**DAA - Développement d'applications Android**

# Activités, Fragments et Services

*16 November 2025*

## Table des matières

# 1 Activités

Une activité est la brique de base de toute application Android. Elle représente un seul écran avec une interface utilisateur. Une application va souvent être composée de plusieurs activités, chacune correspondant à une fonctionnalité ou une section différente de l'application.

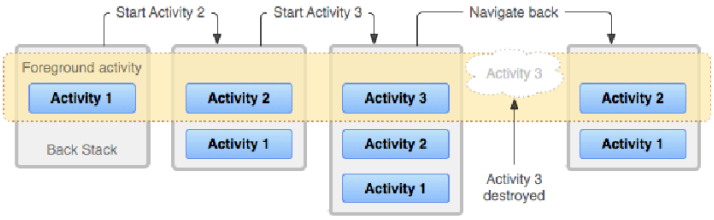


*Fig. 1. – Capture des slides du cours – Passage entre activités*

## 1.1 Déclaration

Une activité doit être déclarée dans le fichier `AndroidManifest.xml` de l'application. Voici un exemple de déclaration d'une activité nommée `MainActivity` :

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  package="ch.heigvd.iict.myapplication">

  <application>
    <activity
      android:name=".MainActivity"
      android:exported="true"
      android:label="@string/app_name" />
  </application>

</manifest>
```

```
package ch.heigvd.iict.myapplication

import android.app.Activity
import android.os.Bundle

class MainActivity : Activity() {

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(android.R.layout.list_content)
  }

}
```

*Fig. 2. – Capture des slides du cours – Déclaration d'une activité dans le manifeste*

Cette déclaration fait le lien avec la classe implémentant l'activité. L'implémentation en Kotlin hérite toujours de `AppCompatActivity` (et non directement de `Activity`) pour assurer la compatibilité entre différentes versions d'Android.

## 1.2 Cycle de vie

Le cycle de vie d'une activité est géré par le système Android et comprend plusieurs états, tels que `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, et `onDestroy()`. Chaque méthode correspond à une étape spécifique du cycle de vie de l'activité.

Les applications ne contrôlent pas directement leur cycle de vie (inversion de contrôle). Le système Android appelle ces méthodes en fonction des interactions de l'utilisateur et des changements dans l'état de l'application.

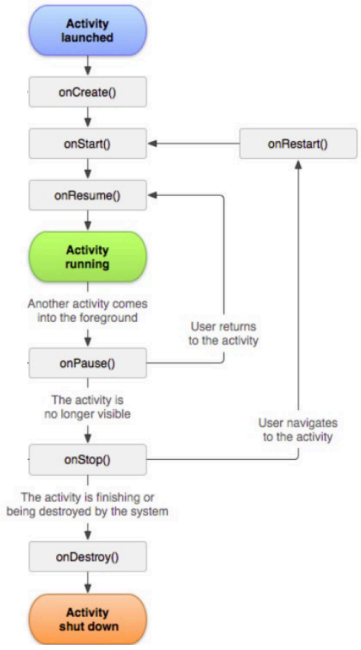

*Fig. 3. – Capture des slides du cours – Cycle de vie d'une activité*

En fonction de leur cycle de vie, les Activités peuvent être dans plusieurs états:

- **Active** : L'activité est au premier plan et l'utilisateur peut interagir avec elle. Elle est au sommet de la pile d'activités.
- **En pause** : L'activité est toujours visible mais n'est pas au premier plan (par exemple, une boîte de dialogue peut être affichée au-dessus d'elle). Elle conserve son état et ses ressources. L'utilisateur ne peut pas interagir avec elle.
- **Arrêtée** : L'activité n'est pas visible pour l'utilisateur. Elle reste chargée en mémoire, mais elle peut être tuée par le système si la mémoire est nécessaire pour d'autres applications.
- **Inactive** : L'activité n'est pas en cours d'exécution. Elle a été détruite et doit être recréée si l'utilisateur souhaite y revenir.

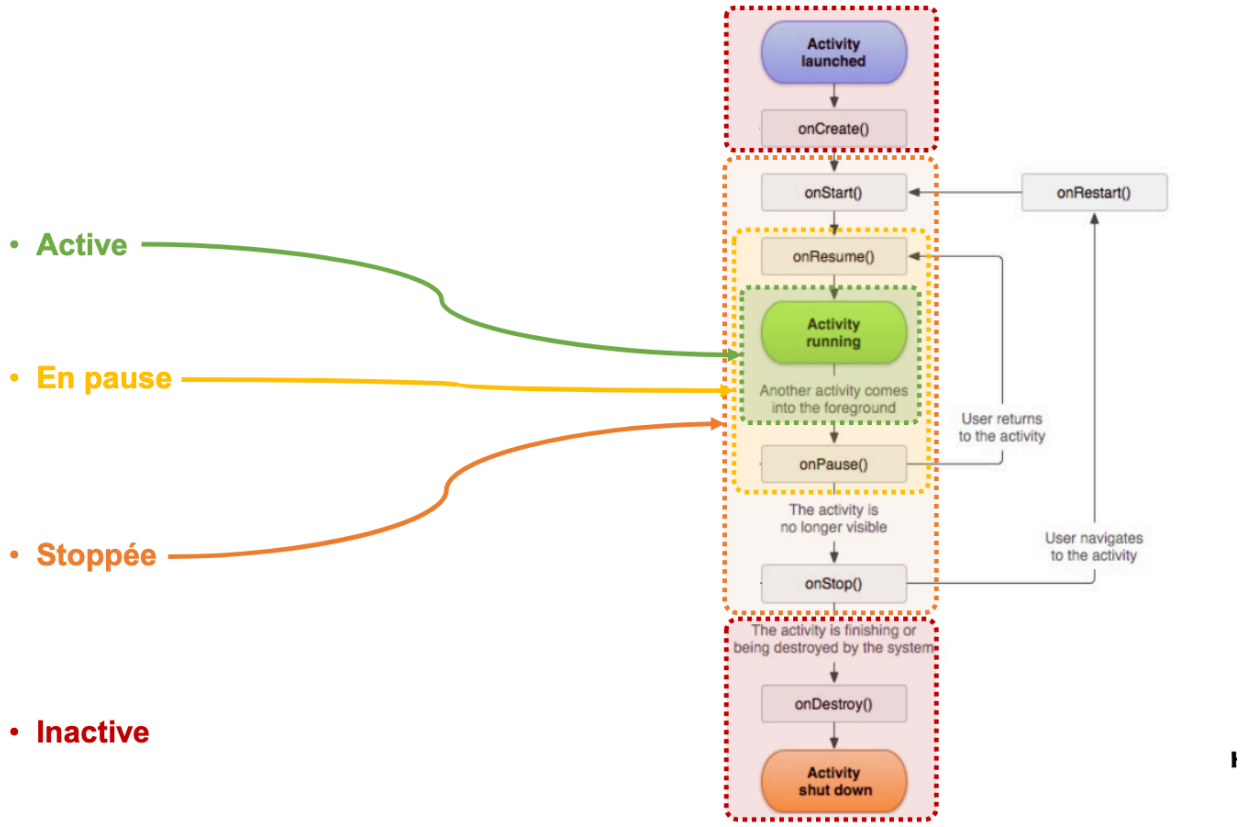


*Fig. 4. – Capture des slides du cours – États d'une activité*

> **i Info**
>
> Les états en pause et stoppée sont souvent confondus. Dans le cas où les applications sont en plein écran, la différence est minime. Cependant, dans le cas d'applications multi-fenêtres (split-screen), une application peut être visible mais pas au premier plan (en pause).

### 1.3 Implémentation minimale

L'implémentation minimum d'une activité affichant une interface graphique nécessite :

1. L'override de la méthode `onCreate()` pour l'initialisation
2. L'appel à `super.onCreate()` (obligatoire)
3. L'appel à `setContentView()` avec l'identifiant d'un layout pour afficher l'interface

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
    }
}
```

## 1.4 Interaction avec la GUI

### *1.4.1 Linkage des vues*

Pour interagir avec les éléments graphiques, il faut récupérer des références vers ceux-ci. Deux approches sont possibles :

**Méthode manuelle avec `findViewById()` :**

```
val myTextView = findViewById<TextView>(R.id.my_textview)
myTextView.text = "Mon nouveau texte !"
```

**View Binding (recommandé)** : Génération automatique d'une classe de binding pour chaque layout. À activer dans `build.gradle` :

```
android {
    buildFeatures {
        viewBinding = true
    }
}
```

Utilisation :

```
private lateinit var binding: ActivityMainBinding

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    binding.myButton.setOnClickListener {
        // Gestion du clic
    }
}
```

### *1.4.2 Gestion des événements*

Les événements utilisateur (clics, etc.) peuvent être gérés de plusieurs façons :

```
// Approche programmatique (recommandée)
myButton.setOnClickListener {
    // Code à exécuter au clic
}

// Approche XML (dépréciée)
// Dans le layout : android:onClick="myMethod"
// Dans l'activité :
fun myMethod(view: View) {
    // Code à exécuter
}
```

## 1.5 Lancement et navigation

### *1.5.1 Les Intents*

Les Intents sont le mécanisme utilisé pour démarrer des activités. Il en existe deux types :

**Intent explicite** : Spécifie directement la classe de destination (navigation interne)

```
val intent = Intent(this, MySecondActivity::class.java)
startActivity(intent)
```

**Intent implicite** : Décrit une action générique, le système choisit l'application appropriée

```
val intent = Intent(Intent.ACTION_VIEW)
intent.data = Uri.parse("http://www.heig-vd.ch")
startActivity(intent)
```

### 1.5.2 Passage de paramètres

Les Intents permettent de passer des données entre activités via des paires clé/valeur :

```
// Envoi
val intent = Intent(this, MySecondActivity::class.java)
intent.putExtra("NAME_KEY", "Toto")
intent.putExtra("AGE_KEY", 21)
startActivity(intent)

// Réception
val name = intent.getStringExtra("NAME_KEY")
val age = intent.getIntExtra("AGE_KEY", -1) // -1 = valeur par défaut
```

### 1.5.3 Récupération de résultat

Pour obtenir un résultat d'une activité, utiliser l'approche moderne avec les Activity Result Contracts :

```
// Définition du contrat
class PickNameContract : ActivityResultContract<Void?, String?>() {
    override fun createIntent(context: Context, input: Void?) =
        Intent(context, MySecondActivity::class.java)

    override fun parseResult(resultCode: Int, result: Intent?): String? {
        if (resultCode != Activity.RESULT_OK) return null
        return result?.getStringExtra("NAME_KEY")
    }
}

// Enregistrement et lancement
private val getName = registerForActivityResult(PickNameContract()) { name ->
    // Traitement du résultat
    Log.d("MainActivity", "Nom reçu : $name")
}

// Lancement
getName.launch(null)
```

L'activité appelée renvoie le résultat avec :

```
val data = Intent()
data.putExtra("NAME_KEY", fieldValue)
setResult(RESULT_OK, data)
finish()
```

## 1.6 Sauvegarde et restauration d'état

Lorsque le système détruit temporairement une activité (rotation d'écran, manque de mémoire), il faut sauvegarder son état :

```
// Sauvegarde
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putInt("COUNTER_VALUE", counter)
}

// Restauration
override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
    counter = savedInstanceState.getInt("COUNTER_VALUE", 0)
}
```

> ⚠ **Warning**

Le système sauvegarde automatiquement l'état de certains widgets possédant un identifiant. Pour les autres données, la sauvegarde doit être gérée manuellement.

### 1.7 Terminer une activité

Une activité peut être terminée de deux manières :

1. **L'utilisateur appuie sur « Back »** : Bouton physique, virtuel, ou geste de swipe selon le modèle
2. **Appel programmatique** : `finish()` pour terminer l'activité courante

L'événement « Back » peut être intercepté :

```
override fun onCreate() {
    onBackPressedDispatcher.addCallback {
        // Traitement personnalisé
        // Ne pas empêcher la propagation !
    }
}
```

# 2 Fragments

## 2.1 Introduction

Les fragments sont des composants modulaires et réutilisables d'interface utilisateur introduits pour améliorer la flexibilité des applications Android. Ils représentent une portion de l'interface utilisateur dans une activité hôte et possèdent leur propre cycle de vie.

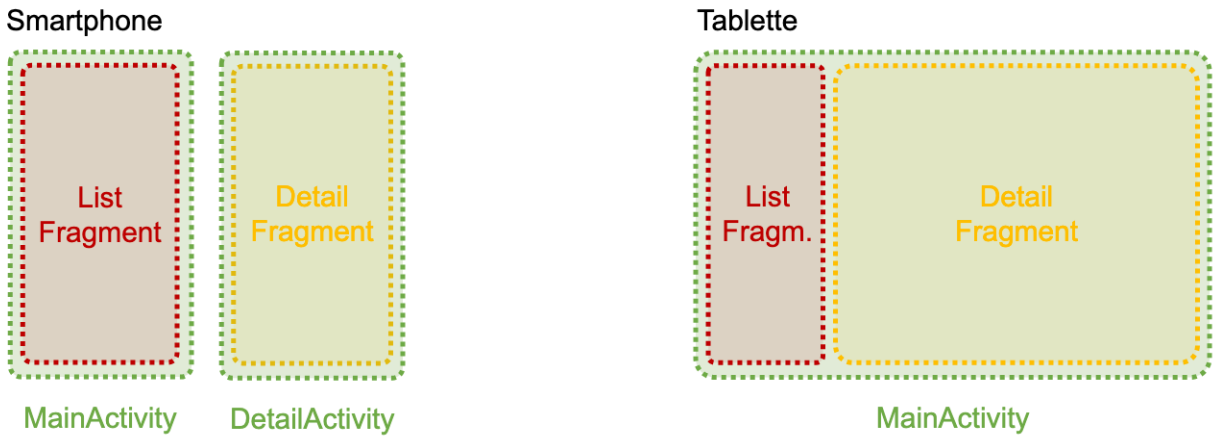


*Fig. 5. – Capture des slides du cours – Fragments sur smartphone et tablette*

## 2.2 Cycle de vie

Les fragments possèdent un cycle de vie distinct mais lié à celui de leur activité hôte. Les principales méthodes de callback sont `onCreate()`, `onCreateView()`, `onViewCreated()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, et `onDestroy()`.

La vue d'un fragment possède son propre cycle de vie, géré indépendamment de celui du fragment lui-même.

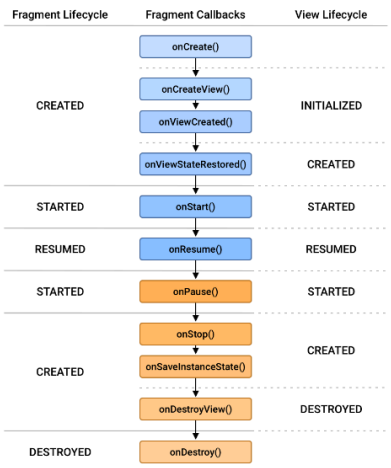


*Fig. 6. – Capture des slides du cours – Cycle de vie d'un fragment*

## 2.3 Gestion

Les fragments sont gérés par le `FragmentManager` qui permet de :

- Effectuer des transitions entre fragments (ajout, remplacement, retrait)
- Gérer une pile de fragments (back stack)
- Gérer les états et transactions des fragments

Les fragments sont placés dans un conteneur ( `FragmentContainerView` ) au sein du layout de l'activité hôte.

### 2.4 Communication

L'échange de données entre fragments et activités peut se faire :

- **Activité → Fragment** : L'activité peut passer des arguments lors de la création du fragment
- **Fragment → Activité** : Via des interfaces callback ou des ViewModels (recommandé)
- **Fragment ↔ Fragment** : En utilisant des ViewModels partagés avec l'activité hôte

# 3 Services

## 3.1 Types de services

Android propose trois types de services :

- **Foreground** : Services de premier plan liés à une notification visible. Utilisés pour des tâches dont l'utilisateur doit être conscient (lecteur audio, téléchargement).
- **Background** : Services d'arrière-plan sans interface utilisateur, limités dans le temps. Utilisés pour des tâches courtes comme une synchronisation.
- **Bounded** : Services liés à un composant (généralement une activité). Détruits lorsque plus aucun composant n'y est lié.

> ⚠ **Warning**
>
> Les services s'exécutent dans le thread principal (UI Thread). Il faut donc créer un thread séparé pour les opérations longues afin d'éviter de bloquer l'interface utilisateur.

## 3.2 Cycle de vie

Les services possèdent leur propre cycle de vie avec les méthodes principales :

- `onCreate()` : Initialisation du service
- `onStartCommand()` : Pour les services foreground/background
- `onBind()` : Pour les services bounded
- `onDestroy()` : Nettoyage des ressources

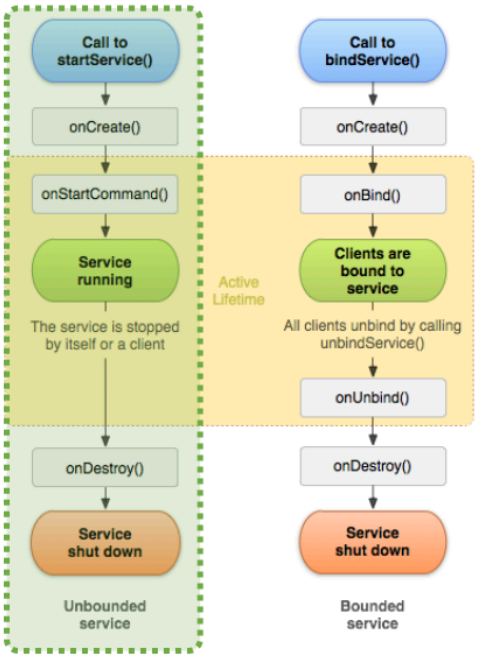


*Fig. 7. – Capture des slides du cours – Cycle de vie des services*

## 3.3 Limitations

Depuis Android 8.0 (API 26), de nombreuses restrictions ont été introduites :

- Les services background sont fortement limités en arrière-plan
- Les services foreground nécessitent une notification obligatoire
- Depuis API 31, impossible de lancer un service foreground depuis l'arrière-plan

> i **Info**

Pour la plupart des cas d'usage, il est recommandé d'utiliser `WorkManager` (pour les tâches background) ou `ViewModel` + `LiveData` (pour la communication activité-service) plutôt que les services traditionnels.

# 4 Permissions

## 4.1 Niveaux de permissions

Android propose trois niveaux de permissions :

- **À l'installation** : Accordées automatiquement lors de l'installation, présentent peu de risques pour la vie privée (ex: accès Internet).
- **À l'exécution** : Permissions « dangereuses » nécessitant l'accord explicite de l'utilisateur via une popup (ex: localisation, caméra, contacts).
- **Spéciales** : Réservées au système ou au constructeur du téléphone.

## 4.2 Bonnes pratiques

Trois principes fondamentaux à respecter :

- **Contrôle** : Laisser l'utilisateur choisir d'accorder ou non les permissions. L'application doit rester fonctionnelle même sans certaines permissions.
- **Transparence** : Informer clairement l'utilisateur des données utilisées et de leur finalité.
- **Minimisation** : Ne demander que les permissions strictement nécessaires au fonctionnement de l'application.

## 4.3 Implémentation

Pour les permissions à l'exécution :

1. Déclarer la permission dans le `AndroidManifest.xml`
2. Vérifier si la permission est déjà accordée avec `checkSelfPermission()`
3. Demander la permission avec `registerForActivityResult()` et `RequestPermission()`
4. Gérer la réponse de l'utilisateur dans le callback

> ⚠ **Warning**
>
> Les méthodes nécessitant des permissions dangereuses doivent être annotées avec `@SuppressLint("MissingPermission")` pour éviter les erreurs de Lint, et ne doivent être appelées qu'après vérification de la permission.