertions personnalisées
- Règles de correspondance définies de manière déclarative
- Permet des tests précis
- Le tutoriel Hamcrest

Hamcrest Matchers

- **ViewMatcher** : trouve les vues par id, contenu, focus, hiérarchie
- **ViewAction** : exécute une action sur une vue
- **ViewAssertion** : affirme l'état et vérifie le résultat

Exemple de test basique

```
@Test
public void changeText_sameActivity() {
   // 1. Trouver la vue par l'Id
   onView(withId(R.id.editTextUserInput))
   // 2. Exécuter une chaîne de type action et cliquer sur le bouton
   .perform(typeText(mStringToBetyped), closeSoftKeyboard()) ;
   onView(withId(R.id.changeTextBt)).perform(click()) ;
   // 3. Vérifier que le texte a été modifié
   onView(withId(R.id.textToBeChanged))
      .check(matches(withText(mStringToBetyped))) ;
}
```

Trouver des vues avec onView

- **withId()** - trouver une vue avec l'id Android spécifié :
  - onView(withId(R.id.editTextUserInput))
- **withText()** - trouver une vue avec un texte spécifique ;
- **allOf()** - trouver une vue qui correspond à plusieurs conditions ;
- **Exemple** : trouver un élément de liste visible avec le texte donné :

```
onView(allOf(withId(R.id.word),
        withText("cliqué ! Mot 15"),
        isDisplayed())
```

onView renvoie l'objet ViewInteraction

- Pour réutiliser la vue retournée par **onView** :
  - Rendre le code plus lisible ou plus explicite
  - Méthodes **check()** et **perform()**

```
ViewInteraction textView = onView(
    allOf(withId(R.id.word), withText("Cliquer ! Mot 12"),
    isDisplayed()));
textView.check(matches(withText(" Cliquer ! Mot 12")));
```

Effectuer des actions

- Effectuer une action sur la vue trouvée par un **ViewMatcher**
- Il peut s'agir de n'importe quelle action sur la vue :

```
// 1. Trouver la vue par l'Id
onView(withId(R.id.editTextUserInput))
 // 2. Exécuter une action : type texte et cliquer sur un bouton
.perform(typeText(mStringToBetyped), closeSoftKeyboard()) ;
onView(withId(R.id.changeTextBt)).perform(click()) ;
```

- Vérifier l'état de la vue :

```
// 3. Vérifier que le texte a été modifié
onView(withId(R.id.textToBeChanged))
 .check(matches(withText(mStringToBetyped))) ;
```

Quand un test échoue

Test :

```
onView(withId(R.id.text_message))
    .check(matches(withText("Ceci est un test échoué.")));
```

Extrait du résultat :

```
android.support.test.espresso.base.DefaultFailureHandler$AssertionFailedWithCauseError: 'with text: is "Ceci est un test échoué."' doesn't match the selected view.
Expected: with text: is "Ceci est un test échoué."
Got: "AppCompatTextView{id=2131427417, res-name=text_message ...
```

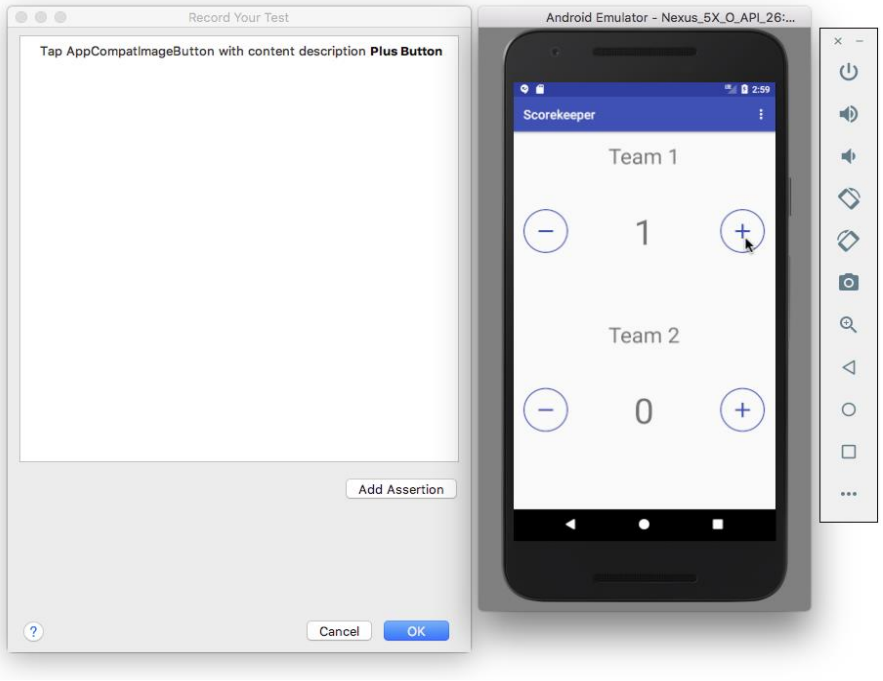
Enregistrement d'un test Espresso

- Utiliser l'application normalement, en cliquant sur l'interface utilisateur
- Code de test modifiable généré automatiquement
- Ajouter des assertions pour vérifier si une vue contient une certaine valeur
- Enregistrer plusieurs interactions en une seule session, ou enregistrer plusieurs sessions

PARTIE 4

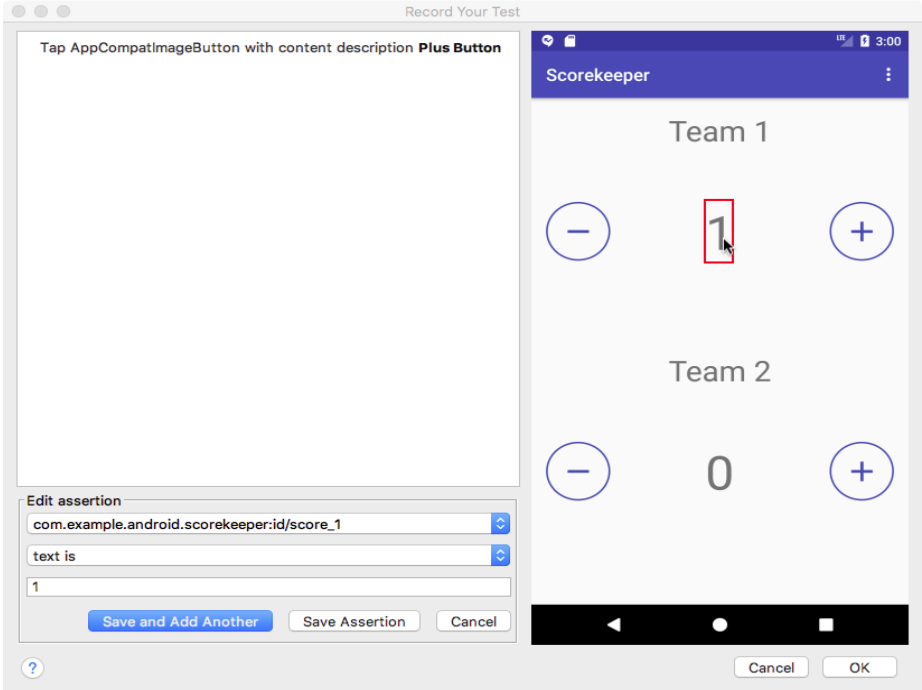
Commencer à enregistrer un test Espresso

1. **Run > Record Espresso Test**
2. Cliquer sur **Restart App**, sélectionner la cible et cliquer sur **OK**
3. Interagir avec l'application pour réaliser le test désiré



Ajouter une assertion à l'enregistrement du test Espresso

4. Cliquer sur **Add Assertion** et sélectionner un élément de l'interface utilisateur
5. Choisir le **text is** et entrer le texte attendu
6. Cliquer sur **Save Assertion** et sur **Complete Recording**



Pour en savoir plus, consulter les documents destinés aux développeurs.

Documentation d'Android Studio

- Test Your App
- Espresso basics
- Espresso cheat sheet

Bibliothèque de support aux tests Android

- Espresso documentation
- Espresso Samples

Vidéos

- Android Testing Support - Android Testing Patterns #1 (introduction)
- Android Testing Support - Android Testing Patterns #2 (onView view matching)
- Android Testing Support - Android Testing Patterns #3 (onData & adapter views)

Documentation pour les développeurs Android

- Best Practices for Testing
- Getting Started with Testing
- Testing UI for a Single App
- Building Instrumented Unit Tests
- Espresso Advanced Samples
- The Hamcrest Tutorial
- Hamcrest API and Utility Classes
- Test Support APIs

OFPPT
La Voie de l'Avenir

WEBFORCE
BE THE CHANGE

CHAPITRE 2

Créer des tests d'interface

1. Utilisation d'Espresso
2. **Utilisation d'UI Automator**

UI Automator

Le framework de test **UI Automator** est une API basée sur l'instrumentation et fonctionne avec le **runner** de test **AndroidJUnitRunner**. Il est bien adapté à l'écriture de tests automatisés de type boîte opaque, où le code de test ne dépend pas des détails de mise en œuvre internes de l'application cible.

Les principales caractéristiques du cadre de test **UI Automator** sont les suivantes :

- Une API permettant de récupérer des informations d'état et d'effectuer des opérations sur le périphérique cible ;
- Des API qui prennent en charge les tests d'interface utilisateur inter-applications.

Remarques

- Les fonctionnalités **d'UI Automator** et d'Espresso se ressemblent un peu, mais Espresso dispose de plus de mécanismes de synchronisation, c'est pourquoi il est préféré pour les tests d'interface utilisateur classique.

Accès à l'état du dispositif

Le framework de test **UI Automator** fournit une classe **UiDevice** pour accéder au périphérique sur lequel l'application cible est exécutée et effectuer des opérations sur ce dernier. Il est possible d'appeler ses méthodes pour accéder aux propriétés du périphérique telles que l'orientation actuelle ou la taille de l'affichage. La classe **UiDevice** permet également d'effectuer les actions suivantes :

1. Modifier la rotation du périphérique
2. Appuyer sur une touche matérielle, telle que "volume haut"
3. Appuyer sur les boutons Retour, Accueil ou Menu
4. Ouvrir l'écran de notification
5. Faire une capture d'écran de la fenêtre actuelle

Par exemple, pour simuler une pression sur le bouton Home, appeler la méthode **UiDevice.pressHome().**

API de l'Automator UI

Les API de **UI Automator** permettent d'écrire des tests solides sans avoir besoin de connaître les détails de l'implémentation de l'application ciblée. Elles permettent de capturer et de manipuler les composants de l'interface utilisateur dans plusieurs applications :

- **UiCollection** : énumère les éléments d'interface utilisateur d'un conteneur afin de les compter ou de cibler des sous-éléments en fonction de leur texte visible ou de leur propriété de description du contenu
- **UiObject** : représente un élément d'interface utilisateur qui est visible sur le périphérique
- **UiScrollable** : fournit un support pour la recherche d'éléments dans un conteneur d'interface utilisateur défilable
- **UiSelector** : représente une requête pour un ou plusieurs éléments d'interface utilisateur cibles sur un périphérique
- **Configurateur** : permet de définir les paramètres clés pour l'exécution des tests de UI Automator

Exemple

Par exemple, le code suivant montre comment écrire un script de test qui affiche le lanceur d'applications par défaut dans l'appareil :

```
device = UiDevice.getInstance(getInstrumentation()) ;
device.pressHome() ;
// Afficher le lanceur par défaut en recherchant un composant d'interface
// utilisateur
// qui correspond au contenu pour le bouton du lanceur.
UiObject allAppsButton = device
.findObject(new UiSelector().description("Apps")) ;

// Effectuer un clic sur le bouton pour charger le lanceur.
allAppsButton.clickAndWaitForNewWindow() ;
```

Configurer UI Automator

Avant de construire le test de l'interface utilisateur avec **UI Automator**, il faut s'assurer de configurer l'emplacement du code source du test et les dépendances du projet.

Dans le fichier **build.gradle** de l'application Android, il faut définir une référence de dépendance à la bibliothèque **UI Automator** :

```
dependencies {
  ...
  androidTestImplementation 'androidx.test.uiautomator:uiautomator:2.2.0'
}
```

Pour optimiser les tests de **l'Automator UI**, il faut d'abord inspecter les composants de l'interface utilisateur de l'application cible et s'assurer qu'ils sont accessibles.

Inspecter l'interface utilisateur sur un appareil

Avant de concevoir le test, inspecter les composants de l'interface utilisateur qui sont visibles sur l'appareil. Afin de garantir que les tests **UI Automator** peuvent accéder à ces composants, vérifier que ces composants ont des étiquettes de texte visibles, des valeurs **android:contentDescription**, ou les deux.

L'outil **uiautomatorviewer** fournit une interface visuelle pratique pour inspecter la hiérarchie de la mise en page et afficher les propriétés des composants de l'interface utilisateur qui sont visibles au premier plan sur le périphérique. Ces informations permettent de créer des tests plus fins à l'aide de **UI Automator**. Par exemple, il est possible de créer un sélecteur d'interface utilisateur qui correspond à une propriété visible spécifique.

Pour lancer l'outil **uiautomatorviewer** :

1. Lancer l'application cible sur un périphérique physique
2. Connecter l'appareil à la machine de développement
3. Ouvrir une fenêtre de terminal et naviguer dans le répertoire **<android-sdk>/tools/**
4. Exécuter l'outil avec cette commande : **$ uiautomatorviewer**

Inspecter l'interface utilisateur sur un appareil

Pour afficher les propriétés de l'interface utilisateur de l'application :

1. Dans l'interface **uiautomatorviewer**, cliquer sur le bouton Capture d'écran du dispositif.
2. Passer la souris sur la capture d'écran dans le panneau de gauche pour voir les composants de l'interface utilisateur identifiés par l'outil **uiautomatorviewer**. Les propriétés sont répertoriées dans le panneau inférieur droit et la hiérarchie de la disposition dans le panneau supérieur droit.
3. En option, cliquer sur le bouton Toggle NAF Nodes pour afficher les composants de l'interface utilisateur qui ne sont pas accessibles à **UI Automator**. Seules des informations limitées peuvent être disponibles pour ces composants.

Garantir l'accessibilité à l'activité

Le framework de test **UI Automator** est plus performant sur les applications qui ont implémenté les fonctionnalités d'accessibilité d'Android. En utilisant des éléments d'interface utilisateur de type **View**, ou une sous-classe de **View** du SDK, il est inutile d'implémenter la prise en charge de l'accessibilité, car ces classes remplissent déjà cette tâche de manière automatique.

Certaines applications, cependant, utilisent des éléments d'interface utilisateur personnalisés pour offrir une expérience utilisateur plus riche. Ces éléments n'offrent pas de prise en charge automatique de l'accessibilité. Si l'application contient des instances d'une sous-classe de **View** qui ne provient pas du SDK, il convient d'ajouter des fonctions d'accessibilité à ces éléments en suivant les étapes suivantes :

1. Créer une classe concrète qui étend **ExploreByTouchHelper**
2. Associer une instance de la nouvelle classe à un élément d'interface utilisateur personnalisé spécifique en appelant **setAccessibilityDelegate()**

Créer une classe de test UI Automator

La classe de test **UI Automator** doit être écrite de la même manière qu'une classe de test **JUnit 4**.

Ajouter l'annotation **@RunWith(AndroidJUnit4.class)** au début de la définition de la classe de test. Également spécifier la classe **AndroidJUnitRunner**, fournie dans **AndroidX** Test, comme l'exécuteur de test par défaut. Cette étape est décrite plus en détail dans Exécuter des tests **UI Automator** sur un appareil ou un émulateur.

Implémenter le modèle de programmation suivant dans la classe de test **UI Automator** :

1. Obtenir un objet **UiDevice** pour accéder au dispositif à tester, en appelant la méthode **getInstance**() et en lui passant un objet Instrumentation comme argument.
2. Obtenir un objet **UiObject** pour accéder à un composant de l'interface utilisateur qui est affiché sur le dispositif (par exemple, la vue actuelle au premier plan), en appelant la méthode **findObject().**
3. Simuler une interaction utilisateur spécifique à exécuter sur ce composant de l'interface utilisateur, en appelant une méthode **UiObject** ; par exemple, appeler **performMultiPointerGesture()** pour simuler un geste multi-touch, et **setText()** pour modifier un champ de texte. Il est possible de faire appel aux API des étapes 2 et 3 de manière répétée, si nécessaire, pour tester des interactions utilisateur plus complexes qui impliquent plusieurs composants de l'interface utilisateur ou des séquences d'actions utilisateur.
4. Vérifier que l'interface utilisateur reflète l'état ou le comportement attendu, une fois ces interactions utilisateur réalisées.

Ces étapes sont traitées plus en détail dans les rubriques suivantes.

Accéder aux composants de l'interface utilisateur

L'objet **UiDevice** est le principal moyen d'accéder à l'état de l'appareil et de le manipuler. Dans les tests, il est possible d'appeler des méthodes **UiDevice** pour vérifier l'état de diverses propriétés, telles que l'orientation actuelle ou la taille de l'affichage. Le test peut utiliser l'objet **UiDevice** pour effectuer des actions au niveau de l'appareil, comme forcer l'appareil à effectuer une rotation spécifique, appuyer sur les boutons matériels du D-pad et appuyer sur les boutons Home et Menu.

Il est recommandé de commencer le test à partir de l'écran d'accueil de l'appareil. À partir de l'écran d'accueil (ou d'un autre emplacement de départ choisi dans l'appareil), il est possible d'appeler les méthodes fournies par l'API **UI Automator** pour sélectionner et interagir avec des éléments spécifiques de l'interface utilisateur.

L'extrait de code suivant montre comment le test peut obtenir une instance de **UiDevice** et simuler un clic sur le bouton Home :

```
@RunWith(AndroidJUnit4.class)
@SdkSuppress(minSdkVersion = 18)
public class ChangeTextBehaviorTest {
  private static final String BASIC_SAMPLE_PACKAGE
  = "com.example.android.testing.uiautomator.BasicSample";
  private static final int LAUNCH_TIMEOUT = 5000;
  private static final String STRING_TO_BE_TYPED = "UiAutomator";
  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 l'apparition de l'application
    device.wait(Until.hasObject(By.pkg(BASIC_SAMPLE_PACKAGE).depth(0)),  LAUNCH_TIMEOUT);
    }
}
```

Accéder aux composants de l'interface utilisateur

Dans l'exemple, l'instruction **@SdkSuppress(minSdkVersion = 18)** permet de s'assurer que les tests ne s'exécuteront que sur les appareils équipés d'Android **4.3 (niveau 18 de l'API)** ou d'une version supérieure, comme l'exige le framework **UI Automator**.

Utiliser la méthode **findObject()** pour récupérer un **UiObject** qui représente une vue correspondant à un critère de sélection donné. Si nécessaire, il est possible de réutiliser les instances **UiObject** créées dans d'autres parties du test de l'application. Noter que le cadre de test **UI Automator** recherche une correspondance dans l'affichage actuel chaque fois que le test utilise une instance **UiObject** pour cliquer sur un élément de l'interface utilisateur ou interroger une propriété.

L'extrait suivant montre comment un test peut construire des instances **UiObject** représentant un bouton Annuler et un bouton OK dans une application.

```
@RunWith(AndroidJUnit4.class)
@SdkSuppress(minSdkVersion = 18)
public class ChangeTextBehaviorTest {

 private static final String BASIC_SAMPLE_PACKAGE
 = "com.example.android.testing.uiautomator.BasicSample";
 private static final int LAUNCH_TIMEOUT = 5000;
 private static final String STRING_TO_BE_TYPED = "UiAutomator";
 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 l'apparition de l'application
  device.wait(Until.hasObject(By.pkg(BASIC_SAMPLE_PACKAGE).depth(0)), LAUNCH_TIMEOUT);
 }
}
```

Spécifier un sélecteur

Pour accéder à un composant spécifique de l'interface utilisateur dans une application, il convient d'utiliser la classe **UiSelector**. Cette classe représente une requête pour des éléments spécifiques dans l'interface utilisateur actuellement affichée.

Si plus d'un élément correspondant est trouvé, le premier élément correspondant dans la hiérarchie de la présentation est renvoyé comme **UiObject** cible. Lors de la construction d'un **UiSelector**, il est possible d'enchaîner plusieurs propriétés pour affiner la recherche. Si aucun élément d'interface utilisateur correspondant n'est trouvé, une exception **UiAutomatorObjectNotFoundException** est levée.

La méthode **childSelector()** permet d'imbriquer plusieurs instances **d'UiSelector**. L'exemple de code suivant montre comment le test peut spécifier une recherche pour trouver la première **ListView** dans l'interface utilisateur actuellement affichée, puis une recherche dans cette **ListView** pour trouver un élément d'interface avec la propriété text Apps.

```
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, s'il existe.

if(okButton.exists() && okButton.isEnabled()) {

  okButton.click();

}
```

Spécifier un sélecteur

En tant que bonne pratique, lors de la spécification d'un sélecteur, il convient d'utiliser un **ID de ressource** (s'il est attribué à un élément d'interface utilisateur) plutôt qu'un élément de texte ou un descripteur de contenu.

Tous les éléments ne disposent pas d'un élément de texte (par exemple, les icônes dans une barre d'outils). Les sélecteurs de texte sont fragiles et peuvent entraîner l'échec des tests en cas de modifications mineures de l'interface utilisateur.

Ils peuvent également ne pas s'adapter à différentes langues ; les sélecteurs de texte peuvent ne pas correspondre aux chaînes de caractères traduites.

Il peut être utile de spécifier l'état de l'objet dans les critères de sélecteur.

Par exemple, si une liste de tous les éléments cochés doit être sélectionnée afin d'être décochée, appeler la méthode **checked()** avec l'argument **true**.

Effectuer des actions

Une fois que le test a obtenu un objet **UiObject**, il est possible d'appeler les méthodes de la classe **UiObject** pour effectuer des interactions utilisateur sur le composant de l'interface utilisateur représenté par cet objet. Les actions possibles sont les suivantes :

- **click()** : clique sur le centre des limites visibles de l'élément d'interface utilisateur
- **dragTo()** : fait glisser cet objet vers des coordonnées arbitraires
- **setText()** : définit le texte d'un champ éditable, après avoir effacé le contenu du champ. Inversement, la méthode **clearTextField()** efface le texte existant dans un champ éditable
- **swipeUp()** : effectue l'action de glisser vers le haut sur **l'UiObject**. De même, les méthodes **swipeDown(), swipeLeft()** et **swipeRight()** effectuent les actions correspondantes

Le cadre de test **UI Automator** permet d'envoyer une intention ou de lancer une activité sans utiliser de commandes **shell**, en obtenant un objet **Context** via **getContext()**

Effectuer des actions

L'extrait suivant montre comment un test peut utiliser un Intent pour lancer l'application testée. Cette approche est utile pour tester uniquement l'application calculatrice, sans se soucier du lanceur :

```
public void setUp() {
...
  // Lancer une application de calculatrice simple :
  Context context = getInstrumentation().getContext();
  Intent intent = context.getPackageManager()
  .getLaunchIntentForPackage(CALC_PACKAGE);
  intent.addFlags(Intent.FLAG_ACTIVITY_CLEAR_TASK);
  // Effacer toutes les instances précédentes
  context.startActivity(intent);
  device.wait(Until.hasObject(By.pkg(CALC_PACKAGE).depth(0)), TIMEOUT);
}
```

Effectuer des actions sur les collections

Utiliser la classe **UiCollection** pour simuler les interactions de l'utilisateur sur une collection d'éléments (par exemple, les chansons d'un album de musique ou une liste d'e-mails dans une boîte de réception).

Pour créer un objet **UiCollection**, spécifier un sélecteur **UiSelector** qui recherche un conteneur d'interface utilisateur ou une enveloppe d'autres éléments d'interface utilisateur enfants, comme une vue de disposition qui contient des éléments d'interface utilisateur enfants.

L'extrait de code suivant montre comment un test peut construire une **UiCollection** pour représenter un album vidéo affiché dans un **FrameLayout** :

Effectuer des actions sur les collections

```
UiCollection videos = new UiCollection(new UiSelector()
.className("android.widget.FrameLayout"));
// Récupérer le nombre de vidéos dans cette collection :
int count = videos.getChildCount(new UiSelector()
.className("android.widget.LinearLayout"));
// Rechercher une vidéo spécifique et simuler un clic de l'utilisateur sur celle-ci.
UiObject video = videos.getChildByText(new UiSelector()
.className("android.widget.LinearLayout"), "Cute Baby Laughing");
video.click();
// Simuler la sélection d'une case à cocher associée à la vidéo.
UiObject checkBox = video.getChild(new UiSelector()
.className("android.widget.Checkbox"));
if(!checkBox.isSelected()) checkbox.click();
```

Effectuer des actions sur des vues défilantes

La classe **UiScrollable** permet de simuler un défilement vertical ou horizontal sur un écran. Cette technique est utile lorsqu'un élément de l'interface utilisateur est positionné hors de l'écran et nécessite un défilement pour le faire apparaître.

L'extrait de code suivant montre comment simuler le défilement du menu Paramètres et un clic sur l'option À propos de la tablette :

```
UiScrollable settingsItem = new UiScrollable(new UiSelector()
.className("android.widget.ListView"));
UiObject about = settingsItem.getChildByText(new UiSelector()
.className("android.widget.LinearLayout"), "About tablet");
about.click();
```

Vérifier les résultats

InstrumentationTestCase étend **TestCase**, ce qui permet d'utiliser les méthodes **Assert standard** de JUnit pour vérifier que les composants de l'interface utilisateur de l'application renvoient les résultats attendus.

L'extrait suivant montre comment le test peut localiser plusieurs boutons dans une application de calculatrice, cliquer dessus dans l'ordre, puis vérifier que le résultat correct s'affiche.

```
private static final String CALC_PACKAGE = "com.myexample.calc";
public void testTwoPlusThreeEqualsFive() {
 // Saisir une équation : 2 + 3 = ?
 device.findObject(new UiSelector()
 .packageName(CALC_PACKAGE).resourceId("two")).click();
 device.findObject(new UiSelector()
 .packageName(CALC_PACKAGE).resourceId("plus")).click();
 device.findObject(new UiSelector()
 .packageName(CALC_PACKAGE).resourceId("three")).click();
 device.findObject(new UiSelector()
 .packageName(CALC_PACKAGE).resourceId("equals")).click();

 // Vérifier le résultat = 5
 UiObject result = device.findObject(By.res(CALC_PACKAGE, "result"));
 assertEquals("5", result.getText());
}
```

Exécuter des tests UI Automator sur un appareil ou un émulateur

Il est possible d'exécuter des tests **UI Automator** à partir d'Android Studio ou de la ligne de commande. Veiller à spécifier **AndroidJUnitRunner** comme l'exécuteur d'instrumentation par défaut dans le projet.