COURS DE DÉVELOPPEMENT D'APPLICATIONS MOBILES

GLO 418 / IC4
Année académique : 2023-2024

UMa
ENSPM
INFOTEL

Touza Isaac
isaac_touza@outlook.fr

Informations générales

- **Code UE : GLO418**
- **Intitulé de L'UE : Développement d'Applications Mobile**
- **Crédit : 3**
- **Durée : 45h**
  - **CM : 10 h**
  - **TP: 15h**
  - **TPE: 5h**
  - **TD : 15h**
- **Evaluations :**
  - CC (théorique ou pratique): 2h
  - Examen (théorique ou pratique) : 2h
  - Rattrapage (théorique ou pratique): 2h

Informations générales

- **Références :**

- **Antonio Leiva,** *Kotlin for Android Developers. Learn Kotlin the easy way while developing an Android App.* 22017-06-20
- **Pierre Nerzic** , *Programmation mobile avec Android.* IUT de Lannion. février-mars 2021.
- **Tutorials Point (I) Pvt. Ltd**, Kotlin. 2019
- **Peter Späth**, *Pro Android with Kotlin. Developing Modern Mobile Apps. Leipzig,* Germany ISBN-13 (pbk): 978-1-4842-3819-6.

Objectifs du cours

Objectif général:

Apprendre à développer des applications mobiles avec Kotlin.

Objectifs spécifiques :

- Maîtriser les bases de Kotlin
- Comprendre la Programmation Orientée Objet avec Kotlin
- Mettre en œuvre la programmation fonctionnelle
- S'initier aux applications Android sous Kotlin
- Utiliser efficacement jetpack compose
- Intégrer le langage Kotlin dans un projet Java existant

Près-requis et consignes

Près requis :

- Maitriser un langage de programmation orienté objet (Java, C#, C++)
- Programmer en HTML

Consignes :

- Assister à tous les cours
- Etre attentifs et actifs pendant le cours
- Faire tous les exercices de TD et TP
- Refaire plusieurs fois les exercices d'applications
- Ne manquez jamais un TP

Recommandations

Outils logiciels:

- JDK et JRE
- Android studio
- IntelliJ IDEA

Plateformes

- YouTube
- Stackoverflow
- GitHub

Plan du cours

Séance 1 : Environnement de développement

Séance 2 : Introduction au langage Kotlin

Séance 3 – 6 : Programmation impérative avec kotlin

- Conception d'interface d'utilisateur
- Vie d'une application
- Liste
- Ergonomie

Séance 7 – 12 : Jetpack compose

- Introduction
- Les bases du compose
- Gestion des données avec compose
- Widgets, Interfaces personnalisées et composables avancées
- La navigation
- Test et gestion de configuration

Séance 13 : SQLite

Séance 14 : Firebase

SÉANCE 1

ENVIRONNEMENT DE DÉVELOPPEMENT

Cette matière présente la programmation d'applications natives sur Android avec Kotlin. Il y aura 14 semaines de cours, chacune comptant 2h CM et 3h TP.

Cette semaine nous allons découvrir l'environnement de développement Android :

- Le SDK Android et Android Studio
- Création d'une application simple
- Communication avec un smartphone ou une tablette

Android: Historique et définition

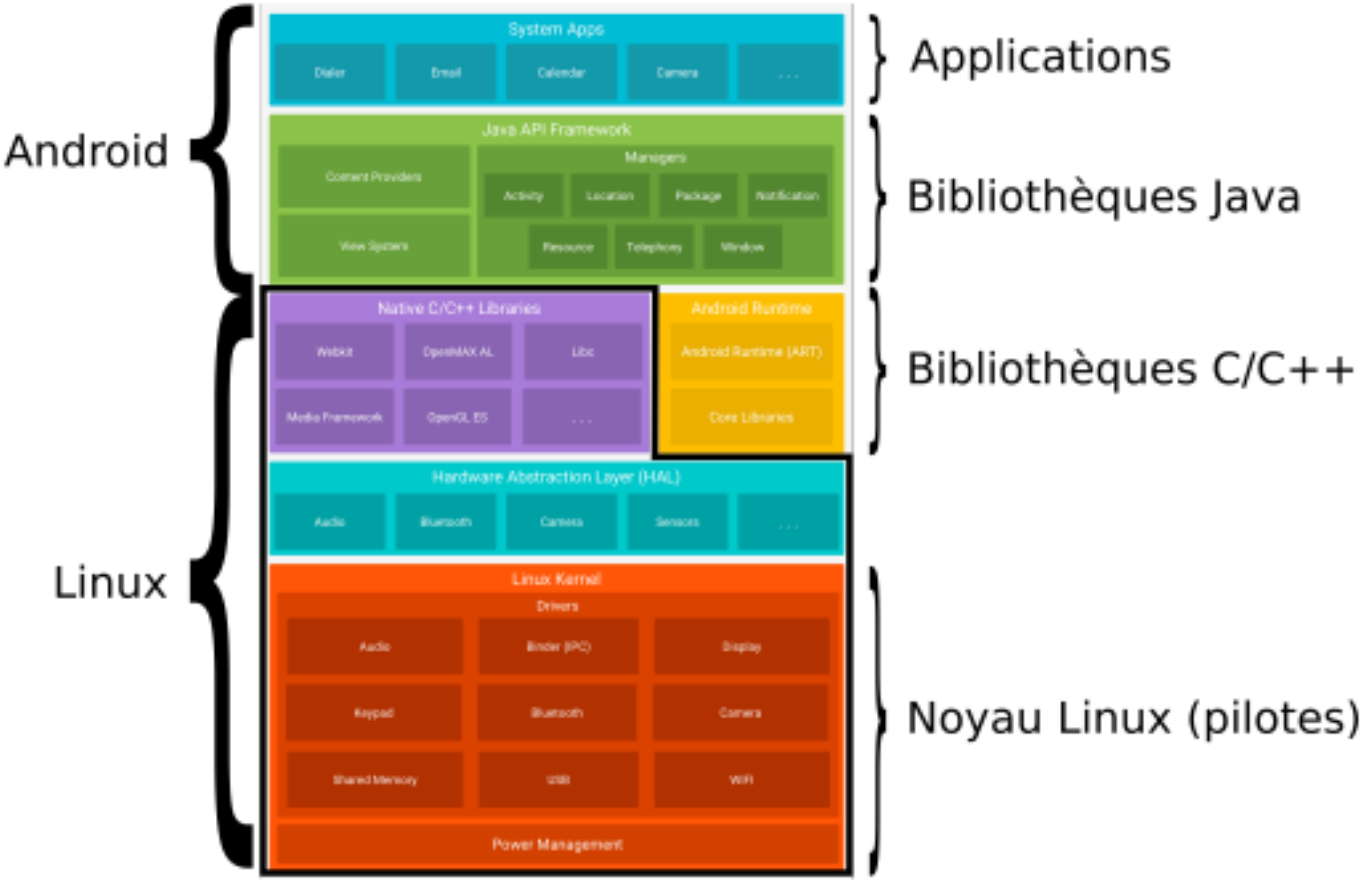
- Né en 2004, racheté par Google en 2005, publié en 2007, version 1.5
- De nombreuses versions depuis. On en est à la version 14 (2023) et l'API 34.
- La version 13 est le numéro pour le grand public, et les versions d'API sont pour les développeurs. Exemples:
  - 4.1 JellyBean = API 16,
  - 6.x Marshmallow = API 23,
  - 9.x Pie = API 28

Android est donc un système complet pour smartphones et tablettes

- Gestion matérielle : système d'exploitation Linux sous-jacent
- API de programmation : interfaces utilisateur, outils. . .
- Applications : navigateur, courrier. . .

Composants d'Android

Android est une surcouche au dessus d'un système Linux.



Distribution des versions d’API

Une **API** (Application Programming Interface) est un ensemble de bibliothèques de classes pour programmer des applications.

Distribution des versions d'API

Version	SDK / API level	Version code	Codename	Cumulative usage [1]	Year [4]
Android 14	Level 34	UPSIDE_DOWN_CAKE	Upside Down Cake [2]	—	2023
	▪ targetSdk will need to be 34+ for new apps and app updates by August 31, 2024.				
Android 13	Level 33	TIRAMISU	Tiramisu [2]	38.8%	2022
	▪ targetSdk must be 33+ for new apps and app updates since August 31, 2023.				
Android 12	Level 32 Android 12L	S_V2	Snow Cone [2]	56.8%	
	Level 31 Android 12	S			2021
Android 11	Level 30	R	Red Velvet Cake [2]	74.2%	2020
Android 10	Level 29	Q	Quince Tart [2]	83.3%	2019
Android 9	Level 28	P	Pie	89.7%	2018
Android 8	Level 27 Android 8.1	O_MR1	Oreo	91.9%	2017
	Level 26 Android 8.0	O		94.9%	
Android 7	Level 25 Android 7.1	N_MR1	Nougat	95.3%	2016
	Level 24 Android 7.0	N		96.8%	
Android 6	Level 23	M	Marshmallow	98.2%	2015
Android 5	Level 22 Android 5.1	LOLLIPOP_MR1	Lollipop	99.1%	
	Level 21 Android 5.0	LOLLIPOP, L		99.4%	2014

https://apilevels.com/

Distribution des versions d'API

Remarques :

- Chaque API apporte des fonctionnalités supplémentaires. Il y a compatibilité ascendante. Certaines fonctionnalités deviennent dépréciées au fil du temps, mais restent généralement disponibles.
- On souhaite toujours programmer avec la dernière API (fonctions plus complètes et modernes), mais les utilisateurs ont souvent des smartphones plus anciens, qui n'ont pas cette API.
- Or Android ne propose aucune mise à jour majeure. Les smartphones restent toute leur vie avec l'API qu'ils ont à la naissance.
- Les développeurs doivent donc choisir une API qui correspond à la majorité des smartphones existant sur le marché.

Applications Android.

Actuellement, les applications sont :

- « **natives** », c'est à dire programmées en Java, C++, Kotlin, compilées et fournies avec leurs données sous la forme d'une archive Jar (fichier APK). C'est ce qu'on étudiera ici.
- « **web app** », c'est une application pour navigateur internet, développée en HTML5, CSS3, JavaScript, dans un cadre logiciel (framework) tel que Node.js, Angular ou React.
- « **hybrides** », elles sont développées dans un framework comme Ionic, Flutter, React Native. . . Ces frameworks font abstraction des particularités du système : la même application peut tourner à l'identique sur différentes plateformes (Android, iOS, Windows, Linux. . .).

NB : La charge d'apprentissage est la même.

Applications Android.

Actuellement, les applications sont :

- « **natives** », c'est à dire programmées en Java, C++, Kotlin, compilées et fournies avec leurs données sous la forme d'une archive Jar (fichier APK). C'est ce qu'on étudiera ici.
- « **web app** », c'est une application pour navigateur internet, développée en HTML5, CSS3, JavaScript, dans un cadre logiciel (framework) tel que Node.js, Angular ou React.
- « **hybrides** », elles sont développées dans un framework comme Ionic, Flutter, React Native. . . Ces frameworks font abstraction des particularités du système : la même application peut tourner à l'identique sur différentes plateformes (Android, iOS, Windows, Linux. . .).

NB : La charge d'apprentissage est la même.

Applications Android.

Une application native Android est composée de :

- **Code sources** (Java ou Kotlin) compilés pour une machine virtuelle appelée « ART», amélioration de l'ancienne machine « Dalvik » (versions ≤ 4.4).
- Fichiers appelés **ressources** :
  - format XML : interface, textes. . .
  - format PNG : icônes, images. . .
- **Manifeste** = description du contenu du logiciel
  - version minimale du smartphone,
  - fichiers présents dans l'archive avec leur signature,
  - demandes d'autorisations, durée de validité, etc.

Tout cet ensemble est géré à l'aide d'un IDE (environnement de développement) appelé Android Studio qui s'appuie sur un ensemble logiciel (bibliothèques, outils) appelé SDK Android.

Android Studio

Le logiciel Android Studio est un logiciel de programmation (IDE) des applications mobiles.

Elle offre :

- un éditeur de sources et de ressources
- des outils de compilation : gradle
- des outils de test et de mise au point

Pour commencer à créer des applications mobiles, il faut installer Android Studio selon la procédure expliquée via cette adresse :

https://developer.android.com/studio/index.html

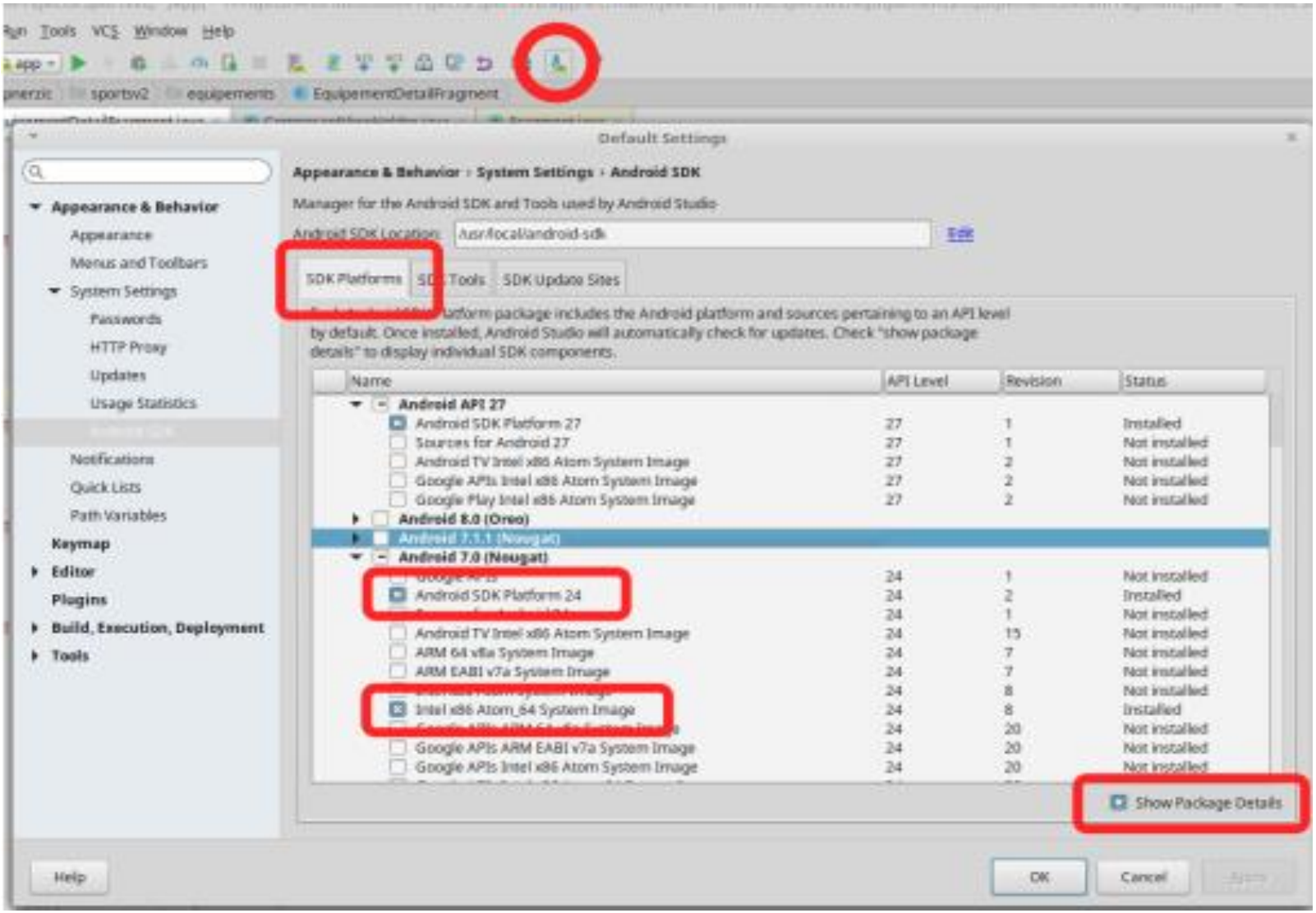
SDK

Le Software Development Kit (SDK) contient :

- ❖ Les librairies Java pour créer des logiciels
- ❖ Les outils de mise en boîte des logiciels
- ❖ AVD : un émulateur de tablettes pour tester les applications
- ❖ ADB : un outil de communication avec les vraies tablettes

L'installation de SDK, se fait soit automatiquement avec Android Studio, soit faire une installation personnalisée. En général, vous pouvez choisir ce que vous voulez ajouter au SDK (version des librairies, versions des émulateurs de smartphones), à l'aide du **SDK Manager**. **C'est le gestionnaire du SDK, une application qui permet de choisir les composants à installer et mettre à jour.**

SDK


Run Tools VCS Window Help
app
sportsv2
equipements
EquipementDetailFragment
Default Settings
Appearance & Behavior
Appearance
Menus and Toolbars
System Settings
Passwords
HTTP Proxy
Updates
Usage Statistics
Notifications
Quick Lists
Path Variables
Keymap
Editor
Plugins
Build, Execution, Deployment
Tools
Appearance & Behavior › System Settings › Android SDK
Manager for the Android SDK and Tools used by Android Studio
Android SDK Location: /usr/local/android-sdk
Edit
SDK Platforms
SDK Tools
SDK Update Sites
by default. Once installed, Android Studio will automatically check for updates. Check "show package details" to display individual SDK components.
Name
API Level
Revision
Status
Android API 27
Android SDK Platform 27
Sources for Android 27
Android TV Intel x86 Atom System Image
Google APIs Intel x86 Atom System Image
Google Play Intel x86 Atom System Image
Android 8.0 (Oreo)
Android 7.1.1 (Nougat)
Android 7.0 (Nougat)
Android SDK Platform 24
Android TV Intel x86 Atom System Image
ARM 64 v8a System Image
ARM EABI v7a System Image
Intel x86 Atom_64 System Image
Google APIs ARM EABI v7a System Image
Google APIs Intel x86 Atom System Image
Show Package Details
Help
OK
Cancel

SDK

Le gestionnaire permet de

- ✓ choisir les versions à installer, ex. : Android 11 (API 30), Android 7.0 (API 24) , …
- ✓ Choisir celles qui correspondent aux tablettes qu'on vise, mais tout n'est pas à installer : il faut cocher **Show Package Details**, puis choisir élément par élément. Seuls ceux-là sont indispensables :
- ❑ Android SDK Platform
- ❑ Intel x86 Atom_64 System Image

Le reste est facultatif (Google APIs, sources, exemples et docs)

SDK

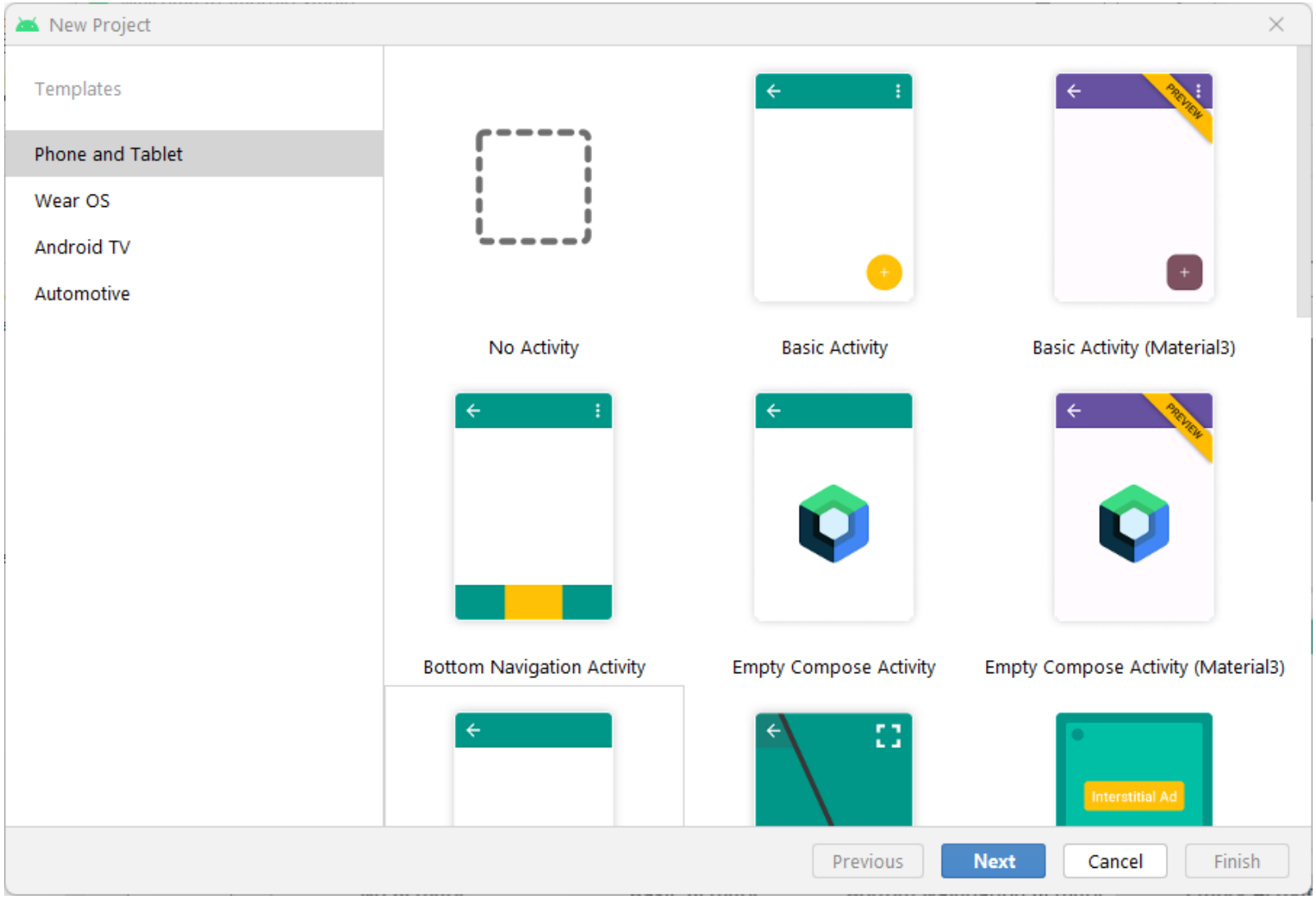
Le dossier SDK contient de sous-dossiers suivants:

- **SDK Tools** : indispensable, contient le gestionnaire,
- **SDK Platform-tools** : indispensable, contient adb,
- **SDK Platform** : indispensable, contient les librairies,
- **System images** : pour créer des AVD,
- **Android Support** : divers outils pour créer des applications,
- Exemples et sources.

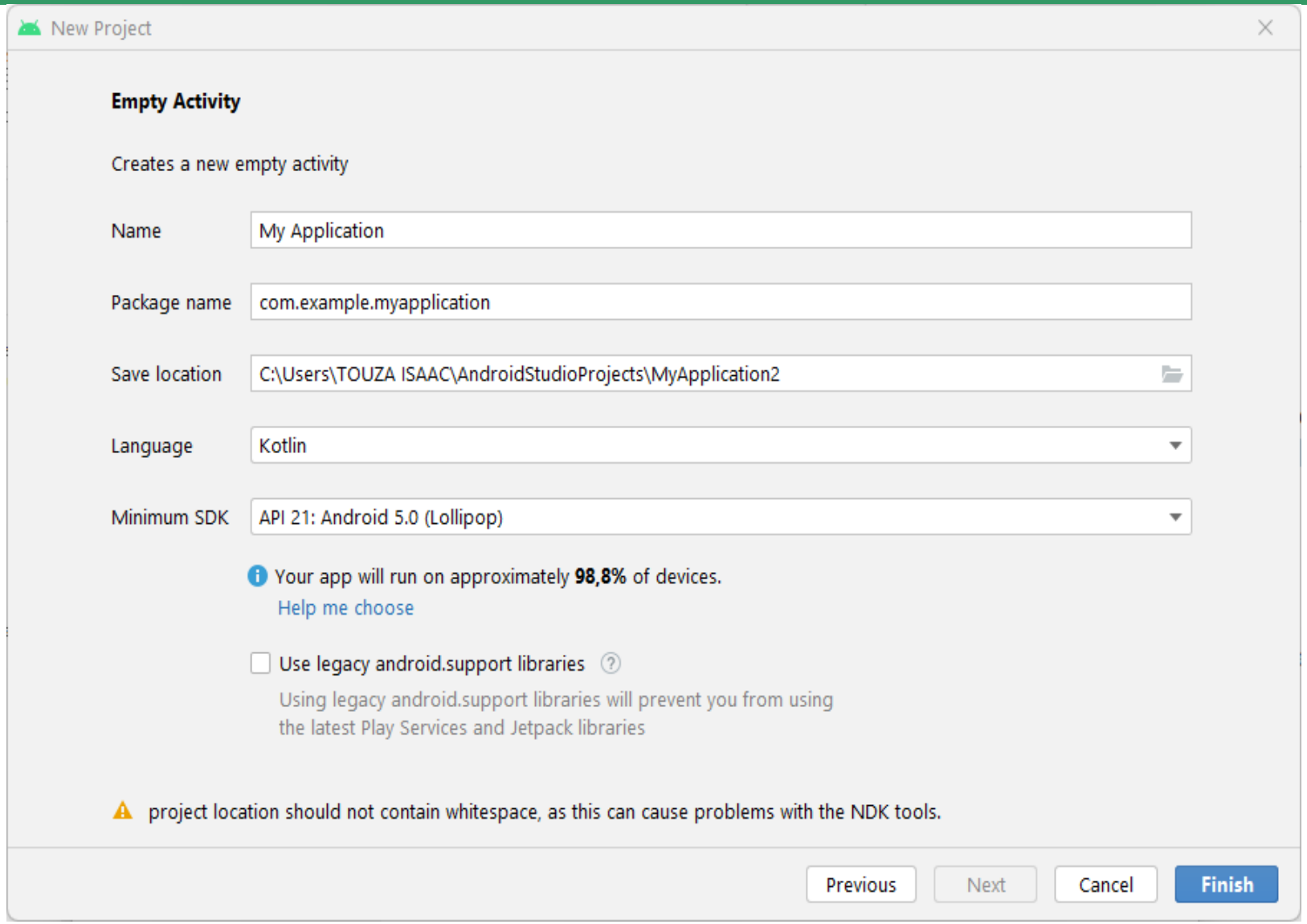
Première application Android

Android Studio contient un assistant de création d'applications :

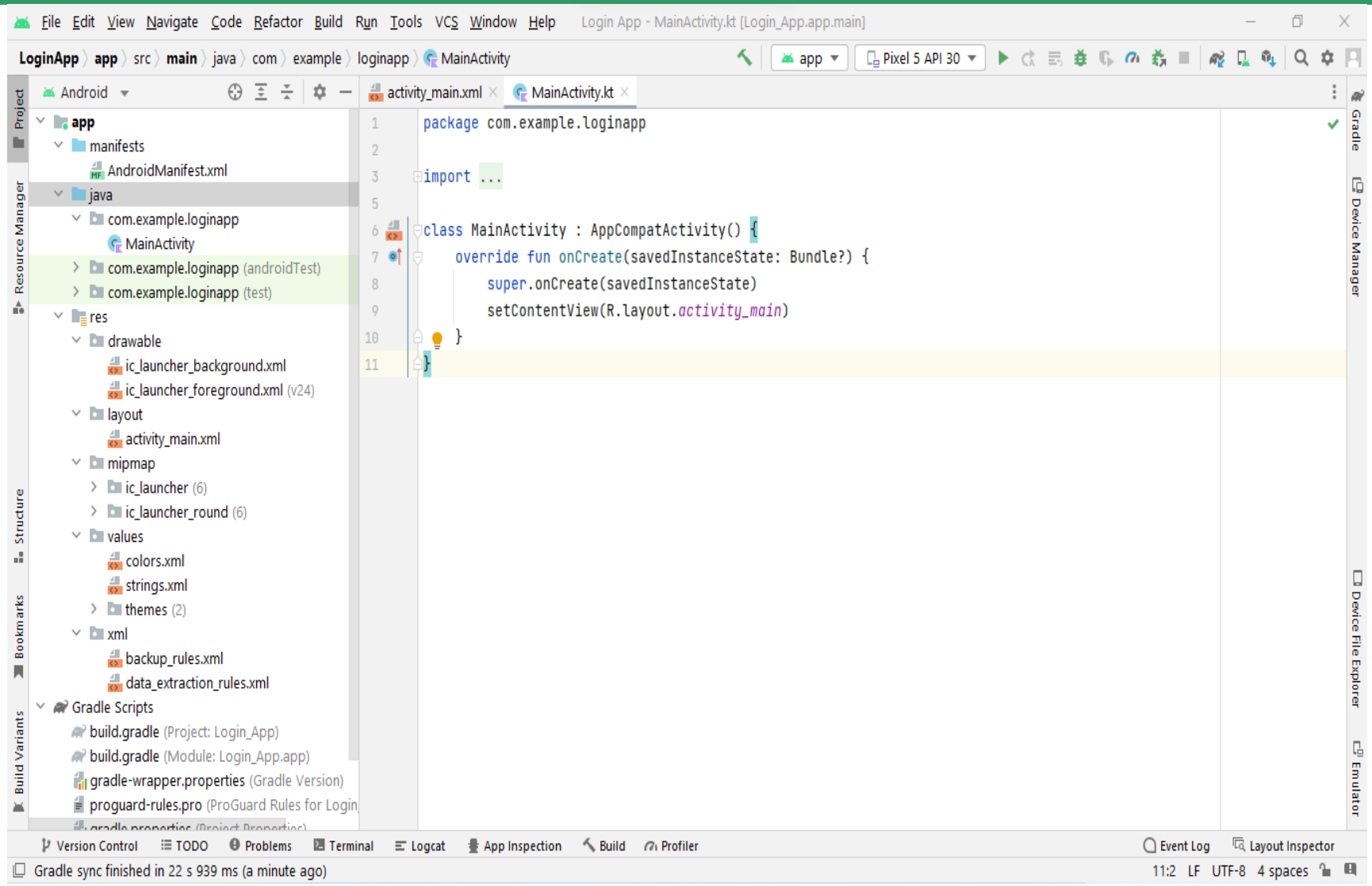
Première application Android



Première application Android



Première application Android



SDK

L'assistant a créé de nombreux éléments visibles dans la colonne de gauche de l'IDE :

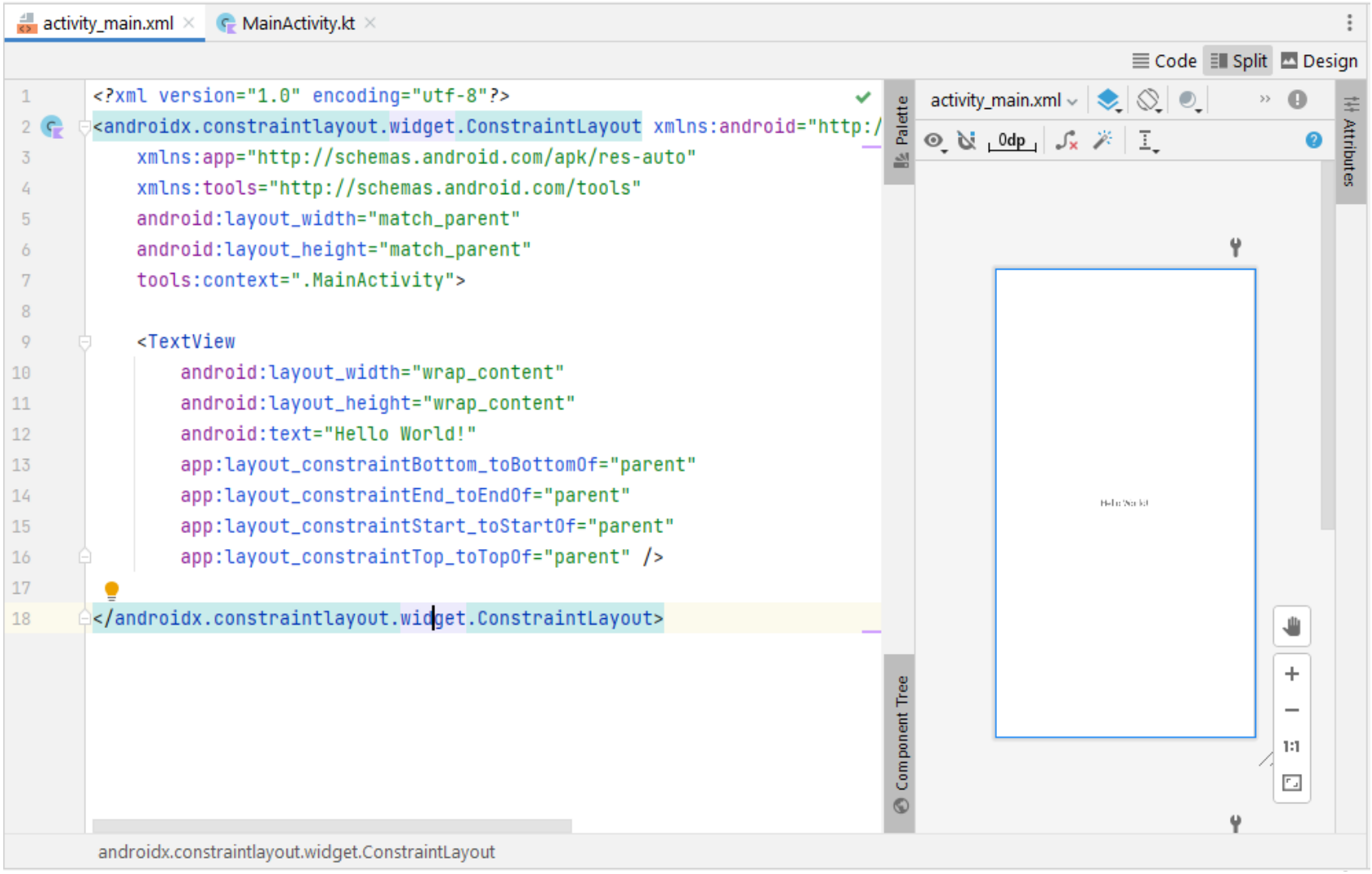
- **manifests** : description et liste des classes de l'application
- **java** : les sources, rangés par paquetage,
- **res** : ressources = fichiers XML et images de l'interface, il y a des sous-dossiers :
  - ✓ layout : interfaces (disposition des vues sur les
  - ✓ écrans)menu : menus contextuels ou d'application
  - ✓ mipmap et drawable : images, icônes de l'interface
  - ✓ values : valeurs de configuration, textes. . .
- **Gradle scripts** : c'est l'outil de compilation du projet.

NB: ne pas chercher à tout comprendre cette semaine

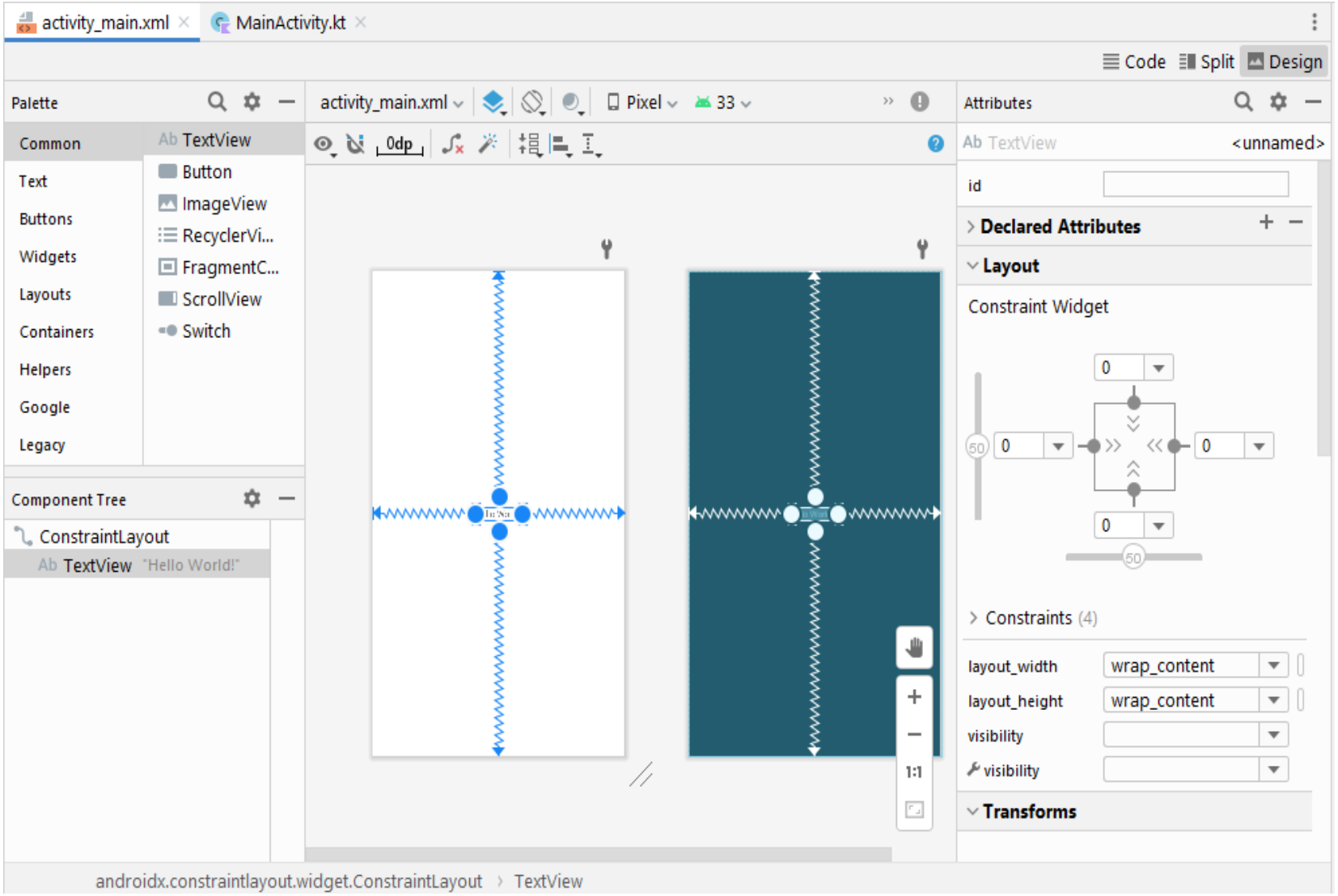
Structure d'un projet Android

```
+-- app/
|   +-- build/                          FICHIERS COMPILÉS
|   +-- build.gradle                    SPÉCIF. COMPILATION
|   `-- src/
|       +-- androidTest/                TESTS UNITAIRES ANDROID
|       +-- main/
|       |   +-- AndroidManifest.xml     DESCR. DE L'APPLICATION
|       |   +-- java/                   SOURCES
|       |   `-- res/                    RESSOURCES (ICONES...)
|       `-- test/                       TESTS UNITAIRES JUNIT
+-- build/                              FICHIERS TEMPORAIRES
+-- build.gradle                        SPÉCIF. PROJET
`-- gradle/                             FICHIERS DE GRADLE
```

Editeurs d'Android studio



Editeurs d'Android studio



Gradle

- Gradle est un outil de construction de projets comme Make (projets C++ sur Unix), Ant (projets Java dans Eclipse) et Maven.
- De même que make se sert d'un fichier Makefile, Gradle se sert de fichiers nommés **build.gradle** pour construire le projet.

On distingue deux **build.gradle** :

- un script build.gradle dans le dossier racine du projet. Il indique quelles sont les dépendances générales (noms des dépôts Maven contenant les librairies utilisées).
- un dossier **app** contenant l'application du projet.
- un script build.gradle dans le dossier app pour compiler l'application

Gradle

Remarque:

Chaque modification d'une source ou d'une ressource fait reconstruire le projet (compilation des sources, transformation des XML et autres). C'est automatique.

Utilisation des bibliothèques

Certains projets font appel à des bibliothèques externes. On les spécifie dans le **build.gradle** du dossier **app**, dans la zone dependencies :

```
dependencies {
    // support
    implementation 'androidx.appcompat:appcompat:1.2.0'

    // annotations
    annotationProcessor 'org.projectlombok:lombok:1.18.16'
    implementation 'androidx.annotation:annotation:1.1.0'

    // fuites mémoire
    debugImplementation 'com.squareup.leakcanary:leakcanary-android:2.5'
}
```

Utilisation des bibliothèques

Certains projets font appel à des bibliothèques externes. On les spécifie dans le **build.gradle** du dossier **app**, dans la zone dependencies :

```
dependencies {
    // support
    implementation 'androidx.appcompat:appcompat:1.2.0'

    // annotations
    annotationProcessor 'org.projectlombok:lombok:1.18.16'
    implementation 'androidx.annotation:annotation:1.1.0'

    // fuites mémoire
    debugImplementation 'com.squareup.leakcanary:leakcanary-android:2.5'
}
```

Les bibliothèques indiquées sont automatiquement téléchargées.

Exécution de l'application

L'application est prévue pour tourner sur un appareil (smartphone ou tablette) réel ou simulé (virtuel). Le SDK Android permet de :

- Installer l'application sur une vraie tablette connectée par USB
- Simuler l'application sur une tablette virtuelle AVD

AVD = Android Virtual Device

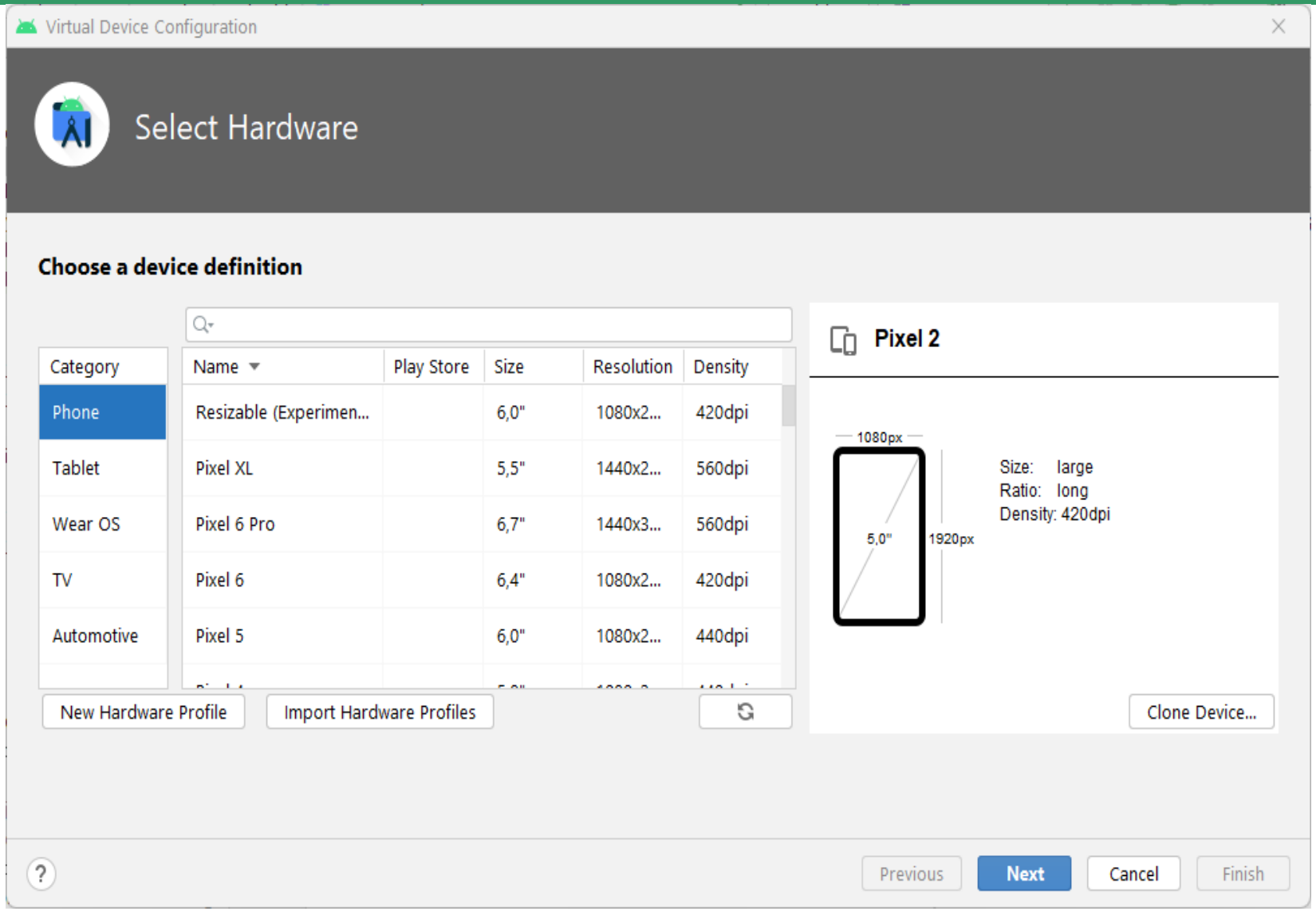
C'est une machine virtuelle comme celles de VirtualBox et VMware, mais basée sur QEMU. QEMU est en licence GPL, il permet d'émuler toutes sortes de CPU dont des ARM7, ceux qui font tourner la plupart des tablettes Android

Exécution de l'application

L'assistant de création de tablette demande :

- **Modèle** de tablette ou téléphone à simuler,
- **Version** du système Android,
- **Orientation** et **densité** de l'écran
- **Options de simulation** :
  - ✓ Snapshot : mémorise l'état de la machine d'un lancement à l'autre, mais exclut Use Host GPU,
  - ✓ Use Host GPU : accélère les dessins 2D et 3D à l'aide de la carte graphique du PC.
- **Options avancées** :
  - ✓ RAM : mémoire à allouer, mais est limitée par votre PC,
  - ✓ Internal storage : capacité de la flash interne,
  - ✓ SD Card : capacité de la carte SD simulée supplémentaire (optionnelle).

Exécution de l’application



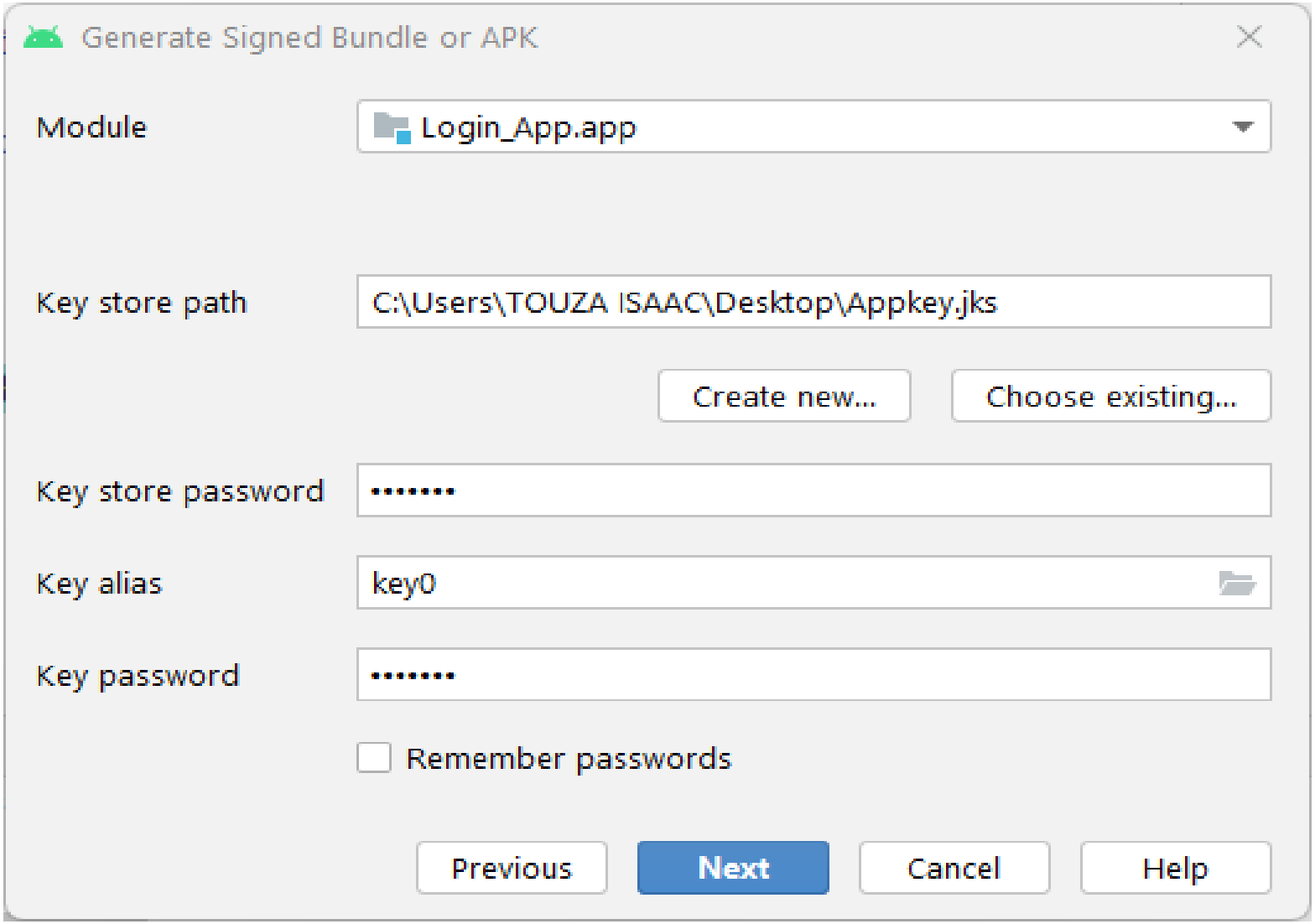
Création d'un paquet installable

Un paquet Android est un fichier .apk. C'est une archive signée (authentifiée) contenant les binaires, ressources compressées et autres fichiers de données.

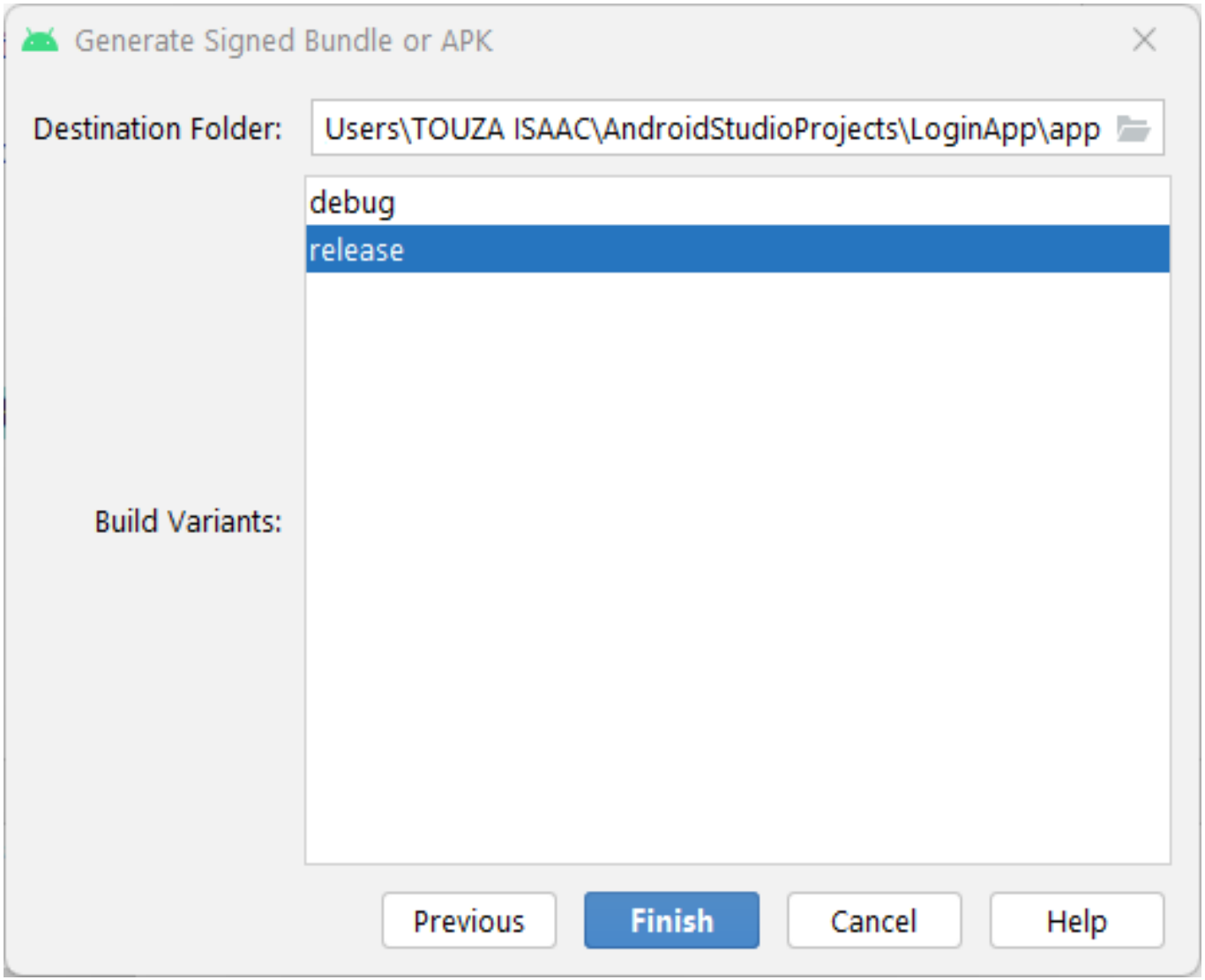
La création est relativement simple avec Studio :

1. Menu contextuel du projet Build..., choisir Generate Signed APK,
2. Signer le paquet à l'aide d'une clé privée,
3. Définir l'emplacement du fichier .apk.

Création d'un paquet installable



Création d’un paquet installable

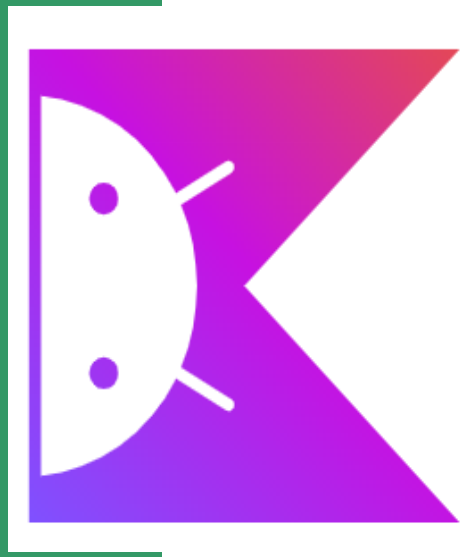


Et Voilà !

C’est fini pour cette semaine, rendez-vous la séance prochaine pour le cours sur Kotlin.

SÉANCE 2

KOTLIN



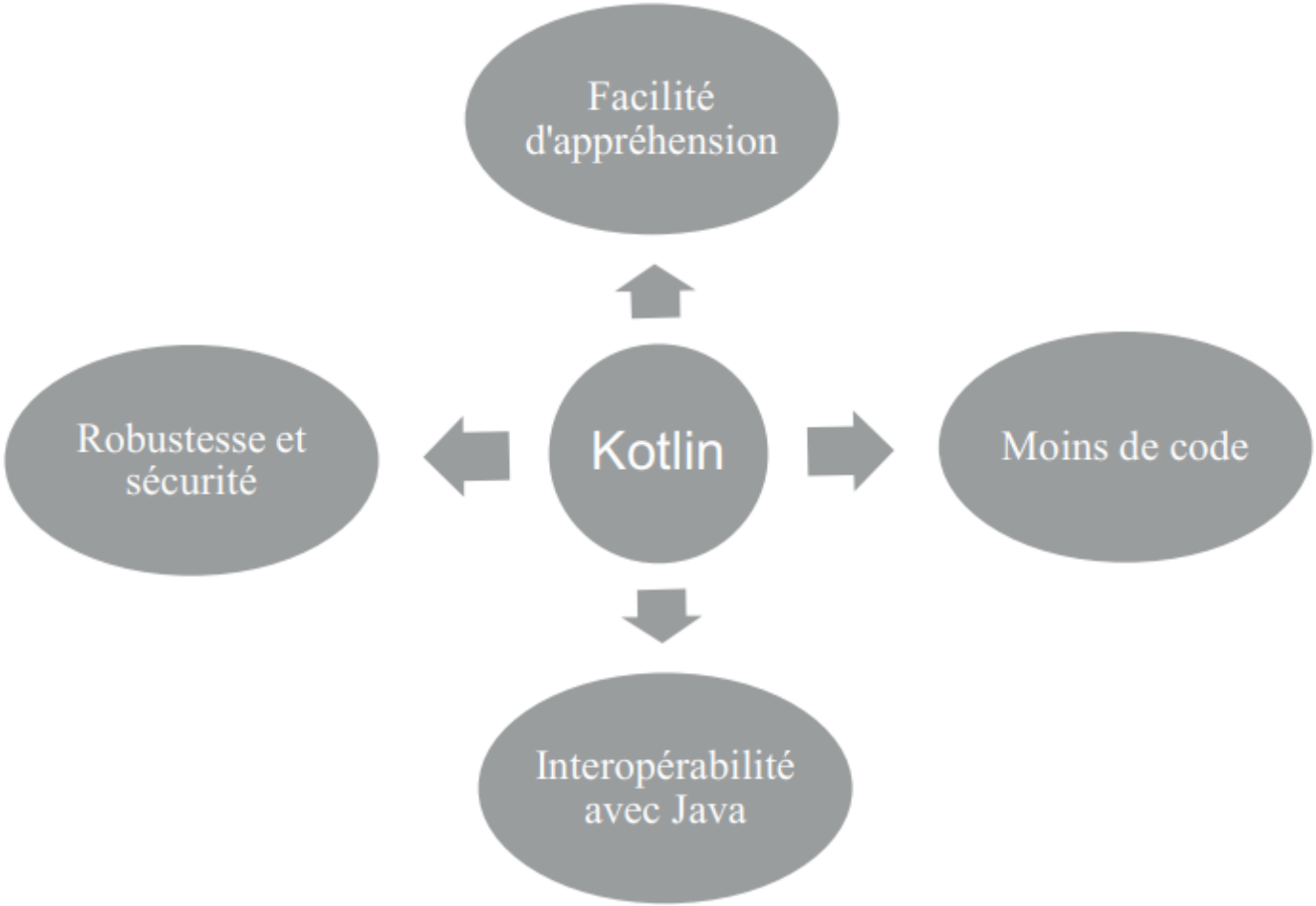
Cette semaine nous allons découvrir le langage Kotlin :

- Maîtriser les bases de Kotlin
- Comprendre la Programmation Orientée Objet avec Kotlin
- Mettre en œuvre la programmation fonctionnelle

Présentation du langage Kotlin

- Kotlin est un langage de programmation **orienté objet** et **fonctionnel**, avec un typage statique qui permet de compiler pour la machine virtuelle Java, JavaScript, et vers plusieurs plateformes en natif grâce à LLVM (Low Level Virtual Machine en français : « machine virtuelle de bas niveau).
- Conçu Par : JetBrains
- Date de première version : 22 juillet 2011
- Dernière version : 1.9.20 (29 septembre 2022)
- Licence : Licence Apache version 2.0
- Paradigme : Objet, fonctionnel
- Site web : kotlinlang.org

Avantages de Kotlin



Pourquoi une application mobile avec Kotlin ?

Expressif et concis

Les fonctionnalités modernes du langage de Kotlin vous aident à vous concentrer sur l'expression de vos idées et à écrire moins de code récurrent.

Un code plus sûr

En intégrant la possibilité de valeur nulle dans son système de typage, Kotlin vous aide à éviter les NullPointerExceptions. La probabilité qu'une application Android écrite en Kotlin plante est réduite de 20 %.

Pourquoi une application mobile avec Kotlin ?

Jetpack Compose

Le kit d'interface utilisateur moderne d'Android est basé sur Kotlin, ce qui vous permet de créer rapidement des UI grâce à des API puissantes et intuitives.

Simultanéité structurée

Les coroutines Kotlin simplifient la programmation asynchrone et permettent d'effectuer de manière facile et efficace des tâches courantes telles que les appels réseau et les mises à jour de bases de données.

Exécuter un script kotlin

- Téléchargez la dernière version (kotlin-compiler-1.8.0.zip) depuis GitHub Releases (https://github.com/JetBrains/kotlin/releases/tag/v1.8.0) .
- Décompressez le compilateur autonome dans un répertoire et ajoutez éventuellement le répertoire bin au chemin système. Le répertoire bin contient les scripts nécessaires pour compiler et exécuter Kotlin sous Windows, macOS et Linux.
- Ecrire votre script kotlin et l'enregister sous l'extension .kt
- Compilez l'application à l'aide du compilateur Kotlin :

```
kotlinc hello.kt -include-runtime -d hello.jar
```

Exécuter un script kotlin

- L'option -d indique le chemin de sortie des fichiers de classe générés, qui peuvent être soit un répertoire, soit un fichier .jar. L'option -include-runtime rend le fichier .jar résultant autonome et exécutable en y incluant la bibliothèque d'exécution Kotlin.
- Exécutez l'application.

```
java -jar hello.jar
```

Kotlin: Déclaration de variable

Kotlin utilise deux mots clés différents pour déclarer des variables : **val** et **var**.

- Utilisez val pour une variable dont la valeur ne change jamais (constante). Vous ne pouvez pas réattribuer une valeur à une variable déclarée à l'aide de val.
- Utilisez var pour une variable dont la valeur peut changer.

```
var count: Int = 10
count = 15
```

```
val languageName: String = "Kotlin"
```

Kotlin: Inférence de type

Dans l'exemple suivant, **languageName** est déduit en tant que String. Vous ne pouvez donc pas appeler des fonctions qui ne font pas partie de la classe String :

```
val languageName = "Kotlin"
val upperCaseName = languageName.toUpperCase()

// Fails to compile
languageName.inc()
```

Kotlin: Sécurité nulle

Dans certaines langues, une variable de type de référence peut être déclarée sans valeur explicite initiale. Dans ces cas, les variables contiennent généralement une valeur nulle. Par défaut, les variables Kotlin ne peuvent pas contenir de valeurs nulles. Cela signifie que l'extrait suivant n'est pas valide :

```
// Fails to compile
val languageName: String = null
```

Pour qu'une variable puisse contenir une valeur **nulle**, elle doit être de type **nullable**. Vous pouvez spécifier qu'une variable **nullable** en ajoutant le suffixe **?** à son type, comme illustré dans l'exemple suivant :

```
val languageName: String? = null
```

Expressions conditionnelles

Kotlin propose plusieurs méthodes permettant d'implémenter une logique conditionnelle. La plus courante est l'instruction if-else.

```
if (count == 42) {
    println("I have the answer.")
} else {
    println("The answer eludes me.")
}
```

Expressions conditionnelles

Pour éviter cette répétition, Kotlin permet d'utiliser des expressions conditionnelles.

L'exemple peut être réécrit comme suit :

```
val answerString: String = if (count == 42) {
    "I have the answer."
} else if (count > 35) {
    "The answer is close."
} else {
    "The answer eludes me."
}

println(answerString)
```

Expressions conditionnelles

Lorsque la complexité de votre instruction conditionnelle augmente, vous pouvez envisager de remplacer votre expression if-else par une expression **when**, comme illustré dans l'exemple suivant :

```
val answerString = when {
    count == 42 -> "I have the answer."
    count > 35 -> "The answer is close."
    else -> "The answer eludes me."
}

println(answerString)
```

Expressions conditionnelles

Exercice :

Ecrire un programme Kotlin qui affiche la mention de l'étudiant en fonction de sa moyenne.

- 10 <= Moyenne < 12 : Passable
- Moyenne > 14 : Très Bien
- Moyenne = 20 : Parfait

NB: utiliser l'expression when.

Les boucles en kotlin

Répéter une action x fois

```
 repeat(x) { i ->
  action_à_repeter
}
```

Itérations sur les listes

```
val list = listOf ("element1", "element2", "element3")
for(str in list) {
  action_à_realiser
}
```

Les boucles en kotlin

Itérations sur les entiers

```
for(i in 0..9) {
    action_à_realiser
 }
```

usage d'un index pour itérer:

```
for((index, element) in iterable.withIndex()) {
 print("$element at index $index")
}
```

Les boucles en kotlin

Approche fonctionnelle de l'itération

```
iterable.forEach {
   print(it.toString())
}
```

Les boucles while et do-while

```
while(condition) {
    doSomething()
 }
```

```
do {
      doSomething()
  } while (condition)
```

Les tableaux en kotlin

- Un tableau c'est un conteneur de variable. Les variables qu'il contient sont du même type et sont stockées dans des cases.
- Les cases représentent des adresses en mémoire
- En Kotlin, on définit donc le tableau de la manière suivante :

Var NomTableau : type

Exemple: Var Tab : IntArray

Les tableaux en kotlin

- Un tableau peut également contenir un objet :

```
Class MonObjet
var MonTableau : Array <MonObjet>
```

En Kotlin, les types Int, Float, Long, Double, Char, Byte, Short, Boolean sont également représenté en tant qu'objet. Par conséquent on peut également écrire nos tableaux comme ci-dessous :

```
var TableauBoolean : Array<Boolean>
var TableauEntier : Array<Int>
```

Les tableaux en kotlin

- Un tableau s'est aussi une variable, il peut également contenir un tableau.

```
var Tableau_de_tableau_entier : Array<Array<Int>>
```

Les tableaux en kotlin

- L'initialisation est très simple, on utilise la méthode qui se nomme le type de notre tableau suivi de "ArrayOf" pour les variables qui ne sont pas des objets. Pour un tableau d'objet on utilise seulement la méthode "arrayOf".

```
var TableauEntier: IntArray=intArrayof(2,10,11,33)
```

Les tableaux en kotlin

On peut parcourir un tableau de deux façons:

- Utilisation de la boucle “for”.

```
var TableauEntier: IntArray=intArrayof(2,10,11,33)
for (I in TableauEntier) {
    TableauEntier[i].toString()
}
```

- la methode forEach()

```
 var TableauEntier: IntArray=intArrayof(2,10,11,33)
TableauEntier.forEach{ it: Int
  it.toString()
}
```

Les fonctions en kotlin

- Pour déclarer une fonction, utilisez le mot clé **fun** suivi du nom de la fonction.
- Ensuite, définissez les types d'entrées que votre fonction reçoit, le cas échéant, et déclarez le type de sortie qu'elle renvoie.
- Le corps d'une fonction est l'endroit où vous définissez les expressions appelées lorsque votre fonction est appelée.

```
fun NomFonction (arguments et types ): type_retour {
  Votre code ici
  return valeur
}
```

Les fonctions en kotlin

- Pour déclarer une fonction, utilisez le mot clé **fun** suivi du nom de la fonction.
- Ensuite, définissez les types d'entrées que votre fonction reçoit, le cas échéant, et déclarez le type de sortie qu'elle renvoie.
- Le corps d'une fonction est l'endroit où vous définissez les expressions appelées lorsque votre fonction est appelée.

```
fun NomFonction (arguments et types ): type_retour {
  Votre code ici
  return valeur
}
```

Les fonctions en kotlin

Exercice :

1. Ecrire une fonction Kotlin qui calcule la moyenne des éléments d'un tableau de taille N=50

2. Ecrire une fonction Kotlin qui prend entrée deux chaines de caractères et compare leur longueur.

PROGRAMMATION ORIENTEE OBJET EN KOTLIN

Les classes en Kotlin

Terminologie

- Les **classes** sont des modèles d'objets.
- Les **objets** sont des instances de classes
- Les **propriétés** sont les caractéristiques des classes.
- Les **méthodes**, également appelées fonctions membres, sont les fonctionnalités de la classe.
- Une **interface** est une spécification qu'une classe peut implémenter.
- Les **packages** sont un moyen de regrouper du code connexe pour le garder organisé ou de créer une bibliothèque de code.
- **Encapsulation**. Encapsule les propriétés et méthodes associées qui effectuent des actions sur ces propriétés dans une classe.

Les classes en Kotlin

Terminologie

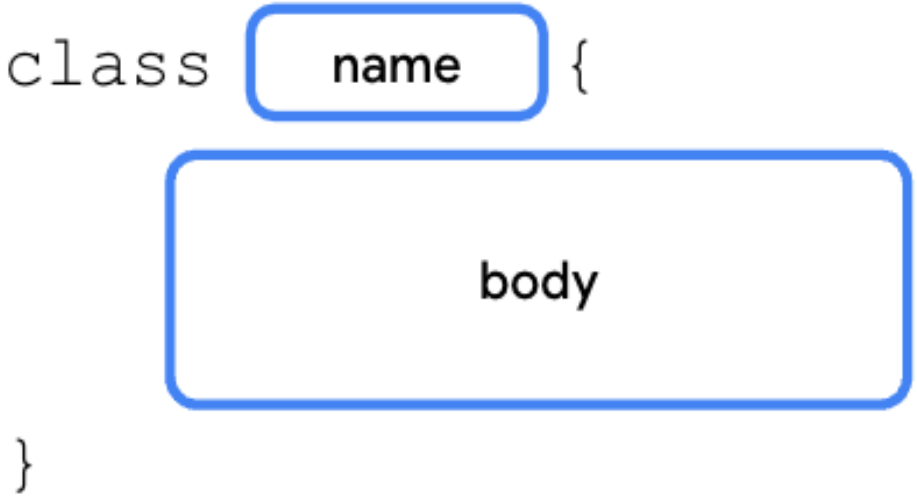
- **Abstraction**. Il s'agit d'une extension de l'encapsulation. L'idée est de masquer autant que possible la logique d'implémentation interne.
- **Héritage**. Permet de construire une classe sur la base des caractéristiques et du comportement d'autres classes en établissant une relation parent-enfant.
- **Polymorphisme**. Le mot est une adaptation de la racine grecque poly-, qui signifie "plusieurs", et de -morphisme, qui signifie forme. Le polymorphisme est la capacité à utiliser différents objets d'une façon commune

Les classes en Kotlin

Définition d'une classe en Kotlin

Tous les types mentionnés jusqu'à présent sont intégrés au langage de programmation Kotlin. Si vous souhaitez ajouter un type personnalisé, vous pouvez définir une classe.

La définition de classe commence par le mot clé **class**, suivi d'un nom et d'une série d'accolades.



Les classes en Kotlin

Définition d'une classe en Kotlin

Une classe comprend trois parties principales :

- **Propriétés**. Variables spécifiant les attributs des objets de la classe.
- **Méthodes**. Fonctions contenant les comportements et les actions de la classe.
- **Constructeurs**. Fonction membre spéciale qui crée des instances de la classe dans tout le programme dans lequel elle est définie.

Les classes en Kotlin

Propriétés d'une classe

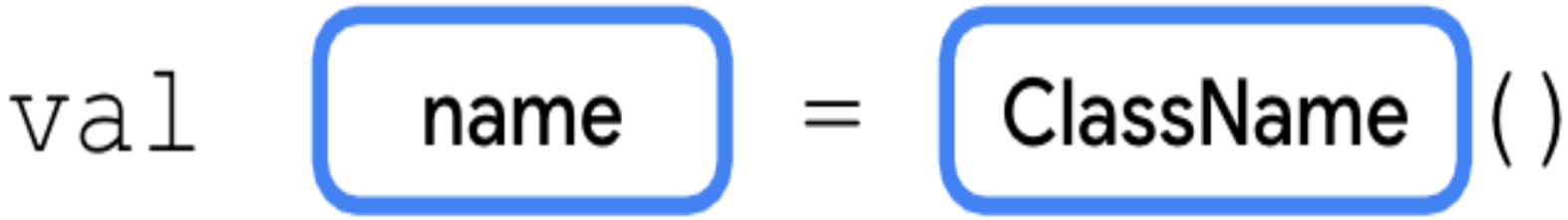
Les propriétés sont essentiellement des variables définies dans le corps de la classe et non dans celui de la fonction. Cela signifie que la syntaxe utilisée pour définir les propriétés et les variables est identique. Vous définissez une propriété non modifiable avec le mot clé **val** et une propriété modifiable avec le mot clé **var**.

```
class SmartDevice {
    val name = "Android TV"
    val category = "Entertainment"
    var deviceStatus = "online"
}
```

Les classes en Kotlin

Instance d'une classe

Pour utiliser un objet, vous devez le créer et l'affecter à une variable, de la même manière que vous définissez une variable. Le mot clé **val** ou **var** est suivi du nom de la variable, d'un opérateur d'affectation =, puis de l'instanciation de l'objet de classe. La syntaxe est illustrée dans le schéma ci-dessous :



Les classes en Kotlin

Les méthodes de classe

Les actions que la classe peut effectuer y sont définies en tant que fonctions (méthodes).

Exemple : Définissons les méthodes turnOn() et turnOff() dans la classe SmartDevice

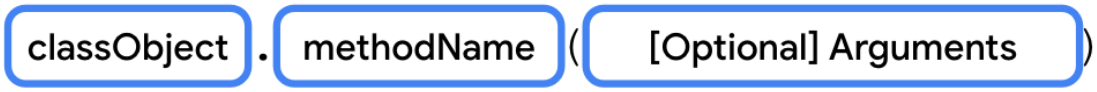
```
class SmartDevice {

    fun turnOn(){
        println("Smart device is turned on.")
    }

    fun turnOff(){
        println("Smart device is turned off.")
    }
}
```

Les classes en Kotlin

Appel d'une méthode sur un objet

- Pour appeler une méthode dans une classe, la procédure est la même que celle utilisée pour appeler d'autres fonctions à partir de la fonction **main**()
- Pour appeler une méthode de classe en dehors de la classe, commencez par l'objet de classe suivi de l'opérateur **.** , du nom de la fonction et d'une paire de parenthèses. Le cas échéant, les parenthèses contiennent les arguments requis par la méthode.

La syntaxe est illustrée dans le schéma ci-dessous :



Les classes en Kotlin

Appel d'une méthode sur un objet

Remarque :

Le principe d'une classe est d'encapsuler un ensemble de données qui ont un certain lien entre elles, nous allons avoir besoin d'assesseurs (getters) pour y avoir accès, et de mutateurs (setters) pour les modifier. En Kotlin, vous n'aurez pas besoin de les déclarer explicitement. Ils seront "générés" automatiquement grâce aux mots-clés **val** et **var** indiqués avant chaque nom de propriété dans le constructeur de la classe :

- **val** : La propriété sera immuable, vous ne pourrez donc pas la modifier. Kotlin générera alors pour vous uniquement un assesseur.
- **var** : La propriété sera muable, vous pourrez la modifier. Kotlin générera alors pour vous un assesseur et un mutateur.

Plus besoin d'utiliser les préfixes **get** et **set** devant le nom d'une propriété : appelez directement la propriété par son nom !

Les classes en Kotlin

Constructeurs d'une classe

Vous pouvez définir un constructeur avec ou sans paramètres.

- **Constructeur par défaut :** est un constructeur sans paramètres. Vous pouvez définir un constructeur de ce type comme indiqué dans cet extrait de code :

```
class SmartDevice constructor() {
    ...
}
```

La concision est l'une des caractéristiques du langage Kotlin. Vous pouvez donc supprimer le mot clé **constructor** si le constructeur ne contient ni annotations ni modificateurs de visibilité

```
class SmartDevice {
    ...
}
```

Les classes en Kotlin

Constructeurs d'une classe

- **Constructeur paramétré**. Le constructeur accepte à présent des paramètres pour configurer ses propriétés. Par conséquent, la façon d'instancier un objet pour une classe de ce type change également. Le schéma ci-dessous illustre la syntaxe complète pour instancier un objet :

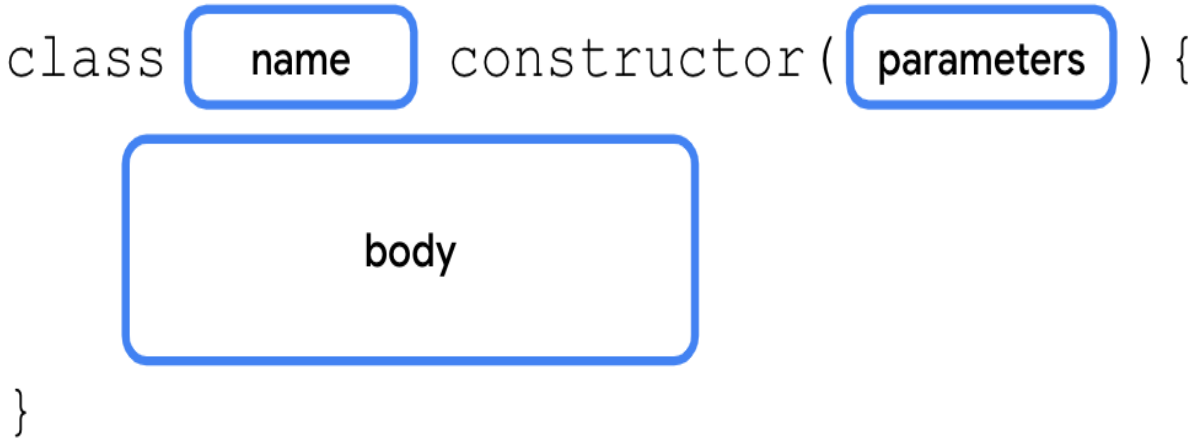

```
class SmartDevice(val name: String, val category: String) {

    var deviceStatus = "online"

    fun turnOn(){
        println("Smart device is turned on.")
    }

    fun turnOff(){
        println("Smart device is turned off.")
    }
}
```

Les classes en Kotlin

Constructeurs d'une classe

En langage Kotlin, il existe deux principaux types de constructeurs :

- **Constructeur principal**. Une classe ne peut avoir qu'un seul constructeur principal, qui est défini dans l'en-tête de la classe. Un constructeur principal peut être un constructeur par défaut ou paramétré. Le constructeur principal ne possède pas de corps, ce qui signifie qu'il ne peut pas contenir de code. Le schéma ci-dessous illustre la syntaxe complète permettant de définir un constructeur principal :



Les classes en Kotlin

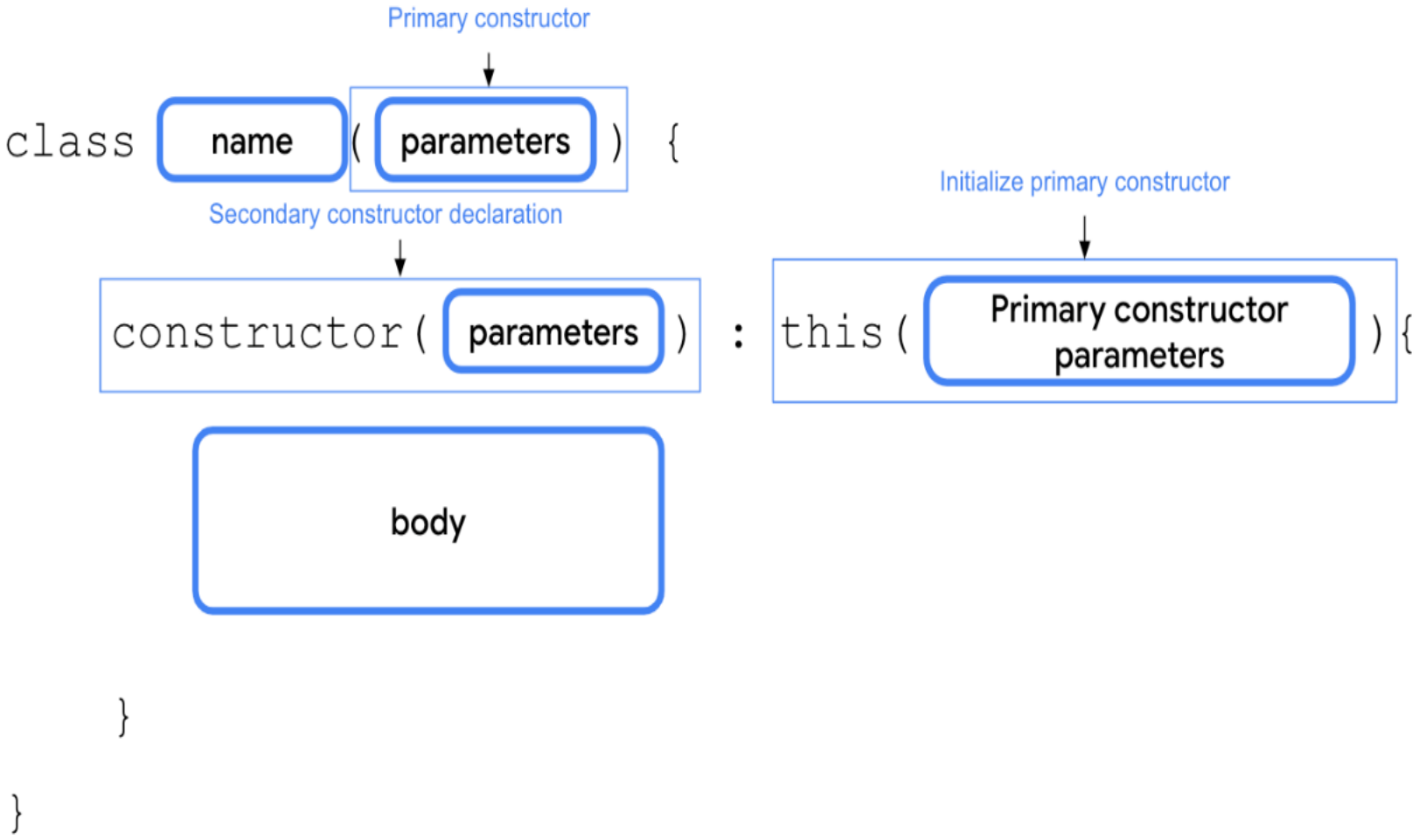
Constructeurs d'une classe

- **Constructeur secondaire**. Une classe peut comporter plusieurs constructeurs secondaires. Vous pouvez définir le constructeur secondaire avec ou sans paramètres. Le constructeur secondaire peut initialiser la classe et a un corps qui peut contenir la logique d'initialisation. Si la classe comporte un constructeur principal, chaque constructeur secondaire doit l'initialiser. Le constructeur secondaire est inclus dans le corps de la classe et sa syntaxe se compose de trois parties :

  1. **Déclaration du constructeur secondaire**
  2. **Initialisation du constructeur principal**
  3. **Corps du constructeur secondaire.**

Les classes en Kotlin

Constructeurs d'une classe



Les classes en Kotlin

Constructeurs d'une classe

```
class SmartDevice(val name: String, val category: String) {
  var deviceStatus = "online"

   constructor(name: String, category: String, statusCode: Int) :
this(name, category) {
       deviceStatus = when (statusCode) {
           0 -> "offline"
           1 -> "online"
           else -> "unknown"
       }
   }
   ...
}
```

Les classes en Kotlin

Relation entre les classes

- En langage Kotlin, toutes les classes sont dites "finales", ce qui signifie que vous ne pouvez pas les étendre. Vous devez donc définir des relations entre elles.
- Le concept **d'héritage** vous permet de créer une classe en vous appuyant sur les caractéristiques et le comportement d'une autre classe. Il s'agit d'un mécanisme efficace qui vous aide à écrire du code réutilisable et à établir des relations entre les classes.
- Pour définir la relation entre la **super-classe** et ses **sous-classes**, on commence par ajouter un mot clé open avant le mot clé class pour informer le compilateur que cette classe est extensible, de sorte que d'autres classes puissent maintenant l'étendre.

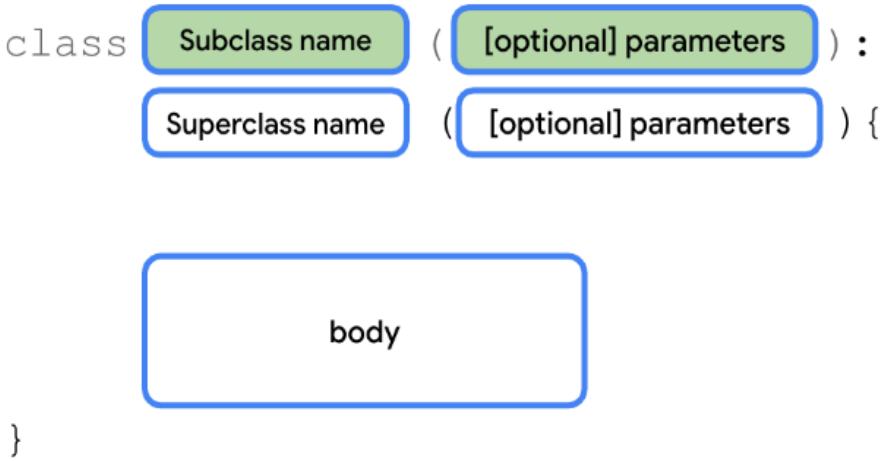
Les classes en Kotlin

Relation entre les classes

Exemple d'une classe extensible

```
open class SmartDevice(val name: String, val category: String) {
    ...
}
```

La syntaxe utilisée pour créer une sous-classe est illustrée dans le schéma ci-dessous :



Les classes en Kotlin

Relation entre les classes

Exemple d'une sous-classe

```
class SmartTvDevice(deviceName: String, deviceCategory: String) :
    SmartDevice(name = deviceName, category = deviceCategory) {
}
```

Une sous classe peut posséder ses propres méthodes pouvant portés les mêmes noms que celles de la superclasse. Pour ce faire :

- Dans le corps de la **super-classe**, ajoutez un mot clé **open** avant le mot clé fun de chaque méthode :

```
open class SmartDevice {
    ...
    var deviceStatus = "online"

    open fun turnOn() {
        // function body
    }
```

Les classes en Kotlin

Relation entre les classes

- Dans la sous-classe, avant le mot clé fun des méthodes, ajoutez le mot clé **override**, pour indiquer à l'environnement d'exécution Kotlin d'exécuter le code inclus dans la méthode définie dans la sous-classe.

```
class SmartLightDevice(name: String, category: String) :
    SmartDevice(name = name, category = category) {

    var brightnessLevel = 0

    override fun turnOn() {
        deviceStatus = "on"
        brightnessLevel = 2
        println("$name turned on. The brightness level is $brightnessLevel.")
    }

    override fun turnOff() {
        deviceStatus = "off"
        brightnessLevel = 0
        println("Smart Light turned off")
    }

    fun increaseBrightness() {
        brightnessLevel++
    }
}
```

Les classes en Kotlin

Relation entre les classes

Appeler une méthode à partir de la **super-classe** revient à l'appeler depuis l'extérieur de la classe. Au lieu d'utiliser un opérateur . entre l'objet et la méthode, vous devez utiliser le mot clé super, qui indique au compilateur Kotlin d'appeler la méthode sur la **super-classe** plutôt que sur la sous-classe.

La syntaxe utilisée pour appeler la méthode à partir de la **super-classe** commence par un mot clé super, suivi de l'opérateur **.** , du nom de la fonction et d'une paire de parenthèses. Le cas échéant, les arguments sont placés entre parenthèses. La syntaxe est illustrée dans le schéma ci-dessous :

Les classes en Kotlin

Modificateurs de visibilité

Les modificateurs de visibilité jouent un rôle important pour l'obtention de l'encapsulation :

- ✓ Dans une classe, ils vous permettent de masquer vos propriétés et méthodes afin d'empêcher tout accès non autorisé depuis l'extérieur.
- ✓ Dans un package, ils vous permettent de masquer les classes et interfaces afin d'empêcher tout accès non autorisé depuis l'extérieur.

Kotlin propose quatre modificateurs de visibilité :

Modificateur	Accessible dans la même classe	Accessible dans la sous-classe	Accessible dans le même module	Accessible en dehors du module
private	✓	X	X	X
protected	✓	✓	X	X
internal	✓	✓	✓	X
public	✓	✓	✓	✓

Les classes en Kotlin

Modificateurs de visibilité

Remarques :

- Un **module** est un ensemble de fichiers sources et de paramètres de compilation permettant de diviser le projet en unités fonctionnelles distinctes. Votre projet peut comporter un ou plusieurs modules. Vous pouvez créer, tester et déboguer chaque module séparément.
- Un **package** est essentiellement un répertoire ou un dossier qui regroupe des classes associées, tandis qu'un module fournit un conteneur pour le code source, les fichiers de ressources et les paramètres de votre application. Un module peut contenir plusieurs packages.

Les classes en Kotlin

Modificateurs de visibilité

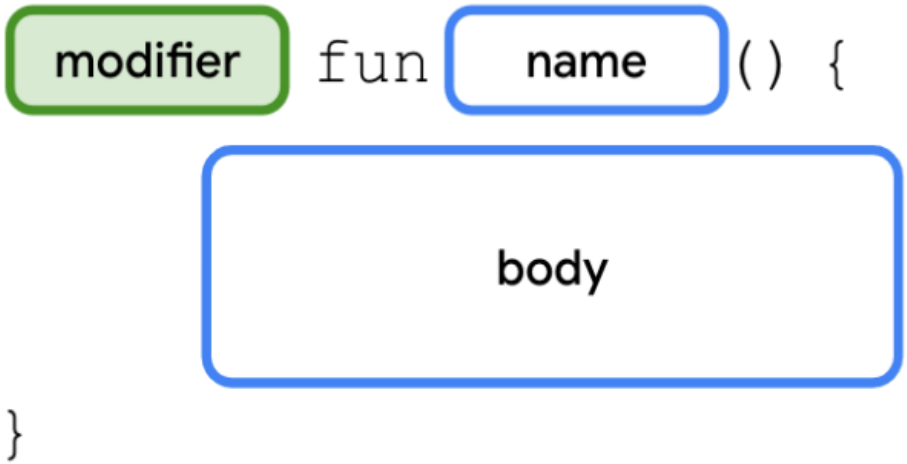
Lorsque vous définissez une classe, elle est visible publiquement et tout package qui l'importe peut y accéder. Cela signifie qu'elle est publique par défaut, sauf si vous spécifiez un modificateur de visibilité. De même, lorsque vous définissez ou déclarez des propriétés et des méthodes dans la classe, elles sont accessibles par défaut en dehors de la classe via l'objet de classe. Il est essentiel de définir une visibilité appropriée pour le code, principalement pour masquer les propriétés et les méthodes auxquelles les autres classes ne doivent pas accéder.

Les classes en Kotlin

Modificateurs de visibilité

Modificateurs de visibilité pour les propriétés

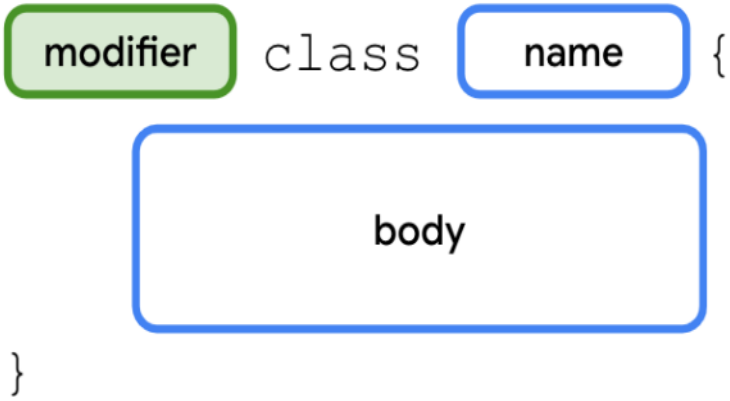
Modificateurs de visibilité pour méthodes



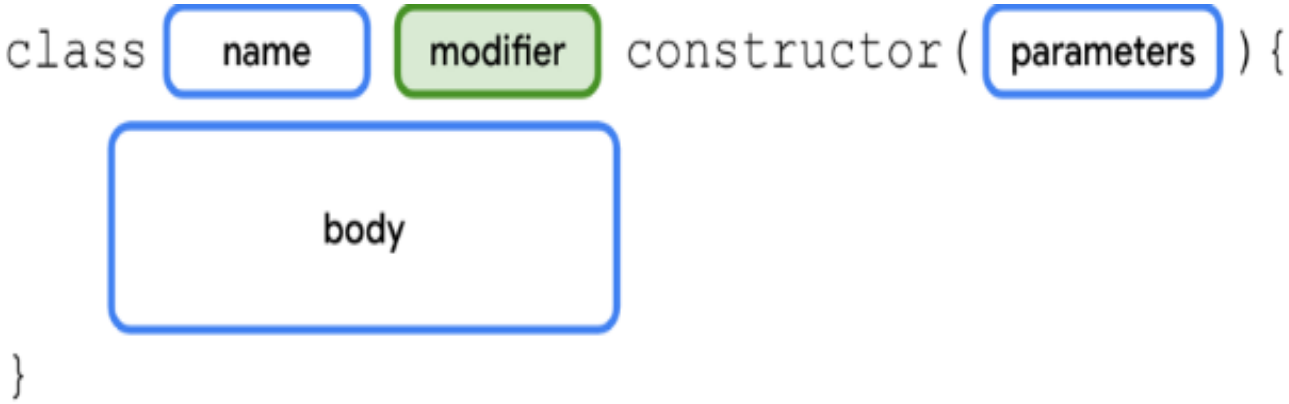
Les classes en Kotlin

Modificateurs de visibilité

Modificateurs de visibilité pour les classes



Modificateurs de visibilité pour les constructeurs



Les classes en Kotlin

Exercice :

- Ecrire un programme qui définit une classe **Forme** avec un constructeur qui donne la valeur de la hauteur(y) et de la largeur (x).
- Définir deux sous classes **Rectangle** et **Triangle**, puis donner dans ces deux sous classes la méthode **area()** qui calcule l'aire.
- Dans le programme principale **main()**, définissez un triangle et un rectangle , puis appelez la fonction **area**() dans ces deux variables.

Et Voilà !

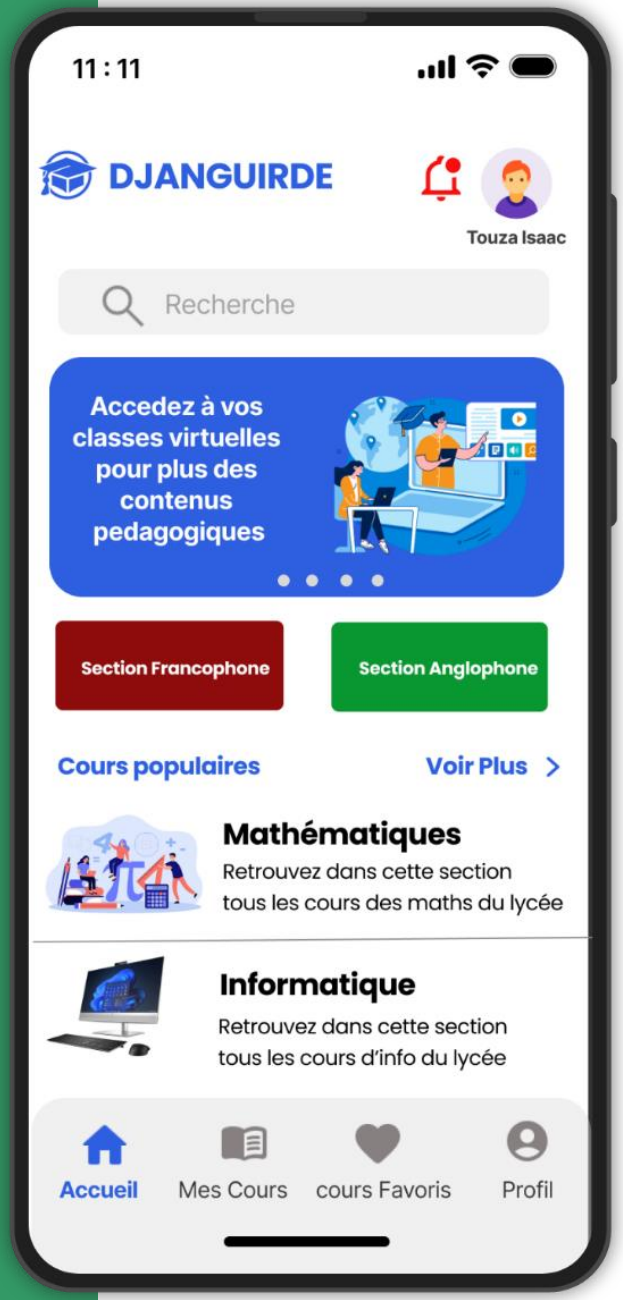
C’est fini pour cette semaine, rendez-vous la séance prochaine pour le cours sur les interfaces de l’utilisateur.

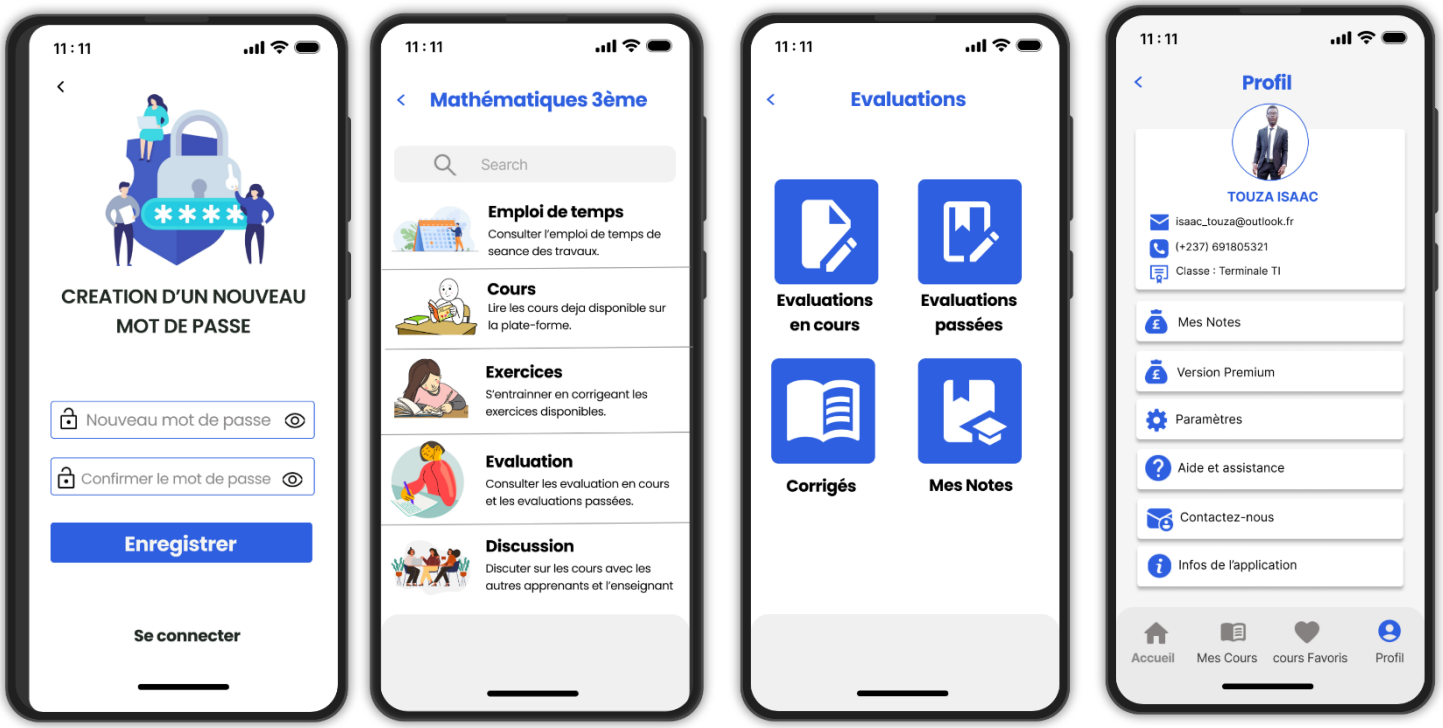
PROGRAMMATION IMPÉRATIVE

KOTLIN + XML

SÉANCE 3

CONCEPTION D'INTERFACES D'UTILISATEUR





Le cours de cette semaine explique la création d'interfaces utilisateur :

- Activités
- Relations entre un source kotlin et des ressources
- Layouts et vues

On ne s'intéresse qu'à la mise en page. L'activité des interfaces sera étudiée la semaine prochaine.

Présentation des concepts
Composition d'une application

L'interface utilisateur d'une application Android est composée d'écrans.

Un « **écran** » correspond à une activité.

Exemples : afficher des informations, éditer des informations

Les dialogues et les pop-up ne sont pas des activités, ils se superposent temporairement à l'écran d'une activité.

L'interface d'une activité est composée de vues :

- **Vues élémentaires** : boutons, zones de texte, cases à cocher. . .
- **Vues de groupement** qui permettent l'alignement des autres vues : lignes, tableaux, onglets, panneaux à défilement. . .

Chaque vue d'une interface est gérée par un objet Kotlin ou Java, comme en Java classique, avec AWT, Swing ou JavaFX.

Présentation des concepts
Création d'une interface

Les objets d'interface pourraient être créés manuellement, mais :

- c'est très complexe, car il y a une multitude de propriétés à définir,
- ça ne permet pas de localiser, c'est à dire adapter une application à chaque pays (sens de lecture de droite à gauche)

Alors, on préfère définir l'interface par l'intermédiaire d'un **fichier XML** qui décrit les vues à créer. Il est lu automatiquement par le système Android lors du lancement de l'activité et transformé en autant d'objets Java ou kotlin qu'il faut.

Chaque objet kotlin est retrouvé grâce à un « identifiant de ressource ».

Présentation des concepts
Création d'un écran

Chaque écran est géré par une instance d'une sous-classe de Activity que vous programmez. Il faut au moins surcharger la méthode **onCreate** selon ce qui doit être affiché sur l'écran :

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

C'est l'appel **setContentView(...)** qui met en place l'interface. Son paramètre est un identifiant de ressource, c'est-à-dire d'une disposition de vues d'interface. C'est ce qu'on va étudier maintenant

Les ressources
Définition

- Les **ressources** sont tout ce qui n'est pas programme dans une application.
- Dans Android, ce sont les textes, messages, icones, images, sons, interfaces, styles, etc.

Comme il va y avoir de très nombreux identifiants dans une application :

- ✓ chaque vue possède un identifiant (si on veut)
- ✓ chaque image, icone possède un identifiant
- ✓ chaque texte, message possède un identifiant
- ✓ chaque style, theme, etc. etc.

Ils ont tous été regroupés dans une classe spéciale appelée **R**

Cette classe R est générée automatiquement par ce que vous mettez dans le dossier res : interfaces, menus, images, chaînes. . . Certaines de ces ressources sont des fichiers XML, d'autres sont des images PNG.

Les ressources

Rappel sur la structure d'un fichier XML

```
<?xml version="1.0" encoding="utf-8"?>
<racine xmlns:exemple="http://...">
        <!-- commentaire -->
        <element attribut1="valeur1" attribut2="valeur2">
            <feuille1 exemple:attribut3="valeur3"/>
            <feuille2>texte</feuille2>
        </element>
        texte en vrac
</racine>
```

Les ressources
Ressources de type chaînes

- Dans **res/values/strings.xml**, on place les chaînes de l'application, au lieu de les mettre en constantes dans le code source :
- **Intérêt** : pouvoir traduire une application sans la recompiler

```
<resources>
    <string name="app_name">My First Application</string>
    <string name="message">Hello World!</string>
<string name="main_menu">Principal Menu </string>
</resources>
```

- Il est possible de traduire des ressources du type texte en plusieurs autres langues. Lorsque les textes sont définis dans **res/values/strings.xml**, il suffit de faire des copies du dossier values, en **values-us**, **values-fr**, **values-de**, etc en fonction de la langue et de traduire les textes en gardant les attributs **name**.

Les ressources
Ressources de type chaînes

- Dans un programme Kotlin, on peut très facilement placer un texte dans une vue de l'interface :

```
val tv: TextView = findViewById<TextView>(R.id.tvId) // ...
nous y reviendrons
tv.text(R.string.hello)
```

- **R.string.hello** désigne le texte de **<string name="Hello world ! ">...** dans le fichier **res/values*/strings.xml**
- Par contre, si on veut récupérer l'une des chaînes des ressources pour l'utiliser dans le programme, c'est un peu plus compliqué :

```
val texte:String =resources.getString(R.string.back)
```

Les ressources
Ressource de type texte

- Maintenant, dans un fichier de ressources décrivant une interface, on peut également employer des ressources texte :

```
<TextView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/cours" />

<Button
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Cours Dev App Mob" />
```

Les ressources
Ressource de type image

- De la même façon, les images PNG placées dans **res/drawable** et **res/mipmaps-*** sont référençables :

```
<ImageView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/logo_skool"
    android:contentDescription="@string/app_name"
/>
```

- La notation **@drawable/nom** référence l'image portant ce nom dans l'un des dossiers.
- **NB**: les dossiers **res/mipmaps-*** contiennent la même image à des définitions différentes, pour correspondre à différents téléphones et tablettes. **Ex**: mipmap-hdpi contient des icônes en 72x72 pixels.

Les ressources

Ressource de type tableau de chaine

- Identifiable par R.array.nom. On les places dans le fichier res/values/arrays.xml. Voici un extrait du fichier.

```
<resources>
    <string-array name="planetes">
        <item>Mercure</item>
        <item>Venus</item>
        <item>Terre</item>
        <item>Mars</item>
    </string-array>
</resources>
```

- Dans le programme Kotlin, il est possible de faire :

```
val  planets : Array <String> = resources.getStringArray(R.array.planetes)
```

Les ressources
Ressource de type tableau de chaine

D'autres notations existent :

- **@style/nom** pour des définitions de **res/style**
- **@menu/nom** pour des définitions de **res/menu**

Certaines notations, @package:type/nom font référence à des données prédéfinies, comme :

- @android:style/TextAppearance.Large
- @android:color/black

Il y a aussi une notation en ?type/nom pour référencer la valeur de l'attribut nom, ex : ?android:attr/textColorSecondary.

Mise en page (layouts)
Structure d'une interface Android

Un écran Android de type formulaire est généralement composé de plusieurs **vues**. Entre autres :

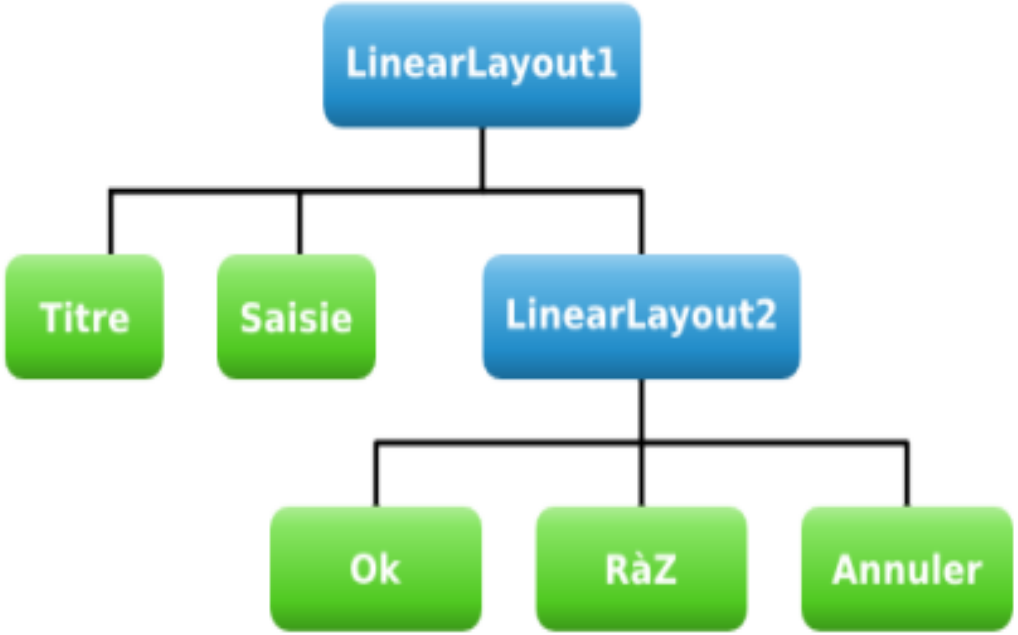
- TextView, ImageView : titre, image
- EditText : texte à saisir
- Button, CheckBox : bouton à cliquer, case à cocher

Ces vues sont alignées à l'aide de groupes sous-classes de **ViewGroup**, éventuellement imbriqués :

- **LinearLayout** : positionne ses vues en ligne ou en colonne
- **RelativeLayout**, **ConstraintLayout** : positionnent leurs vues l'une par rapport à l'autre
- **TableLayout** : positionne ses vues sous forme d'un tableau

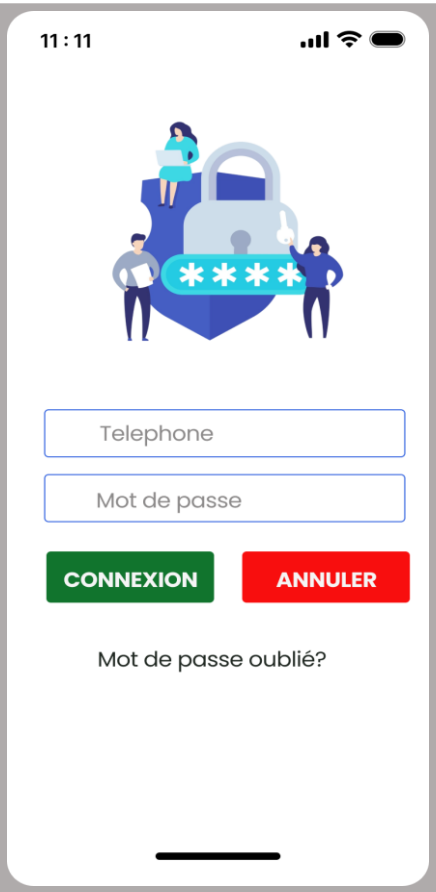
Mise en page (layouts)
Arbre de vue

Les groupes et vues forment un arbre.



Mise en page (layouts)
Arbre de vue

Exercice : Dessiner l'arbre de vue des interfaces ci-dessous.



Mise en page (layouts)

Ressource de type Layout

L'interface est stocké dans un fichier **res/layout/nom.xml** qui est référencé par son identifiant **R.layout.nom_du_fichier** (donc ici c'est R.layout.activity_main) dans le programme kotlin :

```
override fun onCreate(savedInstanceState: Bundle?) {
    WindowCompat.setDecorFitsSystemWindows(window, false)
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
}
```

La méthode **setContentView** fait afficher le layout indiqué.

Mise en page (layouts)

Identifiant et vues

Lorsque l'application veut manipuler l'une de ses vues, elle doit utiliser **R.id.symbole.**

La notation **@+id/nom** définit un identifiant pour une vue.

Exemple :

Avec la définition suivante dans res/layout/main.xml :

```
<LinearLayout ...>
    <TextView
        android:id="@+id/message"
        android:text="@string/bonjour" />
</LinearLayout>


var tv = findViewById<TextView>(R.id.message)
```

Mise en page (layouts)

Identifiant et vues

Remarques :

Dans les fichiers layout.xml, il y a deux notations à ne pas confondre :

- **@+id/nom** pour définir (créer) un identifiant
- **@id/nom** pour référencer un identifiant déjà défini ailleurs

Exemple, le Button **btn** se place sous le TextView **titre** :

```
<RelativeLayout xmlns:android="..." ... >
   <TextView ...
   android:id="@+id/titre"
   android:text="@string/titre" />
    <Button ...
    android:id="@+id/btn"
    android:layout_below="@id/titre"
    android:text="@string/ok" />
</RelativeLayout>
```

Mise en page (layouts)

Paramètres de positionnement

La plupart des groupes utilisent des paramètres de taille et de placement sous forme d'attributs XML. Par exemple, telle vue à droite de telle autre, telle vue la plus grande possible, telle autre la plus petite. Ces paramètres sont de deux sortes :

- **Ceux qui sont obligatoires pour toutes les vues** :
  - android:layout_width : largeur de la vue
  - android:layout_height : hauteur de la vue

Ils peuvent valoir :

- "wrap_content" : la vue prend la place minimale
- "match_parent" : la vue occupe tout l'espace restant
- "valeurdp" : une taille fixe, ex : "100dp" mais c'est peu recommandé, sauf 0dp pour un cas particulier.

Les dp sont une unité de taille indépendante de l'écran. 100dp font 100 pixels sur un écran de 160 dpi tandis qu'ils font 200 pixels sur un écran 320 dpi

Mise en page (layouts)

Paramètres de positionnement

- **Ceux qui sont demandés par le groupe englobant et qui en sont spécifiques** comme :
  - android:layout_weight
  - android:layout_alignParentBottom
  - android:layout_centerInParent.

Mise en page (layouts)

Autres paramètres géométriques

Il est possible de modifier l'espacement des vues :

- **Padding** espace entre le texte et les bords, géré par chaque vue
- **Margin** espace autour des bords, géré par les groupes

On peut définir les marges et les remplissages séparément sur chaque bord (Top, Bottom, Left, Right), ou identiquement sur tous :

```
<Button
    android:layout_margin="10dp"
    android:layout_marginTop="15dp"
    android:padding="10dp"
    android:paddingLeft="20dp"
/>
```

Mise en page (layouts)

Groupes des vues LinearLayout

Il range ses vues soit horizontalement, soit verticalement. Il faut seulement définir l'attribut android:orientation à "horizontal" ou "vertical".

```
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">
  <Button
      android:text="Ok"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"/>

    <Button
       android:text="Annuler"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"/>
</LinearLayout>
```

Mise en page (layouts)

Groupes des vues LinearLayout

Orientation = vertical	Orientation = horizontal
	

Mise en page (layouts)

Groupes des vues TableLayout

C'est une variante du LinearLayout : les vues sont rangées en lignes de colonnes bien alignées. Il faut construire une structure XML comme celle-ci :

```
<TableLayout ...>
        <TableRow>
                <vue 1.1 .../>
                <vue 1.2 .../>
        </TableRow>

        <TableRow>
                 <vue 2.1 .../>
                <vue 2.2 .../>
        </TableRow>
<TableLayout>
```

- Les <**TableRow**> n'ont aucun attribut.
- Ne pas spécifier **android:layout_width** dans les vues d'un TableLayout, car c'est obligatoirement toute la largeur du tableau. Seul la balise <TableLayout> exige cet attribut.

Mise en page (layouts)

Groupes des vues RelativeLayout

C'est le plus complexe à utiliser mais il donne de bons résultats. Il permet de spécifier la position relative de chaque vue à l'aide de paramètres complexes : (LayoutParams)

- Tel bord aligné sur le bord du parent ou centré dans son parent : android:layout_alignParentTop, android:layout_centerVertical. . .
- Tel bord aligné sur le bord opposé d'une autre vue : android:layout_toRightOf, android:layout_above, android:layout_below
- Tel bord aligné sur le même bord d'une autre vue : android:layout_alignLeft, android:layout_alignTop. . .

Pour bien utiliser un **RelativeLayout**, il faut :

- Commencer par définir les vues qui ne dépendent que des bords du Layout : celles qui sont collées aux bords ou centrées.
- Puis créer les vues qui dépendent des vues précédentes.

Composantes d'interfaces

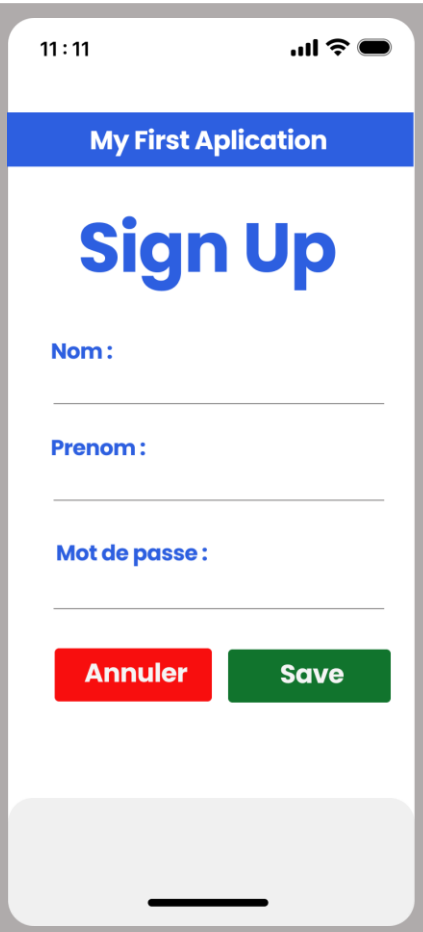
Les vues

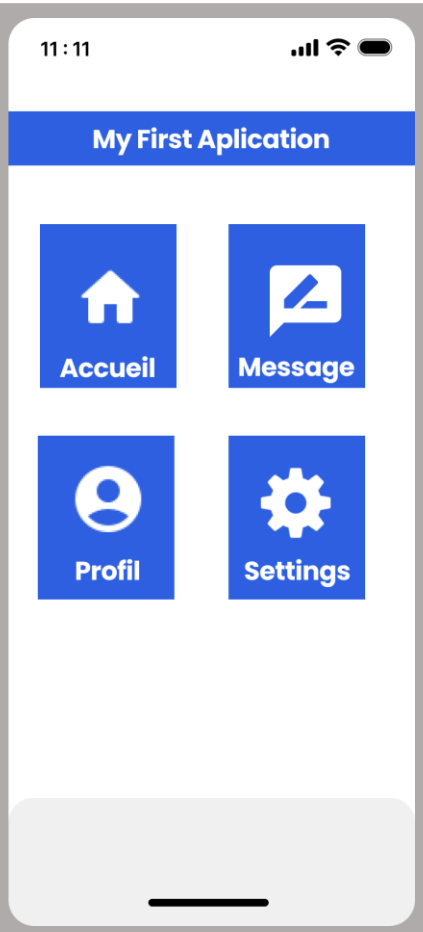
Vue	Rôle
TextView	Insérer du texte
ImageView	Insérer l'image
Button	Créer le bouton
EditText	Insérer une zone de saisie de texte
Chekbox	Créer de cases à cocher.
SearchView	Insérer une barre de recherche

Mise en page (layouts)

Arbre de vue

Exercice : Coder les interfaces suivantes





Et Voilà !

C’est fini pour cette semaine, rendez-vous la séance prochaine pour le cours sur Vie d’une application.

SEMAINE 4

VIE D'UNE APPLICATION

Le cours de cette semaine concerne la vie d'une application :

- Applications et activités, manifeste
- Cycles de vie
- Vues, événements et écouteurs.

Application et activités

Présentation

- Une **application** est composée d'une ou plusieurs *activités*. Chacune gère un écran d'interaction avec l'utilisateur et est définie par une classe Kotlin.
- Les vues d'une activité (boutons, menus, actions) permettent d'aller sur une autre activité. Le bouton *back* permet de revenir sur une précédente activité. C'est la **navigation** *entre activités*.
- Une application complexe peut aussi contenir :
  - des **services** : ce sont des processus qui tournent en arrière-plan,
  - des **fournisseurs de contenu** : ils représentent une sorte de base de données (contacts,. . .),
  - des **récepteurs d'annonces** : pour gérer des messages envoyés d'une application à une autre (notifications,…)

Application et activités

Déclaration d'une application et démarrage d'application

- Le fichier **AndroidManifest.xml** déclare les éléments d'une application
- Une activité qui n'est pas déclarée dans le manifeste ne peut pas être lancée.
- L'une des activités est marquée comme étant démarrable de l'extérieur, grâce à un sous-élément **<intent-filter>**
- Un <intent-filter> déclare les conditions de démarrage d'une activité. Celui-ci indique l'activité principale, celle qu'il faut lancer quand on clique sur son icône.

Application et activités

Démarrage d'une activité et Intents

Les activités sont démarrées à l'aide **d'intents**. Un Intent contient une demande destinée à une activité, par exemple, composer un numéro de téléphone ou lancer l'application.

- **action** : spécifie ce que l'Intent demande. Il y en a de très nombreuses :
- **VIEW** pour afficher quelque chose, EDIT pour modifier une information, SEARCH. . .
- **données** : selon l'action, ça peut être un numéro de téléphone, l'identifiant d'une information. . .
- **catégorie** : information supplémentaire sur l'action, par exemple, ...LAUNCHER pour lancer une application.

Application et activités

Démarrage d'une activité et Intents

Une application a la possibilité de lancer certaines activités d'une autre application, celles qui ont un intent-filter.

Soit une seconde application dans le package **com.infotel.MySencondApp.** Une activité peut la lancer ainsi :

```
val intent = Intent(Intent.ACTION_MAIN)
intent.addCategory(Intent.CATEGORY_LAUNCHER)
intent.setClassName("com.infotel.MySencondApp",
"com.infotel.MySencondApp. .MainActivity")
intent.flags = Intent.FLAG_ACTIVITY_NEW_TASK startActivity(intent)
```

Cela consiste à créer un Intent d'action **MAIN** et de catégorie **LAUNCHER** pour la classe **MainActivity** de l'autre application

Application et activités

Démarrage d'une activité et Intents

Soit une application contenant deux activités : **Activ1** et **Activ2**. La première lance la seconde par :

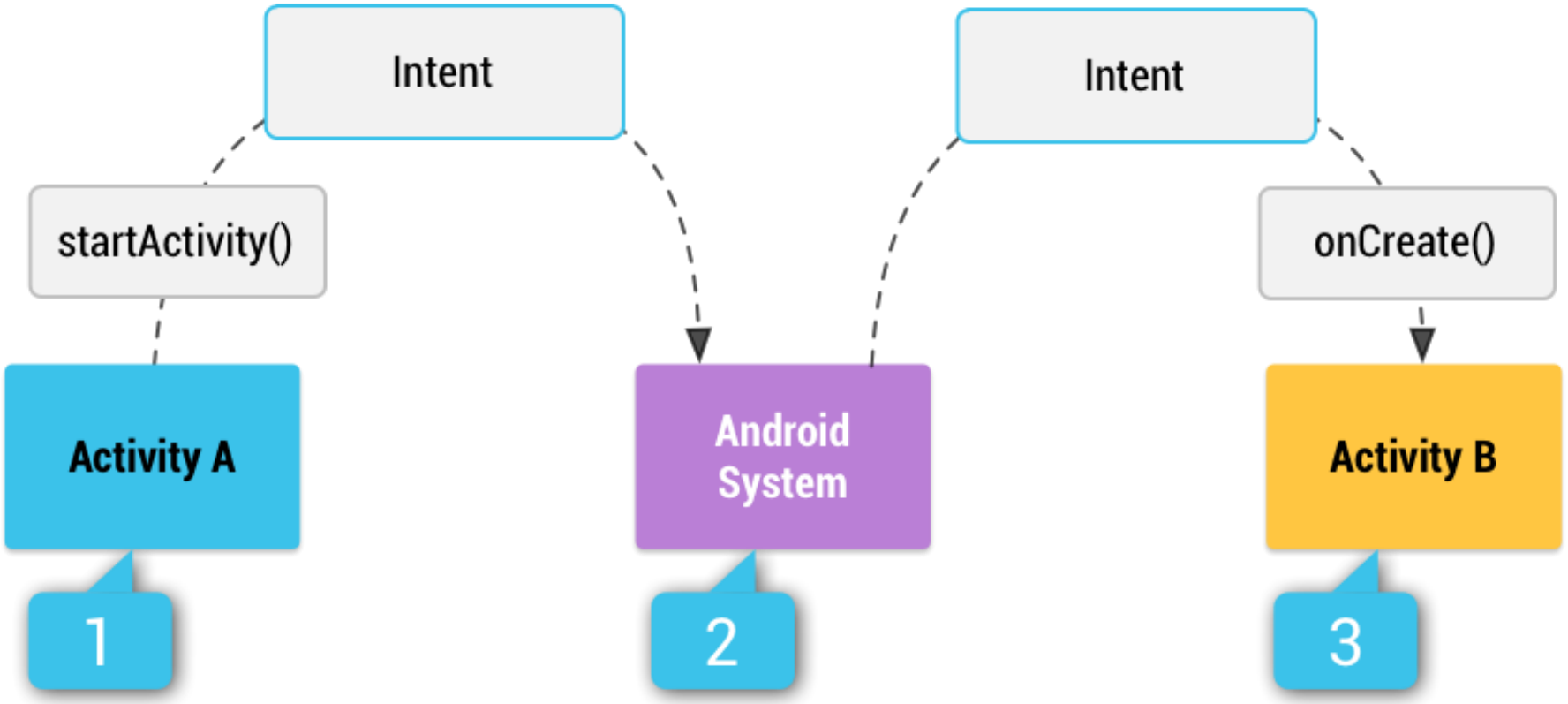
```
val intent = Intent(this, Activ2::class.java)
startActivity(intent)
```

- L'instruction **startActivity** démarre Activ2. Celle-ci se met au premier plan, tandis que Activ1 se met en sommeil.
- Activ1 reviendra au premier plan quand Activ2 se finira ou quand l'utilisateur appuiera sur back.

Ce bout de code est employé par exemple lorsqu'un bouton, un menu, etc. est cliqué. Seule contrainte : que ces deux activités soient déclarées dans AndroidManifest.xml.

Application et activités

Démarrage d'une activité et Intents



Application et activités

Les autorisations d'une application

Une application doit déclarer les autorisations dont elle a besoin : accès à internet, caméra, carnet d'adresse, GPS, etc. Cela se fait en rajoutant des éléments dans le manifeste :

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

Consulter cette page pour les permissions : http://developer.android.com/reference/android/Manifest.permission.html

Application et activités

Fonctionnement d'une application

Au début, le système Android lance l'activité qui est marquée action=MAIN et catégorie=LAUNCHER dans AndroidManifest.xml. Ensuite, d'autres activités peuvent être démarrées. Chacune se met « devant » les autres comme sur une pile. Deux cas sont possibles :

- La précédente activité se termine, on ne revient pas dedans. Par exemple, une activité où on tape son login et son mot de passe lance l'activité principale et se termine.
- La précédente activité attend la fin de la nouvelle car elle lui demande un résultat en retour. Exemple : une activité de type liste d'items lance une activité pour éditer un item quand on clique longuement dessus, mais attend la fin de l'édition pour rafraîchir la liste.

Application et activités

Navigation entre deux activités

- Rappel, pour lancer Activ2 à partir de Activ1 :

```
val intent = Intent(this, Activ2::class.java)
startActivity(intent)
```

- On peut demander la terminaison de this après lancement de Activ2 ainsi :

```
val intent = Intent(this, Activ2::class.java)
startActivity(intent)
finish()
```

finish() fait terminer l'activité courante. L'utilisateur ne pourra pas faire back dessus, car elle disparaît de la pile.

Application et activités

Terminaison d'une activité

L'activité lancée par la première peut se terminer pour deux raisons :

- Volontairement, en appelant la méthode finish()
- À cause du bouton « back » du téléphone

Dans ces deux cas, on revient dans l'activité appelante (sauf si elle-même avait fait finish().

Application et activités

Transport d'informations dans un Intent

Les **Intents** servent aussi à transporter des informations d'une activité à l'autre : les **extras**.
Voici comment placer des données dans un **Intent** :

```
val intent = Intent(this, DeleteInfoActivity::class.java)
intent.putExtra("idInfo", idInfo)
intent.putExtra("hiddencopy", hiddencopy)
startActivity(intent)
```

putExtra(nom, valeur) rajoute un couple (nom, valeur) dans l'intent. La valeur doit être sérialisable : nombres, chaînes et structures simples.

Application et activités

Extractions d'informations dans un Intent

Ces instructions récupèrent les données d'un Intent :

```
val OTP = intent.getStringExtra("OTP").toString()
val PhoneNumber = intent.getStringExtra("phoneNumber")
```

- **intent** retourne l'Intent qui a démarré cette activité.
- **getTypeExtra(nom**, valeur par défaut) retourne la valeur de ce nom si elle en fait partie, la valeur par défaut sinon.

Application et activités

Contexte d'application

Pour finir sur les applications, il faut savoir qu'il y a un objet global vivant pendant tout le fonctionnement d'une application : le **contexte d'application**. Voici comment le récupérer :

```
var context: Application = this.applicationContext as Application
```

Par défaut, c'est un objet neutre ne contenant que des informations Android.
Il est possible de le sous-classer afin de stocker des variables globales de l'application.

Activités

Présentation

Voyons maintenant comment fonctionnent les activités.

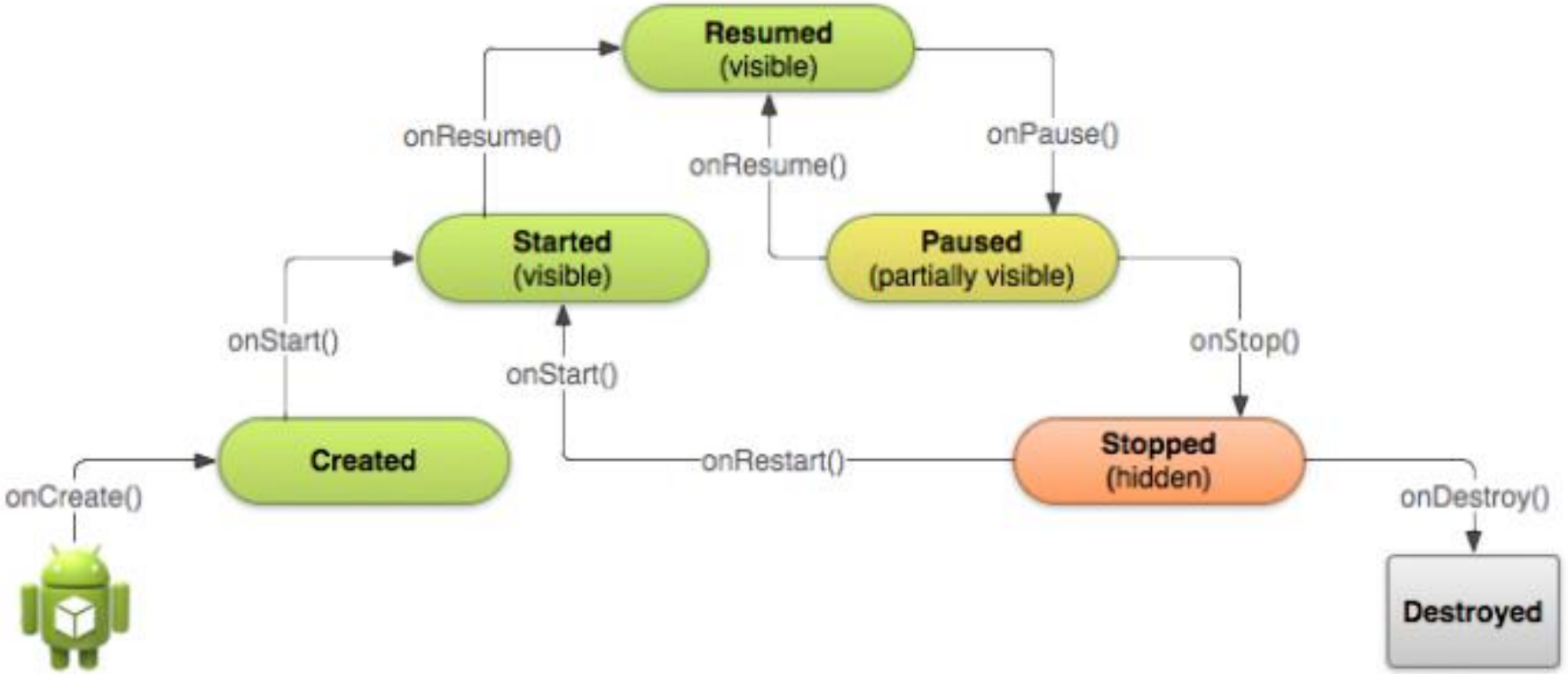
- Démarrage (à cause d'un Intent)
- Apparition/masquage sur écran
- Terminaison

Une activité se trouve dans l'un de ces états :

- active (**resumed**) : elle est sur le devant, l'utilisateur peut jouer avec,
- en pause (**paused**) : partiellement cachée et inactive, car une autre activité est venue devant,
- stoppée (**stopped**) : totalement invisible et inactive, ses variables sont préservées mais elle ne tourne plus.

Activités

Cycle de vie d'une activité



Activités

Événements de changement d'état

La classe Activity reçoit des événements de la part du système Android, ça appelle des fonctions appelées callbacks.

Exemples :

- *onCreate :* Un Intent arrive dans l'application, il déclenche la création d'une activité, dont l'interface.
- *onPause :* Le système prévient l'activité qu'une autre activité est passée devant, il faut enregistrer les informations au cas où l'utilisateur ne revienne pas.

Activités

Squelette d'une activité

```
class sign_in : AppCompatActivity() {

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

Override signifie que cette méthode remplace celle héritée de la superclasse. Il faut quand même l'appeler sur super en premier.

Vues et activités

Obtention des vues

La méthode **setContentView** charge une mise en page (layout) sur l'écran. Ensuite l'activité peut avoir besoin d'accéder aux vues, par exemple lire la chaîne saisie dans un texte. Pour cela, il faut obtenir l'objet Kotlin correspondant.

```
val btn :Button = findViewById(R.id.btn_confimer)
val code: TextView = findViewById(R.id.resendCode)
```

Cette méthode cherche la vue qui possède cet identifiant dans le layout de l'activité. Si cette vue n'existe pas (mauvais identifiant, ou pas créée), la fonction retourne null.

Vues et activités

Propriétés des vues

La plupart des vues ont des setters et getters Kotlin pour leurs propriétés XML. Par exemple TextView.
En **XML** :

```
<TextView
   android:id="@+id/PwdForgetId"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Mot de passe oublié ?"
/>
```

En Kotlin

```
val text_pwd: TextView = findViewById(R.id. PwdForgetId)
text_pwd.text ="Mot de passe oublié ?"
```

Vues et activités

Actions de l'utilisateur

Il y'a deux façons de définir l'action d'un utilisateur sur une vue:

- on peut procéder directement dans le fichier xml de la vue en insérant l'attribut **onClick()**

```
<Button
    android:onClick="onValider"
    android:id="@+id/btnValider"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/valider"/>
```

Pour cet exemple, Lorsque l'utilisateur appuie sur le bouton, ça appelle automatiquement la méthode **onValider** de l'activité grâce à l'attribut **onClick="onValider".**

Il faut définir la méthode onValider dans l'activité :

Vues et activités

Actions de l'utilisateur

```
fun onValider(view: View) {
   // votre code ici
}
```

- La deuxième façon de définir une réponse à un clic est de définir un écouteur (listener).

Un écouteur est une instance de classe implémentant l'interface View.OnClickListener qui possède la méthode public void onClick(View v)

Vues et activités

Actions de l'utilisateur

Exemple d'écouteur sur le bouton

```
val btn :Button = findViewById(R.id.btn_onValider)
btn.setOnClickListener(View.OnClickListener {

    onValider()
  }
)

fun onValider(view: View) {
   // votre code ici
}
```

Vues et activités

Actions de l'utilisateur

L'exemple précèdent est un écouteur privé. Il s'agit d'une classe qui est définie à la volée, lors de l'appel à **setOnClickListener**. Elle ne contient qu'une seule méthode.
Il est intéressant de transformer cet écouteur en expression **lambda** qui une fonction sans nom écrite ainsi :

(paramètres avec ou sans types) -> expression
(paramètres avec ou sans types) -> { corps }

Cette transformation de l'écouteur est possible parce que l'interface **View.OnClickListener** ne possède qu'une seule méthode. On parle alors **d'écouteur privé anonyme**

Vues et activités

Actions de l'utilisateur

Exemple :

```
btn.setOnClickListener { btn: View? ->
    //code ici
}
```

Il existe une autre façon de définir un écouteur: L'activité elle-même en tant qu'écouteur. Il suffit de mentionner **this** comme écouteur et d'indiquer qu'elle implémente l'interface **OnClickListener**.

L'exemple est donné ci-dessous:

Vues et activités

Actions de l'utilisateur

Exemple :

```
class HomeActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_home)

       btn.setOnClickListener(this)
}

fun OnClickListener(btn : View?){

}
```

Ici, par contre, tous les boutons appelleront la même méthode.

Vues et activités

Actions de l'utilisateur

Dans le cas où le même écouteur est employé pour plusieurs vues, il faut les distinguer en se basant sur leur identifiant obtenu avec **Id** :

```
fun OnClickListener(v:View){
        when (v.id) {
            R.id.supprimer -> {
                supprimer()
                true
            }
            R.id.modifier -> {
                modifier()
                true
            }
            else -> {
                // autres methodes
                true
            }
    }
}
```

Vues et activités

Actions de l'utilisateur

Il y a une dernière façon très pratique d'associer un écouteur, avec une référence de méthode, c'est à dire son nom précédé de l'objet qui la définit.

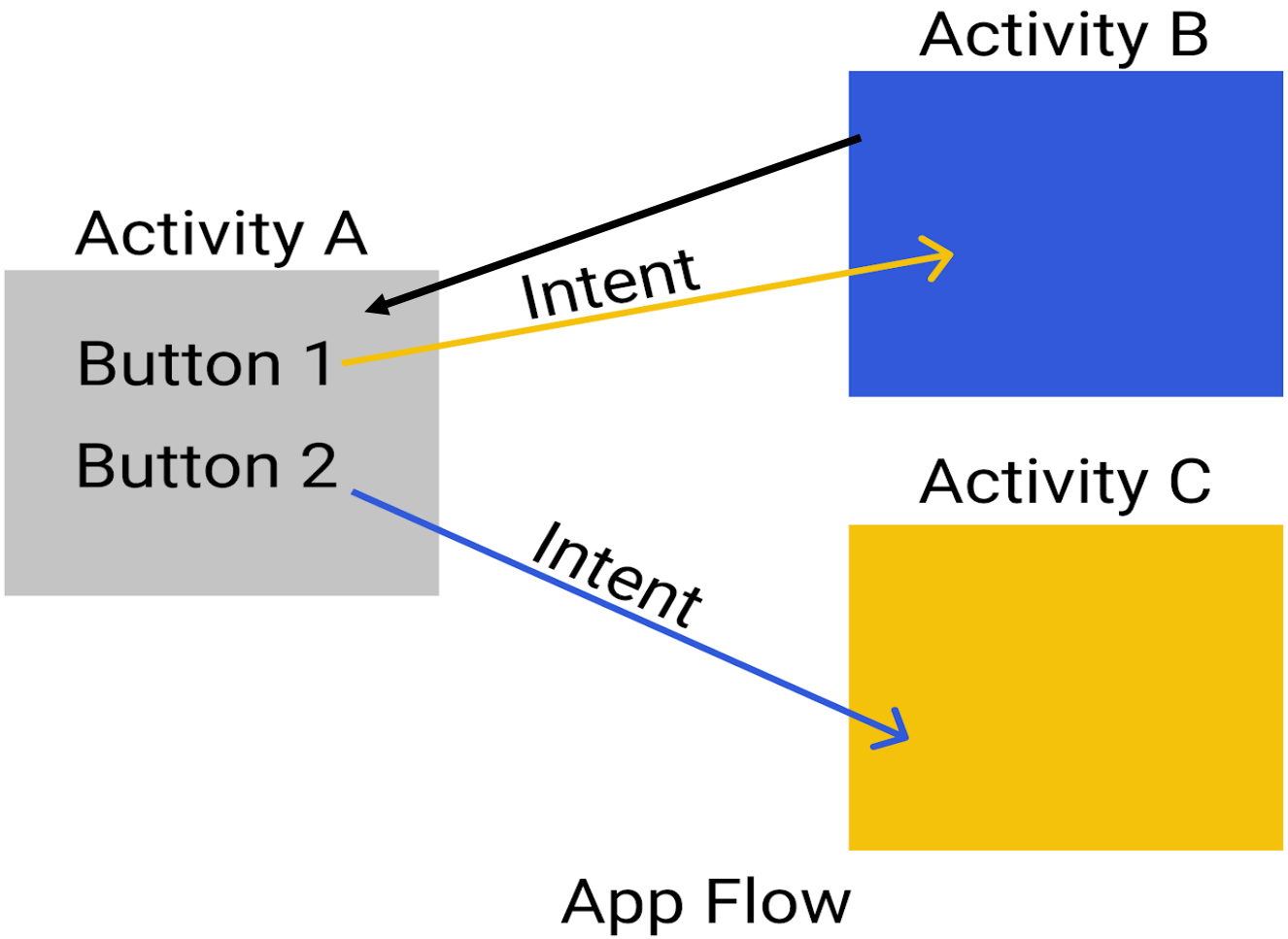
```
class HomeActivity : AppCompatActivity() {
   override fun onCreate(savedInstanceState: Bundle?) {
      super.onCreate(savedInstanceState)
      setContentView(R.layout.activity_home)

     btn.setOnClickListener(this::OnBtnClick)
}

fun OnBtnClick(btn : View?){

}
```

La syntaxe **this::nom_methode** est une simplification de l'expression lambda (params) -> nom_methode(params)

Exercice

Description:
Cet exercice est une simple application Android avec trois activités, la première activité est l'activité principale avec deux boutons. Bouton 1 : vous amène à l'activité B lorsque vous cliquez dessus. Bouton 2 : vous amène à l'activité C lorsque vous cliquez dessus.
Activité B : est la page contenant un formulaire de connexion destiné à collecter les informations suivantes: login, mot de passe et contient deux boutons (connexion et annuler). En cliquant sur le bouton connexion , l'utilisateur est redirigé vers la page A en affichant le message « Connexion reussite » si les informations ne sont pas vides. Sinon afficher un message pour lui demander de renseigner les informations
L'activité C est votre profil, elle contient une **ImageView** qui devrait afficher votre photo, **TextViews** indiquant votre nom, piste, pays, e-mail et numéro de téléphone.

Exercice

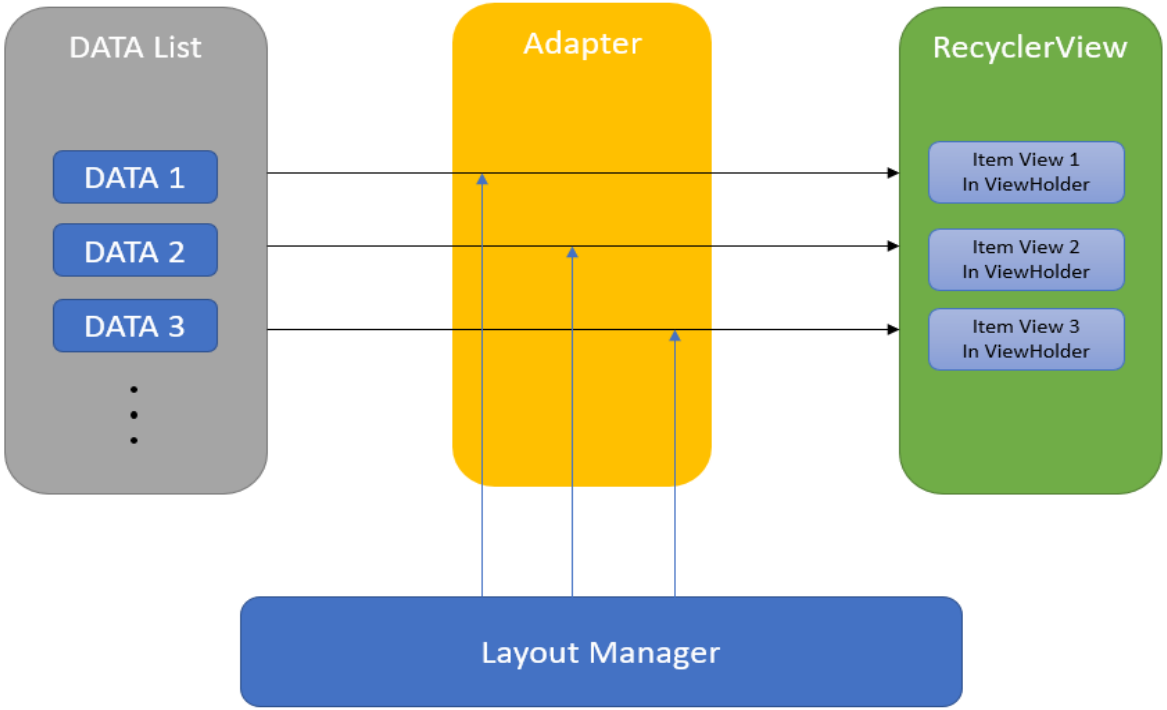

Activity B
Activity A
Intent
Button 1
Button 2
Activity C
Intent
App Flow

Et Voilà !

C’est fini pour aujourd’hui. C’est assez pour cette semaine, rendez-vous la semaine prochaine pour un cours sur les applications listes.

SÉANCE 5

APPLICATION LISTE



Durant cette semaine prochaine, nous allons nous intéresser aux applications de gestion d'une liste d'items.

- Stockage d'une liste
- Affichage d'une liste, adaptateurs
- Consultation et édition d'un item

Présentation

Principe général

- On veut programmer une application pour afficher et éditer une liste d'items.
- La liste est stockée dans un tableau type **ArrayList**
- L'écran est occupé par un **Listview** ou **RecyclerView**. Ce sont des vues spécialisées dans l'affichage de listes quelconques.
- Il existe deux façon de faire des listes dans Android avec Kotlin :
  - Liste simple (basé sur les adaptateurs simples) : On utilise **Listview**
  - Liste personnalisée (basée sur les adaptateurs personnalisé) : On utilise **RecyclerView**

Présentation

Principe général

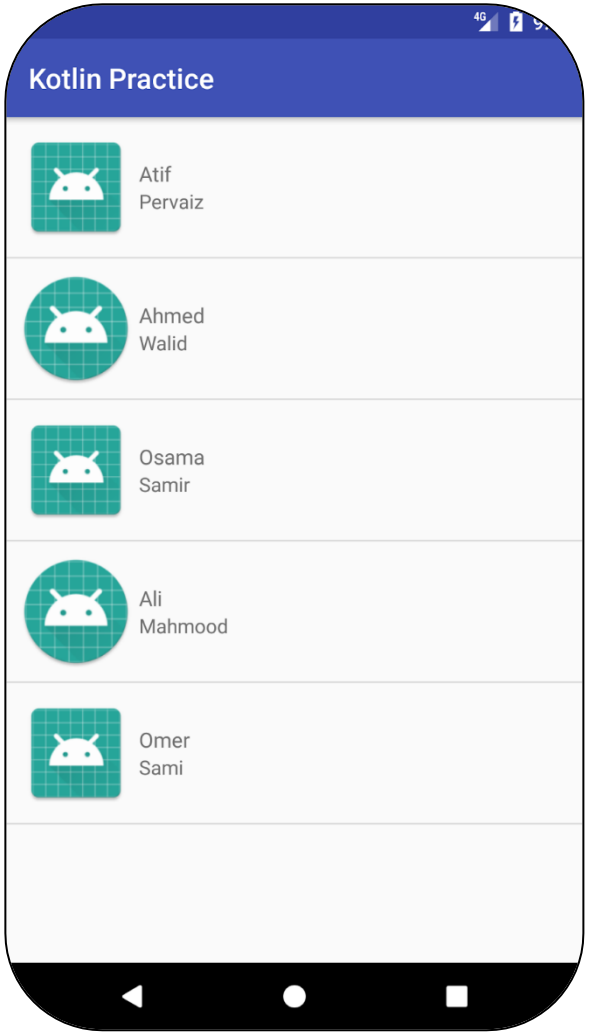
Définition

Android ListView est une vue utilisée pour afficher les éléments d'un tableau sous forme de liste défilante.

Présentation

Principe général

Liste simple

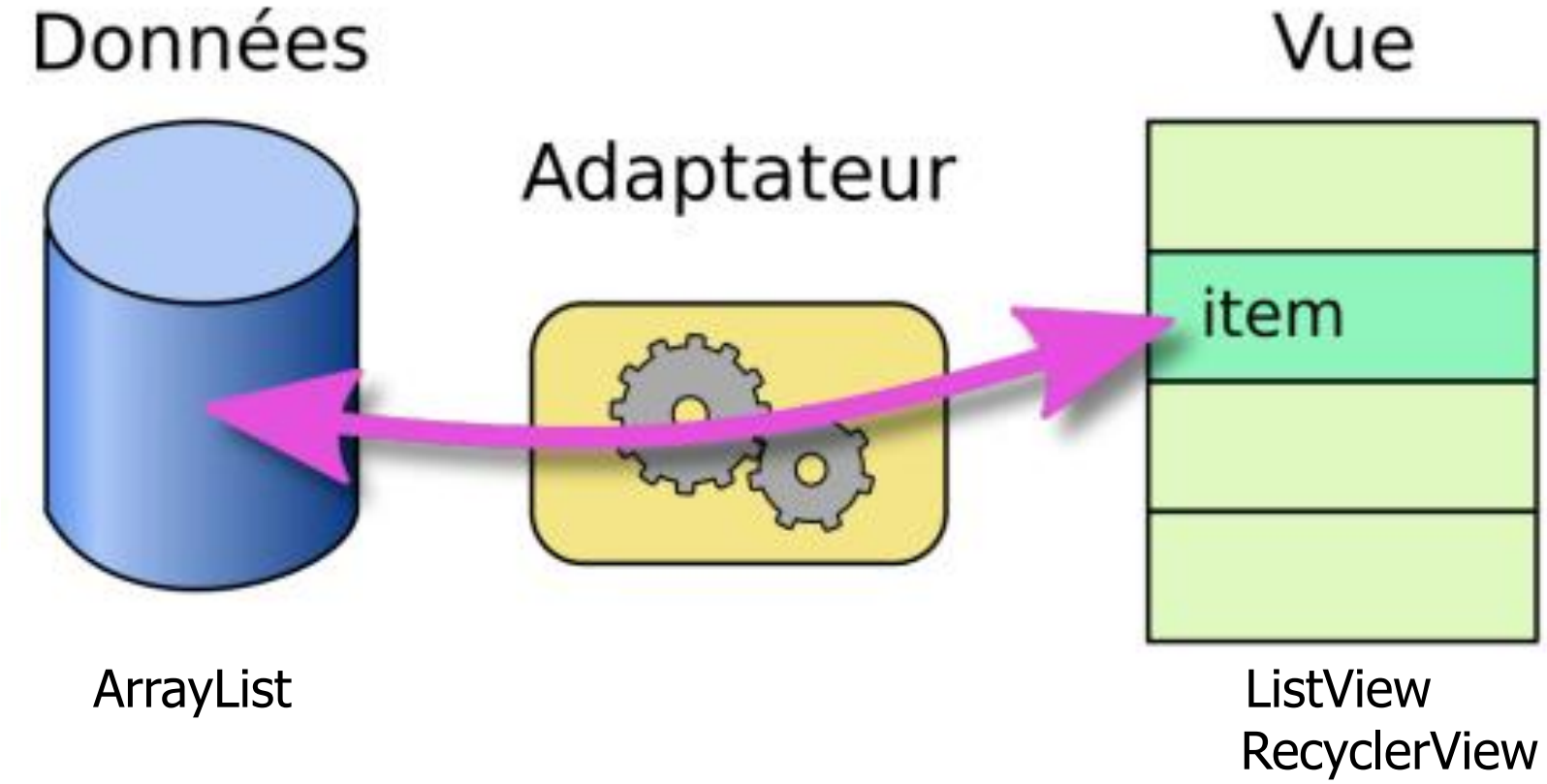


Liste personnalisée

Présentation

Schéma global

- Modèle MVC : le contrôleur entre les données et la vue s'appelle un adaptateur.



Présentation

Représentation des données initiales

- Pour commencer, il faut représenter les données :
  - Utiliser les classes pour une liste personnalisée avant de stocker les données dans un **ArrayList**
  - Pas besoin des classes pour une liste simple. On peut directement stocker les données initiales dans un tableau.

```
//création de la classe Personne
class Personne (var Age:Int, var Nom:String) {
}
// création des instances de Personne
val P1=Personne (12,Banane)
val P2=Personne (22,INFO)

// création d'une liste de noms des Personnes
val ListePersonne : ArrayList<Personne>
var ListePersonne = arrayOf(P1.Nom, P2.Nom)
```

Présentation

Représentation des données initiales

NB: ArrayList Crée une liste non modifiable.
C'est un type de données générique, c'est à dire paramétré par le type des éléments mis entre <. . . > ; ce type doit être un objet (String, Int, Classe, …).
Quelques méthodes utiles de la classe abstraite **List**, héritées par **ArrayList** :

Méthodes	fonctions
liste.size()	retourne le nombre d'éléments présents,
liste.clear()	supprime tous les éléments
liste.add(elem) :	ajoute cet élément à la liste,
liste.remove(elem ou indice) :	retire cet élément
liste.get(indice) :	retourne l'élément présent à cet indice,
liste.contains(elem) :	true si elle contient cet élément,
liste.indexOf(elem) :	indice de l'élément, s'il y est

Présentation

Représentation des données initiales dans les ressources

On crée deux tableaux dans le fichier res/values/arrays.xml :

```
<resources>
   <string-array name="Noms">
      <item>Jean</item>
      <item>Abbas</item>
      ...
   </string-array>
   <integer-array name="Age">
      <item>18</item>
      <item>108</item>
      ...
   </integer-array>
</resources>
```

Présentation

Représentation des données initiales dans les ressources

Ensuite, on récupère ces ressources tableaux pour remplir le ArrayList :

```
// accès aux ressources
// accès aux ressources
val res: Resources = resources
val noms: Array<String> = res.getStringArray(R.array.Noms)
val ages: IntArray = res.getIntArray(R.array.Age)
// recopie dans le ArrayList
liste = ArrayList()
for (i in noms.indices) {
   liste.add(Personne(noms[i], ages[i]))
}
```

Ça semble plus complexe, mais c'est préférable à la solution du tableau pré-initialisé pour la séparation entre programme et données

Affichage de la liste

Activité

L'affichage de la liste est fait par un **ListView** ou **RecyclerView**. Ce sont des vues qui intègre un défilement automatique et qui veille à économiser la mémoire pour l'affichage.

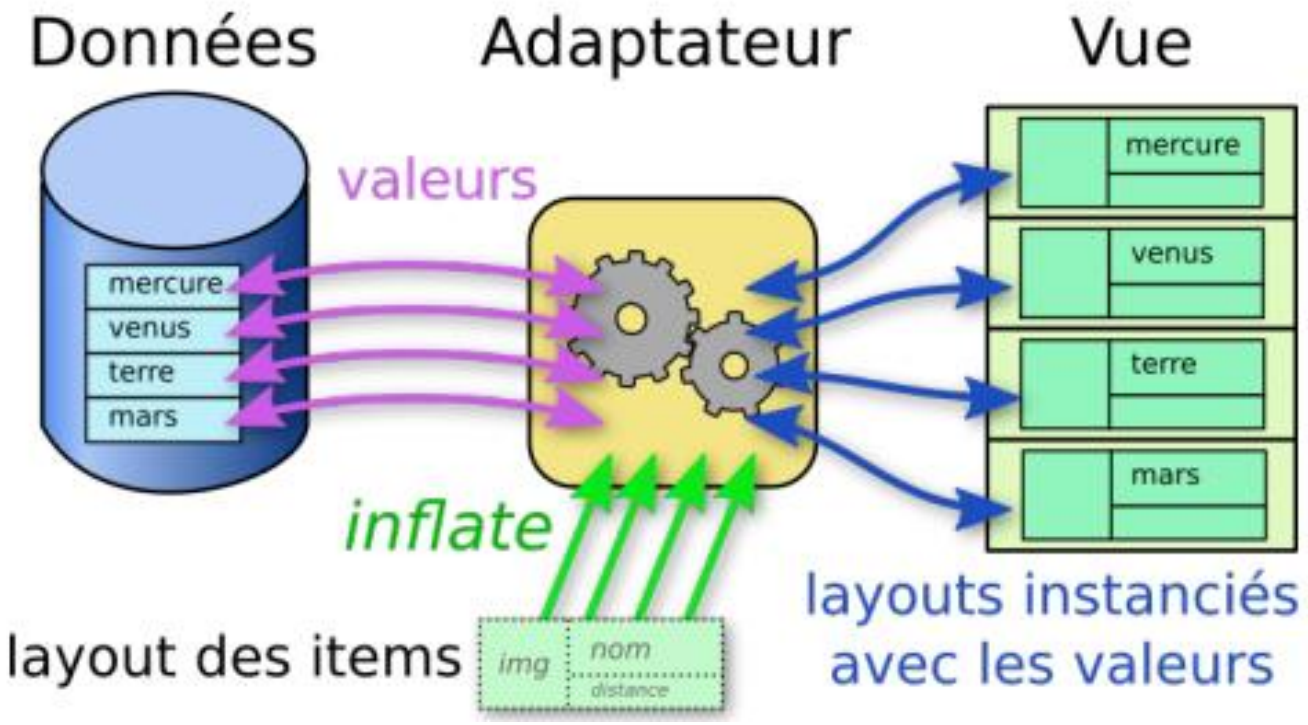
```
<ListView
        android:id="@+id/listview"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />
```

```
<androidx.recyclerview.widget.RecyclerView
        xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/recycler"
        android:layout_width="match_parent"
        android:layout_height="match_parent" />
```

Adaptateurs et ViewHolders

Relations entre la vue et les données

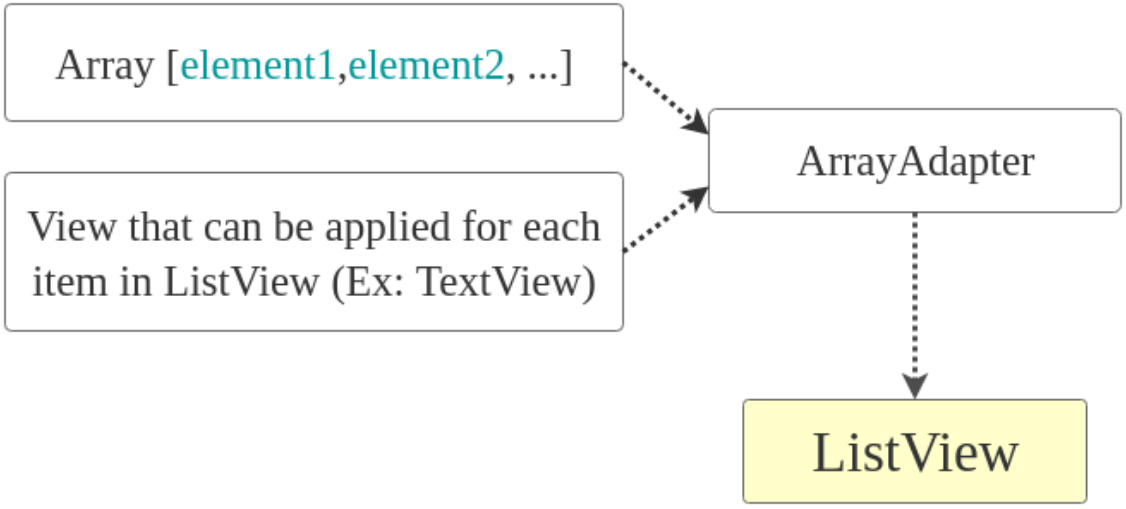
ListView ou **RecyclerView** affiche les items d'une liste à l'aide d'un adaptateur :



Adaptateurs et ViewHolders

Adaptateur pour une liste simple

- On utilise un **ArrayAdapter** (qui est un adaptateur de tableau) pour une afficher une liste simple.
- Initialisez ArrayAdapter avec le **contexte** d'application, la **ressource** (layout des items) à utiliser comme vue pour chaque élément de la liste et le **tableau** d'éléments lui-même comme arguments.
- Définissez l'adaptateur créé à l'étape précédente sur ListView.



Adaptateurs et ViewHolders

Adaptateur pour une liste simple

```
 //création d'un adaptateur de tableau
  val adapter = ArrayAdapter(this,R.layout.listview_item, array)

 //Référencement de Listview à utiliser
  val listView : ListView = findViewById(R.id.listview)

//définission de l'adapteur sur Listview
  listView.setAdapter(adapter)
```

Adaptateurs et ViewHolders

Adaptateur pour une liste simple

Exercice

Créer une application Android qui affiche la liste des 10 pays africains de votre choix.

NB: utilisez une liste simple.

Adaptateurs et ViewHolders

RecyclerView

Définition

RecyclerView est un widget d'affichage de liste avancé disponible dans la bibliothèque de support d'Android. Il permet de gérer de grandes quantités de données de manière efficace en réutilisant les éléments de la liste qui ne sont plus visibles pour économiser les ressources de la mémoire.
Il offre une flexibilité et une personnalisation accrues par rapport à la ListView, qui était le widget d'affichage de liste standard dans les versions antérieures d'Android. Le RecyclerView peut être utilisé pour afficher une liste verticale, horizontale ou en grille, avec des mises en forme personnalisées pour chaque élément de la liste.

Adaptateurs et ViewHolders

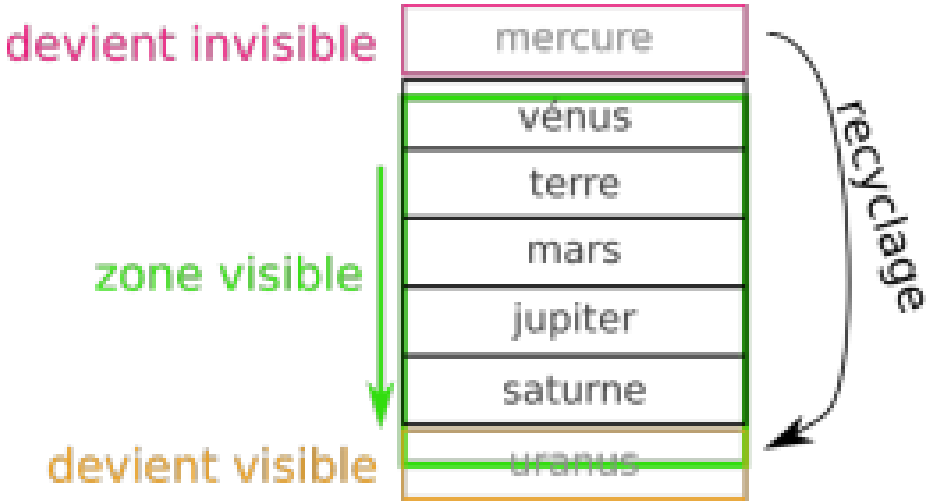
RecyclerView

- La vue ne sert qu'à afficher les éléments de la liste. En réalité, seuls quelques éléments seront visibles en même temps. Cela dépend de la hauteur de la liste et la hauteur des éléments.
- Le principe du **RecyclerView** est de ne gérer que les éléments visibles. Ceux qui ne sont pas visibles ne sont pas mémorisés. Mais lorsqu'on fait défiler la liste ainsi qu'au début, de nouveaux éléments doivent être rendus visibles.
- Le **RecyclerView** demande alors à l'adaptateur de lui instancier (*inflate*) les vues pour afficher les éléments.
- Le nom « **RecyclerView** » vient de l'astuce : les vues qui deviennent invisibles à cause du défilement vertical sont recyclées et renvoyées de l'autre côté mais en changeant seulement le contenu à afficher.

Adaptateurs et ViewHolders

Adaptateur pour une liste personnalisée : RecyclerView

- Une vue qui devient invisible d'un côté, à cause du scrolling, est renvoyée de l'autre côté, comme sur un tapis roulant, en modifiant seulement son contenu : Il économise de la mémoire en réutilisant les vues lorsque vous faites défiler une activité au lieu de les créer toutes au début lors du premier chargement de l'activité.



Adaptateurs et ViewHolders

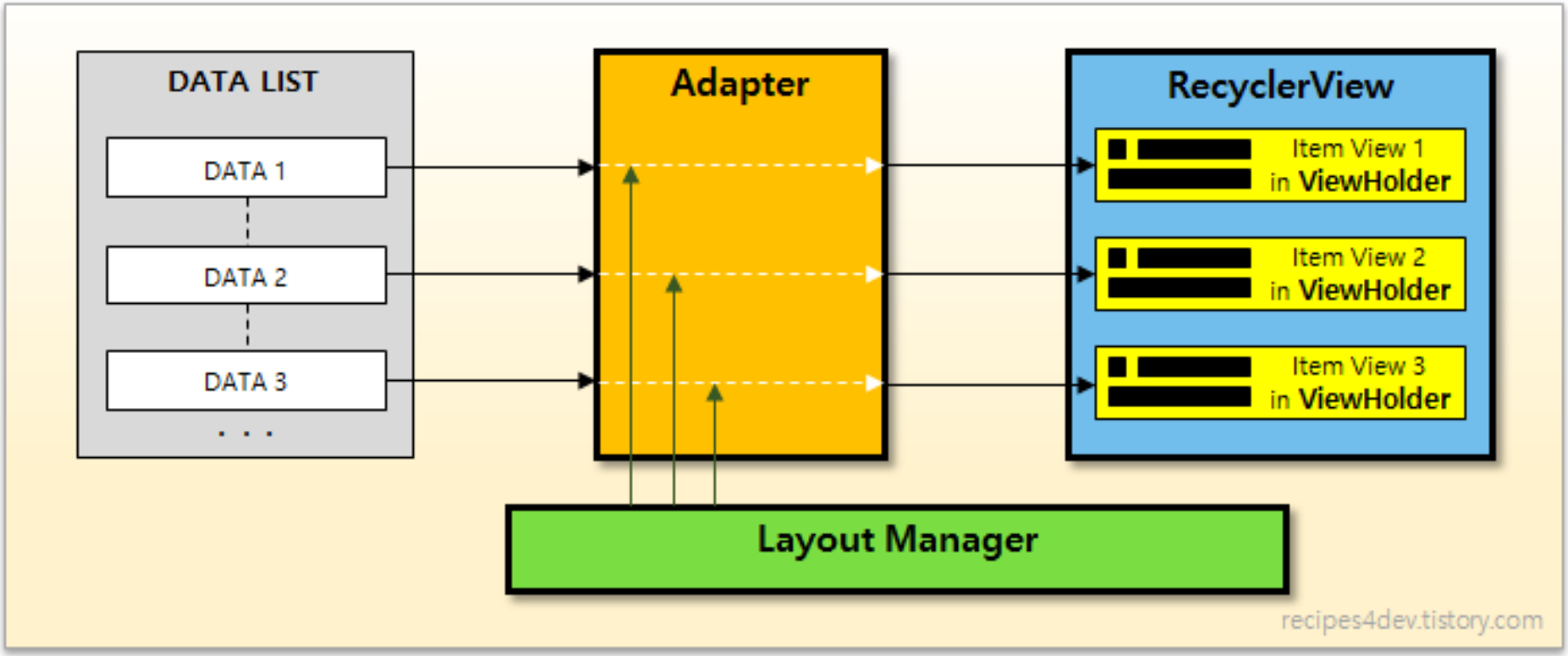
Adaptateur pour une liste personnalisée : RecyclerView

- **RecyclerView** comporte trois parties principales : la mise en page, le ViewHolder et l'adaptateur.
- La **mise en page** est la vue qui sera créée pour chaque élément à charger dans RecyclerView (Layout des items).
- Un **ViewHolder** est utilisé pour mettre en cache les objets de vue afin d'économiser de la mémoire.
- **L'adaptateur** crée de nouveaux éléments sous la forme de ViewHolders, remplit les ViewHolders avec des données et renvoie des informations sur les données.
- Pour utiliser **RecyclerView** il faut ajouter la dépendance du projet recyclerview-v7 dans le fichier **build.gradle** (Module : app).

```
implémentation 'com.android.support:recyclerview-v7:26.1.0'
```

Adaptateurs et ViewHolders

Adaptateur pour une liste personnalisée : RecyclerView



Adaptateurs et ViewHolders

La mise en page

Créez un layout pour chaque élément de la liste. Exemple : planet_item.xml:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="wrap_content"
android:orientation="horizontal">
<ImageView
   android:id="@+id/planetImage"
   android:layout_width="50dp"
   android:layout_height="50dp" />
<TextView
   android:id="@+id/planetName"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:textSize="20sp"
   android:textStyle="bold"
   android:layout_marginStart="16dp" />
</LinearLayout>
```

Adaptateurs et ViewHolders

ViewHolder

- Créez une classe de base (par exemple Planète) pour stocker les données pour chaque élément de la liste:

```
data class Planet(val name: String, val image: Int)
```

- Créez un Adapter pour le RecyclerView qui étendra RecyclerView.Adapter. Il va contenir la classe **ViewHolder** qui est associée à une donnée de base, ex: une planète et qui permet son affichage dans un RecyclerView.

Adaptateurs et ViewHolders

ViewHolder

- La classe **ViewHolder** contient les méthodes suivantes:
  - **onCreateViewHolde** :méthode qui crée les ViewHolder , Elle est appelée au début de l'affichage de la liste, pour initialiser ce qu'on voit à l'écran.
    inflate = transformer un fichier XML en vues Kotlin.
  - **getItemCount**: Elle qui retourne le nombre d'éléments
  - **onBindViewHolder** : méthode qui recycle les ViewHolder . Cette méthode est appelée pour remplir un ViewHolder avec l'un des éléments de la liste, celui qui est désigné par position (numéro dans la liste à l'écran). C'est très facile avec le setter.

Adaptateurs et ViewHolders

ViewHolder

```
class PlanetAdapter(private val planets: List<Planet>) :
    RecyclerView.Adapter<PlanetAdapter.ViewHolder>() {

    class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        fun bind(planet: Planet) {
            itemView.planetName.text = planet.name
            itemView.planetImage.setImageResource(planet.image)
        }
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.planet_item, parent, false)
        return ViewHolder(view)
    }

    override fun onBindViewHolder(holder: ViewHolder, position: Int) {
        holder.bind(planets[position])
    }

    override fun getItemCount() = planets.size
}
```

Adaptateurs et ViewHolders

Définition de l'adaptateur

- Ajoutez le RecyclerView à votre layout principal et définissez l'Adapter:

```
<androidx.recyclerview.widget.RecyclerView
android:id="@+id/recyclerView"
android:layout_width="match_parent"
android:layout_height="match_parent" />
```

```
val planets = listOf(
    Planet("Mercury", R.drawable.mercury),
    Planet("Venus", R.drawable.venus),
    Planet("Earth", R.drawable.earth),
    // ...
)

val adapter = PlanetAdapter(planets)
recyclerView.adapter = adapter
recyclerView.layoutManager = LinearLayoutManager(this)
```

Adaptateurs et ViewHolders

Définition de l'adaptateur

Exercice

Refaire la liste de l'application précédente, on insérant cette fois ci dans la liste le nom du pays suivi de son drapeau et ainsi que sa zone. L'exemple est donné ci-dessous:

Cameroun

Afrique centrale

Actions sur la liste

Clic sur un élément

- Avec ce qui précède, la liste s'affiche automatiquement. On s'intéresse maintenant à un clic sur un élément de la liste. Dans Android, rien n'est prévu pour les clics sur les éléments. On doit construire soi-même une architecture d'écouteurs.

Voici d'abord la situation, pour comprendre la solution :

- 1. L'activité MainActivity veut être prévenue quand l'utilisateur clique sur un élément de la liste.
- 2. L'objet qui reçoit les événements utilisateur est le ViewHolder. Il lui suffit d'implémenter la méthode onClick(View v) de l'interface View.OnClickListener comme un simple Button pour être prévenu d'un clic sur lui.

Actions sur la liste

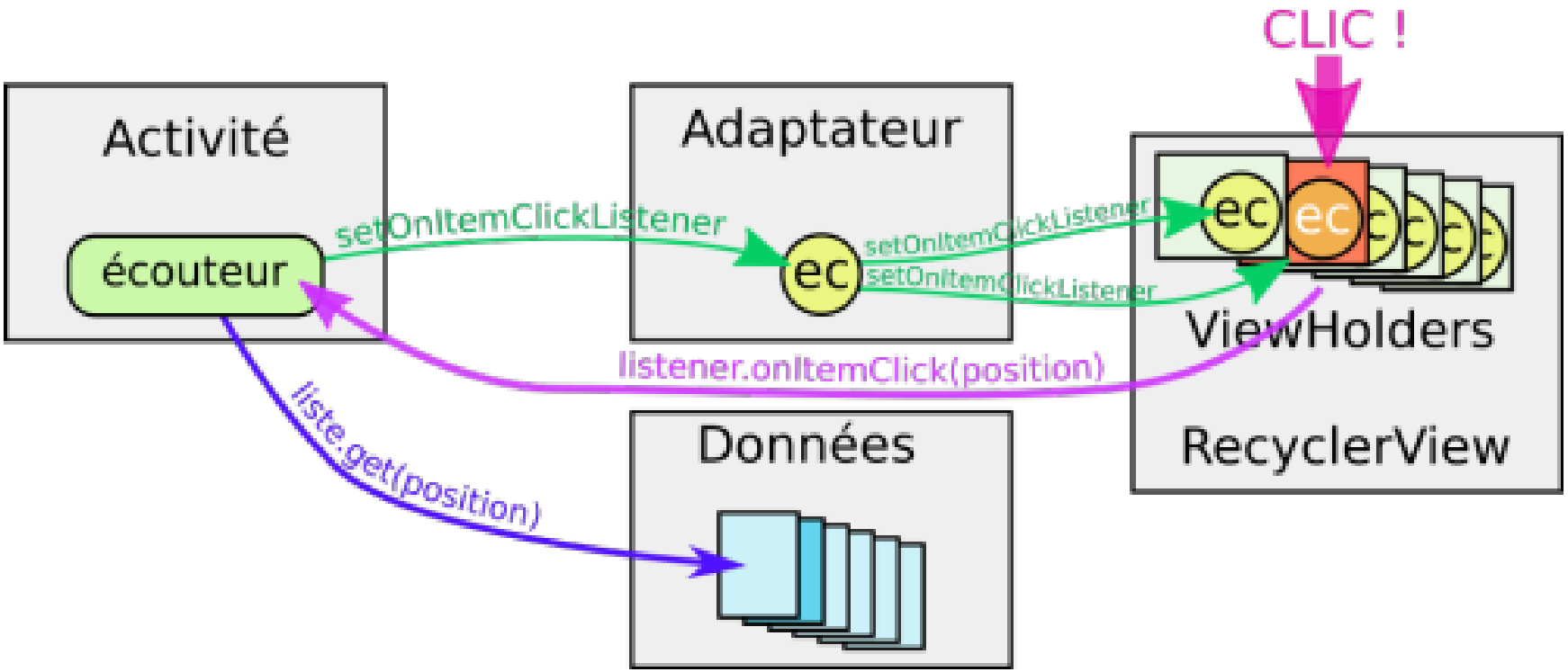
Clic sur un élément

- 3. Un RecyclerView regroupe plusieurs ViewHolder ; chacun peut être cliqué (un à la fois). Celui qui est cliqué peut faire quelque chose, mais son problème, c'est qu'il ne connaît pas l'activité. Il faut donc faire le lien entre ces ViewHolder et l'activité. Ça va passer par l'adaptateur, le seul qui soit au contact des deux.

1. Il faut que l'activité définisse un écouteur de clics et le fournisse à l'adaptateur. Tant qu'à faire, on peut définir notre propre sorte d'écouteur qui recevra la position de l'objet cliqué en paramètre (c'est le plus simple à faire).
2. L'adaptateur transmet cet écouteur à tous les ViewHolder qu'il crée ou recycle.
3. Chaque ViewHolder possède donc cet écouteur et peut le déclencher le cas échéant.

Actions sur la liste

Clic sur un élément



Actions sur la liste

Définition d'un écouteur de clic

- utiliser un OnClickListener dans votre ViewHolder:

```
class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
    fun bind(planet: Planet) {
        itemView.planetName.text = planet.name
        itemView.planetImage.setImageResource(planet.image)
        itemView.setOnClickListener {
            // traitement lors du clic sur un élément de la liste
        }
    }
}
```

Actions sur la liste

Définition d'un écouteur de clic

On récupère ici le numéro de l'élèment dans la liste

```
class ViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView), View.OnClickListener {
    fun bind(planet: Planet) {
        itemView.planetName.text = planet.name
        itemView.planetImage.setImageResource(planet.image)
        itemView.setOnClickListener(this)
    }

    override fun onClick(v: View?) {
        val position = adapterPosition
        Toast.makeText(v?.context, "Element $position cliqué", Toast.LENGTH_SHORT).show()
    }
}
```

Actions sur la liste

Définition d'un écouteur de clic

Exercice

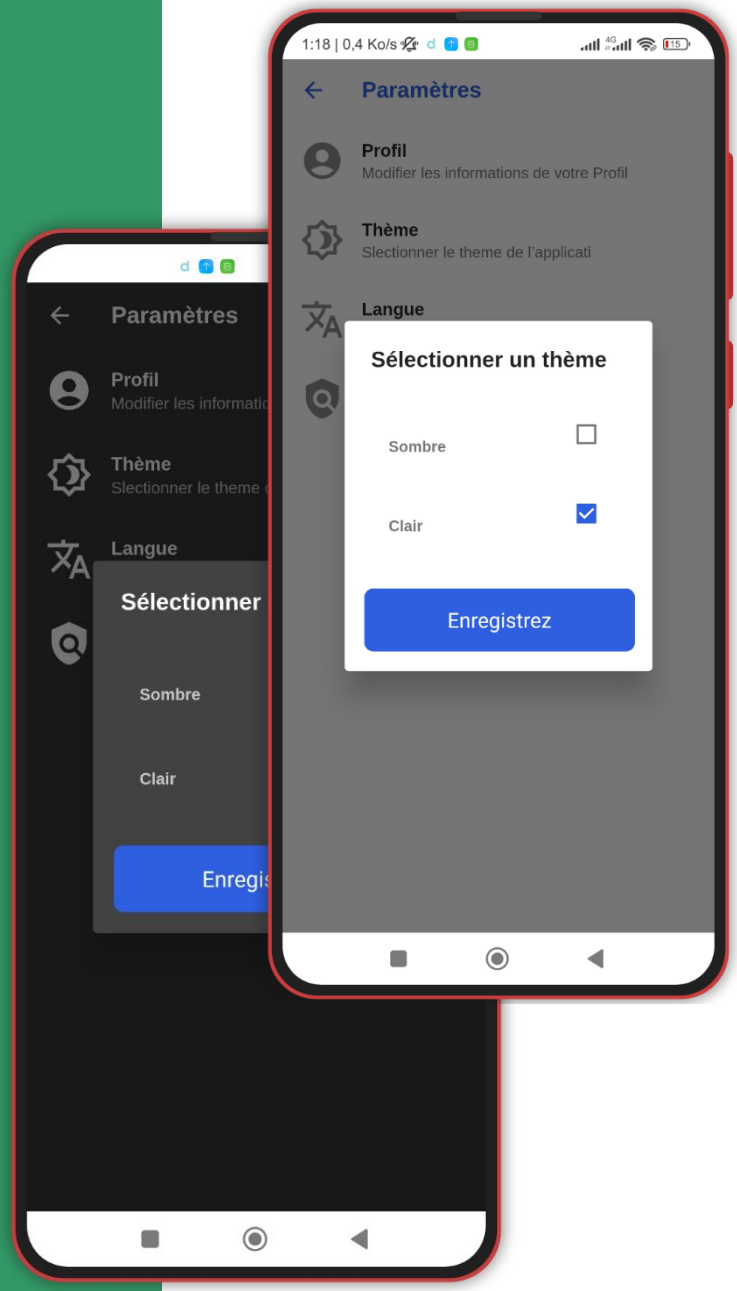
Modifier la liste précédente de telle sorte que le nom du pays et son numéro dans la liste puisse etre afficher à l'utilisateur après avoir cliqué dessus.

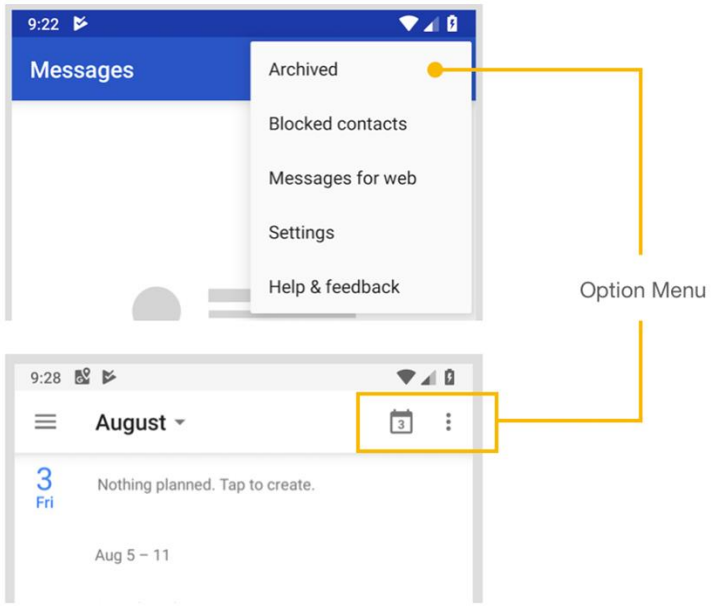
Ouf, c’est fini !

C’est tout pour cette séance. La séance prochaine nous parlerons des menus, dont les menus contextuels qui apparaissent quand on clique longuement sur un élément d’une liste, dialogues et fragments

SÉANCE 6

ERGONOMIE





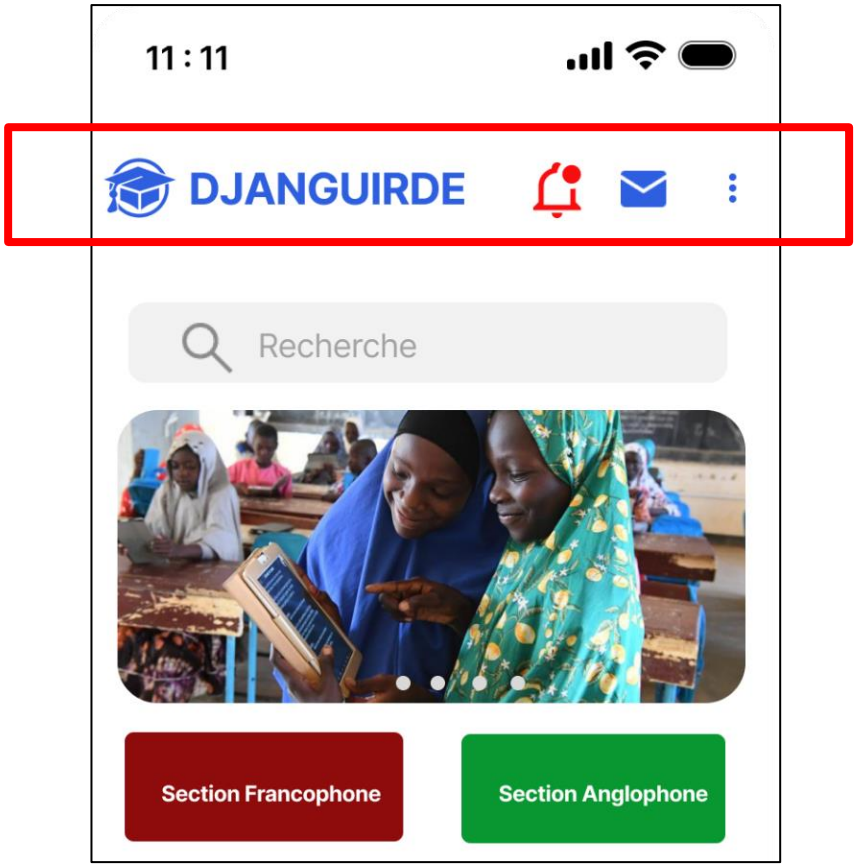
Le cours de cette séance concerne l'ergonomie d'une application Android.

- Menus et barre d'action
- Pop-up : messages et dialogues
- Activités et fragments
- Préférences (pour info)

Barre d’action et menus

Barre d’action

- La barre d’action contient l’icône et le nom de d’application, quelques items de menu et un bouton pour avoir les autres menus .



Barre d'action et menus

Le menu

- Un menu est une liste de d'items présentés dans la barre d'action. La sélection d'un item déclenche une callback.

- Pour créer un menu sous Android, il faut définir :
  - un fichier res/menu/nom_du_menu.xml qui est une sorte de layout spécialisé pour les menus,

  - deux méthodes d'écouteur pour gérer les menus :
    - ajout du menu dans la barre,
    - activation de l'un des items.

Barre d'action et menus

Le menu

- Exemple de fichier menu.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/menu_item_1"
    android:title="Menu Item 1"
    android:icon="@drawable/ic_menu_item_1"
    android:showAsAction="ifRoom" />
  <item
    android:id="@+id/menu_item_2"
    android:title="Menu Item 2"
    android:icon="@drawable/ic_menu_item_2"
    android:showAsAction="ifRoom" />
  <item
    android:id="@+id/menu_item_3"
    android:title="Menu Item 3"
    android:icon="@drawable/ic_menu_item_3"
    android:showAsAction="never" />
</menu>
```

Barre d'action et menus

Le menu

L'attribut **showAsAction** définit comment une option de menu doit être affichée dans l'interface utilisateur. Il peut avoir les valeurs suivantes :

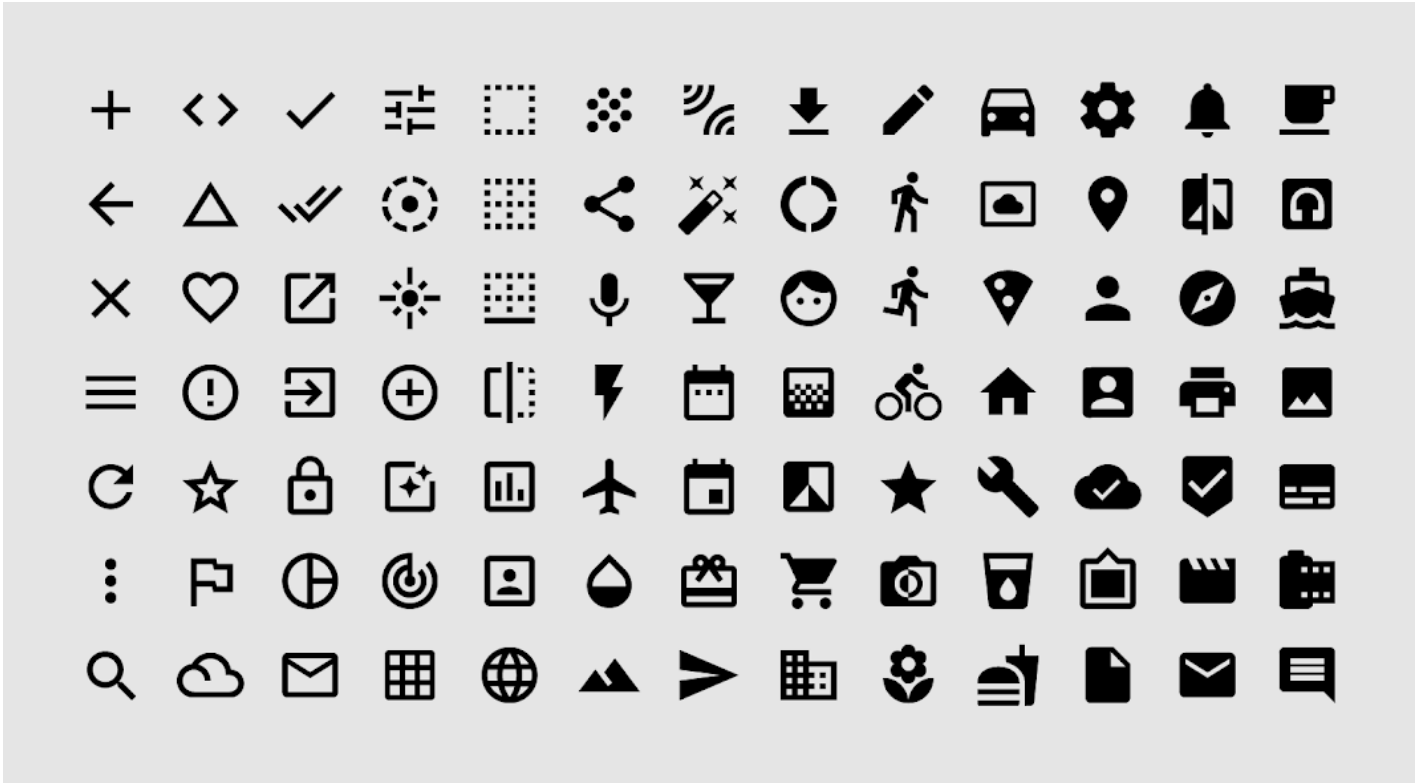
- always : l'option de menu sera toujours affichée en tant qu'icône dans la barre d'actions.
- ifRoom : l'option de menu sera affichée en tant qu'icône dans la barre d'actions si elle peut s'y insérer. Sinon, elle sera affichée dans le menu déroulant.
- never : l'option de menu sera toujours affichée dans le menu déroulant, jamais en tant qu'icône dans la barre d'actions.

Cet attribut est à modifier en app:showAsAction si on utilise androidx.

Barre d'action et menus

Le menu : les icones

Android distribue gratuitement un grand jeu d'icônes pour les menus, dans les deux styles **MaterialDesign** : HoloDark et HoloLight.



Barre d'action et menus

Les écouteurs pour afficher les menus

Pour afficher un menu dans une activité Android en Kotlin, vous pouvez utiliser la méthode **onCreateOptionsMenu** de votre activité. Cette méthode est appelée lorsque le système crée le menu pour la première fois. Vous pouvez y définir les options de menu à afficher en utilisant la méthode **menuInflater.inflate**. Voici un exemple de code pour créer un écouteur qui affiche un menu :

```
override fun onCreateOptionsMenu(menu: Menu): Boolean {
  // Inflate the menu; this adds items to the action bar if it is present.
   menuInflater.inflate(R.menu.main_menu, menu)
   return true
}
```

Barre d’action et menus

Réactions aux sélections d’items

- Lorsqu'un utilisateur sélectionne une option de menu dans votre application Android en Kotlin, vous pouvez réagir à cette sélection en implémentant la méthode onOptionsItemSelected de votre activité.

- Cette méthode est appelée lorsque l'utilisateur sélectionne une option de menu et prend en entrée un objet MenuItem qui représente l'option de menu sélectionnée.

Barre d’action et menus

Réactions aux sélections d’items

```
class MainActivity : AppCompatActivity() {
    override fun onCreateOptionsMenu(menu: Menu): Boolean {
        // Inflate the menu; this adds items to the action bar if it is present.
        menuInflater.inflate(R.menu.main_menu, menu)
        return true
    }

    override fun onOptionsItemSelected(item: MenuItem): Boolean {
        // Handle item selection
        return when (item.itemId) {
            R.id.menu_item_1 -> {
                // Action to perform when menu item 1 is selected
                true
            }
            R.id.menu_item_2 -> {
                // Action to perform when menu item 2 is selected
                true
            }
            else -> super.onOptionsItemSelected(item)
        }
    }
}
```

Barre d’action et menus

Menus en cascade

Les menus en cascade, ou sous-menus, permettent de créer des niveaux supplémentaires dans un menu, pour afficher des options supplémentaires ou des fonctionnalités plus avancées.

Pour créer des menus en cascade dans une application Android en Kotlin, vous pouvez utiliser l'élément <menu> imbriqué à l'intérieur d'un autre élément <item> dans votre fichier de menu XML. Voici un exemple de fichier XML pour un menu en cascade :

Barre d'action et menus

Menus en cascade

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
<item
   android:id="@+id/menu_item_1"
   android:title="Menu Item 1"
   android:icon="@drawable/ic_menu_item_1"
   android:showAsAction="ifRoom">
   <menu>
      <item
         android:id="@+id/submenu_item_1"
         android:title="Submenu Item 1"
         android:icon="@drawable/ic_submenu_item_1" />
      <item
         android:id="@+id/submenu_item_2"
         android:title="Submenu Item 2"
         android:icon="@drawable/ic_submenu_item_2" />
   </menu>
</item>
<item
   android:id="@+id/menu_item_2"
   android:title="Menu Item 2"
   android:icon="@drawable/ic_menu_item_2"
   android:showAsAction="ifRoom" />
</menu>
```

Barre d’action et menus

Menus contextuels

- Les menus contextuels (ou "menus contextuels") sont des menus qui apparaissent dans une interface graphique lorsqu'un utilisateur effectue un clic droit sur un élément. Ces menus fournissent une liste d'options contextuelles en fonction de l'élément sur lequel l'utilisateur a cliqué.

Barre d’action et menus

Menus contextuels

Pour créer un menu contextuel en Kotlin, vous pouvez utiliser la classe ContextMenu fournie par le framework Android. Voici les étapes à suivre pour créer un menu contextuel dans une application Android en Kotlin :

1. Créez une vue ou un widget pour lequel vous souhaitez créer un menu contextuel.
2. Ajoutez un listener de clic droit à la vue ou au widget à l'aide de la méthode setOnCreateContextMenuListener().
3. Dans le listener de clic droit, créez une instance de la classe ContextMenu et ajoutez des éléments de menu à l'aide de la méthode add().
4. Définissez un listener pour chaque élément de menu en utilisant la méthode setOnMenuItemClickListener().
5. Affichez le menu contextuel à l'aide de la méthode show()

Barre d’action et menus

Menus contextuels

```
val myView = findViewById<View>(R.id.my_view)

myView.setOnCreateContextMenuListener { menu, v, menuInfo ->
    menu.add("Option 1")
    menu.add("Option 2")
    menu.findItem(R.id.option1)?.setOnMenuItemClickListener {
        // Code à exécuter lorsque l'option 1 est sélectionnée
        true
    }
    menu.findItem(R.id.option2)?.setOnMenuItemClickListener {
        // Code à exécuter lorsque l'option 2 est sélectionnée
        true
    }
}

myView.setOnLongClickListener {
    it.showContextMenu()
    true
}
```

Barre d’action et menus

Menus contextuels

- Dans cet exemple, myView est la vue à laquelle le menu contextuel est associé.
- Le listener setOnCreateContextMenuListener() crée le menu contextuel et ajoute deux éléments de menu.
- Les listeners setOnMenuItemClickListener() définissent le code à exécuter lorsque chaque élément de menu est sélectionné.
- Enfin, le listener setOnLongClickListener() affiche le menu contextuel lorsque l'utilisateur effectue un clic long sur la vue.

Annonces et dialogues

Les annonces

Pour afficher une annonce (message d'information) à l'utilisateur dans une application Android en utilisant la classe Toast de Kotlin.

Un « toast » est un message apparaissant en bas d'écran pendant un instant, par exemple pour confirmer la réalisation d'une action. Un toast n'affiche aucun bouton et n'est pas actif.

Voici les étapes à suivre :

1. Importez la classe Toast dans votre fichier de code :

```
import android.widget.Toast
```

Annonces et dialogues

Les annonces

2. Utilisez la méthode makeText() de la classe Toast pour créer une nouvelle instance de Toast en spécifiant le contexte de l'application, le texte de l'annonce et la durée pendant laquelle l'annonce doit être affichée (valeur de la constante LENGTH_SHORT pour une courte durée et LENGTH_LONG pour une durée plus longue) puis affichez l'annonce en appelant la méthode show() de l'instance Toast : :

```
val message = "Ceci est une annonce"
val duration = Toast.LENGTH_SHORT
val toast = Toast.makeText(context, message, duration)
toast.show()
```

Annonces et dialogues

Les dialogues

- Les dialogues sont des fenêtres pop-up qui s'affichent au-dessus de l'interface utilisateur principale d'une application Android. Ils peuvent être utilisés pour afficher des messages d'erreur, demander une confirmation de l'utilisateur ou pour fournir des informations supplémentaires à l'utilisateur.

- Il existe plusieurs types de dialogues dans une application Android :
  1. Dialogue d'alerte
  2. Dialogue de sélection
  3. Dialogue de progression
  4. Dialogues personnalisés

Annonces et dialogues

Les dialogues

- **Dialogues d'alerte** (AlertDialog) : ces dialogues sont utilisés pour afficher des messages d'erreur, des avertissements ou des messages d'information à l'utilisateur. Ils ont un titre, un message et un ou plusieurs boutons pour l'utilisateur.

```
val builder = AlertDialog.Builder(this)
builder.setTitle("Confirmer l'action")
.setMessage("Voulez-vous vraiment effectuer cette action ?")
.setPositiveButton("OK") { dialog, which ->
    // Code à exécuter lorsque l'utilisateur appuie sur le bouton OK
}
.setNegativeButton("Annuler") { dialog, which ->
    // Code à exécuter lorsque l'utilisateur appuie sur le bouton Annuler
}
val dialog = builder.create()
dialog.show()
```

Annonces et dialogues

Les dialogues

- **Dialogues de sélection (Dialog)** : ces dialogues sont utilisés pour permettre à l'utilisateur de faire un choix parmi une liste d'options. Ils ont généralement un titre, une liste de choix et des boutons pour l'utilisateur.

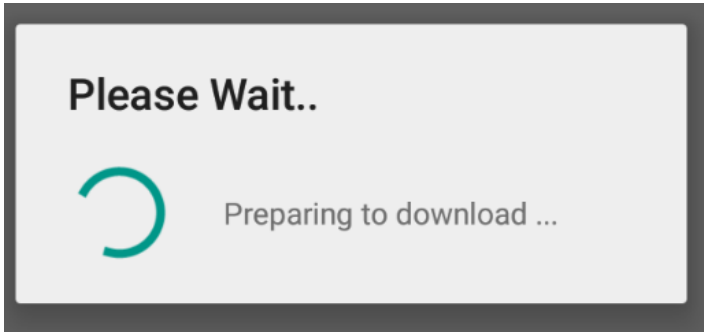
```
val items = arrayOf("Option 1", "Option 2", "Option 3")
val builder = AlertDialog.Builder(this)
builder.setTitle("Sélectionnez une option")
.setItems(items) { dialog, which ->
    // Code à exécuter lorsque l'utilisateur sélectionne une
option
}
val dialog = builder.create()
dialog.show()
```

Annonces et dialogues

Les dialogues

- **Dialogues de progression (ProgressDialog)** : ces dialogues sont utilisés pour afficher la progression d'une tâche ou d'une opération en cours. Ils ont généralement un message qui décrit la tâche en cours et une barre de progression pour indiquer la progression.

```
val progressDialog = ProgressDialog(this)
progressDialog.setTitle("Please Wait")
progressDialog.setMessage(« Preparing to download...")
progressDialog.setCancelable(false)
progressDialog.show()
```



Annonces et dialogues

Les dialogues

- **Dialogues personnalisés (DialogFragment)** : ces dialogues sont utilisés pour créer des dialogues personnalisés avec un contenu et une apparence personnalisés. Ils peuvent contenir des champs de texte, des boutons, des images ou tout autre élément d'interface utilisateur.

```
class CustomDialogFragment : DialogFragment() {
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Charger le fichier de mise en page personnalisé
        val view = inflater.inflate(R.layout.dialog_layout, container, false)

        // Définir les listeners de bouton ou d'autres éléments d'interface utilisateur

        return view
    }
}
```

Annonces et dialogues

Les dialogues

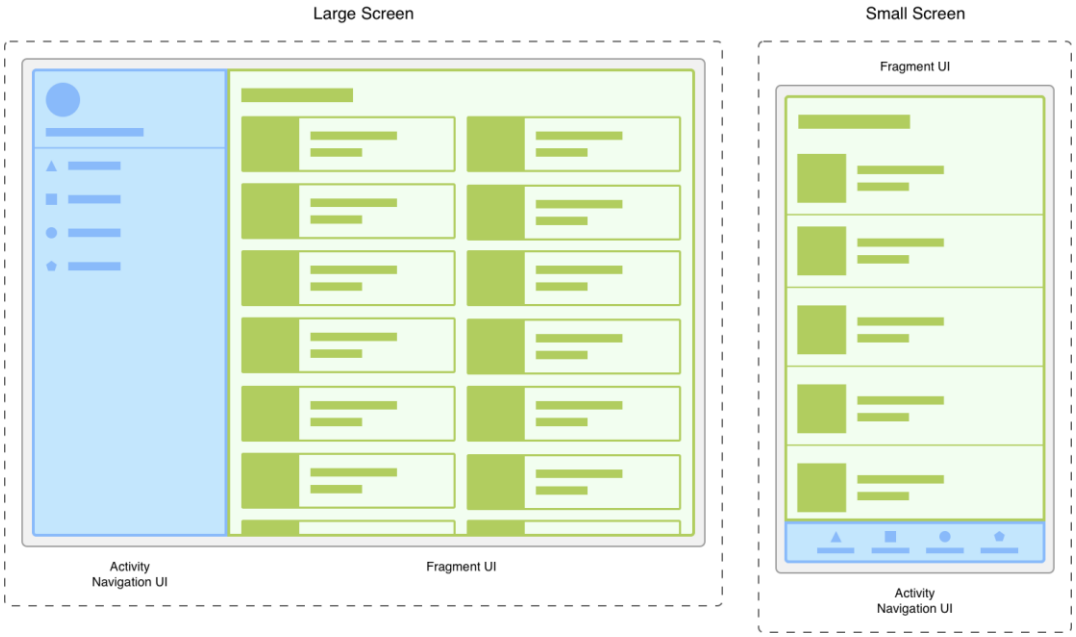
Remarque:

Pour fermer un dialogue en Kotlin, vous pouvez appeler la méthode **dismiss()** sur l'objet Dialog ou DialogFragment

Les fragments

Présentation

- Les fragments en Kotlin sont des éléments d'interface utilisateur réutilisables qui peuvent être combinés pour créer des interfaces utilisateur plus complexes.
- Les fragments sont similaires aux **activités**, mais contrairement aux activités, ils ne peuvent pas être exécutés indépendamment.
- Les fragments doivent être intégrés à une activité hôte pour être affichés.



Les fragments

Création de fragments en kotlin

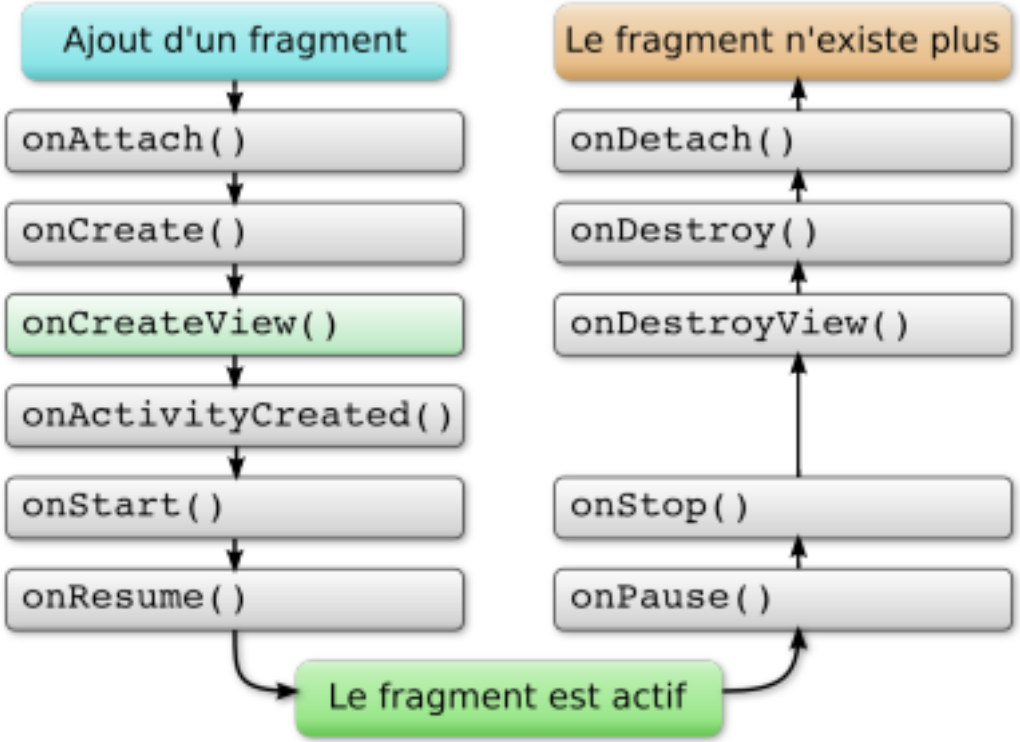
- Pour créer un nouveau fragment en Kotlin, vous pouvez créer une nouvelle classe qui étend la classe Fragment. Par exemple :

```
class MyFragment : Fragment() {
    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        // Inflater le layout du fragment
Val view = inflater.inflate(R.layout.my_fragment, container, false)
        return view
    }
}
```

Les fragments

Cycle de vie des fragments

- Les fragments ont un cycle de vie similaire à celui des activités, avec quelques méthodes de plus correspondant à leur intégration dans une activité.



Les fragments

Intégration de fragment à une activité

Pour ajouter un fragment à une activité hôte, vous devez ajouter un **FrameLayout** à votre layout d'activité pour contenir le fragment, puis utiliser FragmentManager pour ajouter le fragment à ce conteneur. Par exemple :

```
<FrameLayout
android:id="@+id/fragment_container"
android:layout_width="match_parent"
android:layout_height="match_parent"/>
```

```
class MainActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        // Ajouter le fragment
        supportFragmentManager.beginTransaction()
            .add(R.id.fragment_container, MyFragment())
            .commit()
    }
}
```

Les fragments

Communication entre Activité et Fragments

La communication entre une activité et un fragment peut se faire de différentes manières en Kotlin.

1. Utilisation de l'interface :

Vous pouvez créer une interface dans votre fragment et implémenter cette interface dans votre activité. Ensuite, vous pouvez appeler la méthode de l'interface depuis votre fragment pour communiquer avec votre activité.

Voici un exemple de code:

Les fragments

Communication entre Activité et Fragments

Dans le fragment

```
interface MyListener {
    fun onButtonClick()
}

class MyFragment : Fragment() {
    private var listener: MyListener? = null

    override fun onAttach(context: Context) {
        super.onAttach(context)
        listener = context as? MyListener
    }

    override fun onDetach() {
        super.onDetach()
        listener = null
    }

    fun onButtonClicked() {
        listener?.onButtonClick()
    }
}
```

Les fragments

Communication entre Activité et Fragments

Dans l'activité

```
class MyActivity : AppCompatActivity(), MyFragment.MyListener {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val fragment = MyFragment()
        supportFragmentManager.beginTransaction()
            .replace(R.id.fragment_container, fragment)
            .commit()
    }

    override fun onButtonClicked() {
        // Faites quelque chose ici en réponse au clic sur le bouton dans le fragment
    }
}
```

Les fragments

Communication entre Activité et Fragments

On peut aussi utiliser:

- **Bundle** : Vous pouvez ajouter des données à Bundle dans votre activité, puis récupérer ces données dans votre fragment

- **ViewModel** pour partager des données entre votre activité et votre fragment. ViewModel est un composant de l'architecture Android qui permet de stocker et de gérer les données de manière à survivre aux changements de configuration (par exemple, lorsque l'écran est tourné)

Préférences d'application

Présentation

- Les préférences d'application sont importantes car elles permettent aux utilisateurs de personnaliser l'expérience utilisateur et de se sentir plus à l'aise en utilisant une application. Les préférences peuvent inclure des paramètres tels que la langue, la taille du texte, la couleur de fond, les notifications, etc. Les utilisateurs ont souvent des préférences personnelles en matière de conception et de fonctionnalité, et en permettant de personnaliser les préférences, les applications peuvent répondre aux besoins et aux préférences individuels des utilisateurs.
- Les développeurs peuvent également utiliser les préférences d'application pour stocker les données de configuration, telles que les paramètres de connexion ou les informations d'utilisateur, ce qui peut améliorer l'expérience utilisateur en évitant à l'utilisateur de saisir les mêmes informations plusieurs fois.

Préférences d'application

création de préférences

En Kotlin pour Android, vous pouvez définir les préférences d'application en utilisant la classe **SharedPreferences**.
Voici un exemple de code pour créer et accéder aux préférences d'application :
1. Créer un fichier de préférences xml: **res/xml/prefs.xml** :

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
<PreferenceCategory android:title="Paramètres">
  <CheckBoxPreference
    android:key="notif_pref"
    android:title="Activer les notifications"
    android:defaultValue="true"/>
  <ListPreference
    android:key="lang_pref"
    android:title="Langue"
    android:entries="@array/langues"
    android:entryValues="@array/langue_valeurs"
    android:defaultValue="fr"/>
</PreferenceCategory>
</PreferenceScreen>
```

Préférences d'application

création et accès aux préférences

En Kotlin pour Android, vous pouvez définir les préférences d'application en utilisant la classe **SharedPreferences**.
Voici un exemple de code pour créer et accéder aux préférences d'application :
1. Créer un fichier de préférences xml: **res/xml/prefs.xml** :

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android">
<PreferenceCategory android:title="Paramètres">
  <CheckBoxPreference
    android:key="notif_pref"
    android:title="Activer les notifications"
    android:defaultValue="true"/>
  <ListPreference
    android:key="lang_pref"
    android:title="Langue"
    android:entries="@array/langues"
    android:entryValues="@array/langue_valeurs"
    android:defaultValue="fr"/>
</PreferenceCategory>
</PreferenceScreen>
```

Préférences d'application

création et accès aux préférences

2. Dans votre activité, vous pouvez accéder aux préférences d'application en créant une instance de **SharedPreferences** :

```
val prefs = PreferenceManager.getDefaultSharedPreferences(this)
```

3. Vous pouvez maintenant accéder aux valeurs des préférences en utilisant les clés que vous avez définies dans le fichier prefs.xml :

```
val notificationsActivées = prefs.getBoolean("notif_pref", true)
val langue = prefs.getString("lang_pref", "fr")
```

Remarque: Vous pouvez également écouter les changements de préférences en ajoutant un listener :

Préférences d'application

création et accès aux préférences

```
prefs.registerOnSharedPreferenceChangeListener { sharedPreferences, key ->
    when (key) {
        "notif_pref" -> {
            val notificationsActivées = sharedPreferences.getBoolean(key, true)
            // Faire quelque chose avec la nouvelle valeur des notifications
        }
        "lang_pref" -> {
            val langue = sharedPreferences.getString(key, "fr")
            // Faire quelque chose avec la nouvelle valeur de la langue
        }
    }
}
```

Cela vous permettra de détecter les changements de préférences en temps réel et d'adapter votre application en conséquence.

Préférences d'application

Stockage des entiers, chaines et array

SharedPreferences est une interface de stockage de données légère et facile à utiliser dans Android, qui vous permet de stocker des données sous forme de clé-valeur. Vous pouvez stocker des entiers, des chaînes et des tableaux dans SharedPreferences en Kotlin de la manière suivante :

Pour stocker un entier :

```
val sharedPreferences = getSharedPreferences("MyPreferences", Context.MODE_PRIVATE)
val editor = sharedPreferences.edit()
editor.putInt("key", 123)
editor.apply()
```

Dans cet exemple, nous utilisons la méthode putInt() pour stocker l'entier 123 avec la clé "key" dans SharedPreferences. Nous devons également utiliser la méthode apply() pour enregistrer les changements.

Préférences d'application

Stockage des entiers, chaines et array

Pour stocker une chaine:

```
val sharedPreferences = getSharedPreferences("MyPreferences",
Context.MODE_PRIVATE)
val editor = sharedPreferences.edit()
editor.putString("key", "valeur")
editor.apply()
```

Ici, nous utilisons la méthode putString() pour stocker la chaîne "valeur" avec la clé "key".

Préférences d'application

Stockage des entiers, chaines et array

Pour stocker un tableau:

```
val sharedPreferences = getSharedPreferences("MyPreferences",
Context.MODE_PRIVATE)
val editor = sharedPreferences.edit()
val tableau = arrayOf("valeur1", "valeur2", "valeur3")
editor.putStringSet("key", tableau.toSet())
editor.apply()
```

Nous utilisons la méthode putStringSet() pour stocker un ensemble de chaînes avec la clé "key".
Pour convertir un tableau en ensemble, nous utilisons la méthode toSet().

Préférences d'application

Récupérer des entiers, chaines et array

```
val nom_chaine = sharedPreferences.getString()("clé", "valeur par defaut")
val nom_int = sharedPreferences.getInt()("clé", 0)
val nom_array = sharedPreferences.getStringSet() ("clé",
emptySet()?.toTypedArray)
```

Il est temps de faire une pause

C’est fini pour cette séance, rendez-vous la séance prochaine pour un cours sur la programmation declarative avec Jetpack Compose

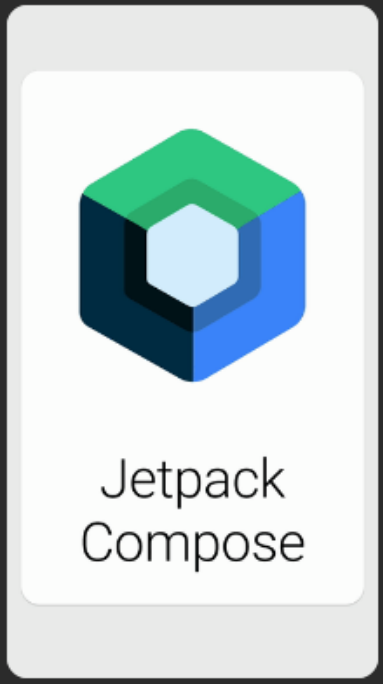
PROGRAMMATION DECLARATIVE

AVEC JETPACK COMPOSE

SÉANCE 7-12

JETPACK COMPOSE

```
@Composable
fun JetpackCompose() {
    Card {
        var expanded by remember { mutableStateOf(false) }
        Column(Modifier.clickable { expanded = !expanded }) {
            Image(painterResource(R.drawable.jetpack_compose))
            AnimatedVisibility(expanded) {
                Text(
                    text = "Jetpack Compose",
                    style = MaterialTheme.typography.bodyLarge,
                )
            }
        }
    }
}
```



Le cours de cette semaine porte sur Jetpack compose.

- Introduction à Jetpack Compose
- Installation, configuration et concepts de base de Jetpack Compose
- Layouts , contraintes et widgets
- Thèmes et styles
- Navigation et interactions

Introduction à Jetpack Compose

Présentation

- **Jetpack Compose** est une bibliothèque d'interface utilisateur (UI) Android moderne et déclarative, développée par Google. Elle permet de créer des interfaces utilisateur dynamiques et personnalisées pour les applications Android, en utilisant le langage de programmation Kotlin.
- **Jetpack Compose** utilise une approche déclarative pour créer des interfaces utilisateur. Contrairement aux méthodes traditionnelles basées sur le code XML, où vous devez définir chaque élément de l'interface utilisateur séparément, Jetpack Compose vous permet de définir l'interface utilisateur dans une seule fonction, en utilisant des widgets réutilisables et des fonctions pour modifier leur apparence.

Introduction à Jetpack Compose

Présentation

- Le code Jetpack Compose est plus simple, plus concis et plus facile à comprendre que les méthodes traditionnelles basées sur XML. Il est également plus facile de créer des interfaces utilisateur dynamiques et personnalisées en utilisant Jetpack Compose.

- Kotlin et Jetpack Compose sont des technologies très prometteuses pour le développement Android, et les développeurs Android devraient les apprendre pour rester compétitifs et productifs.

Introduction à Jetpack Compose

Présentation

Code allégé

Faites-en plus avec moins de code et évitez des classes entières de bugs : le code est simple et facile à gérer.

Convivialité

Il vous suffit de décrire votre interface utilisateur, et Compose se charge du reste. Lorsque l'état de l'application change, l'interface utilisateur est automatiquement mise à jour.

Introduction à Jetpack Compose

Présentation

Développement accéléré

Compatible avec l'ensemble de votre code existant pour que vous puissiez l'adopter quand et où vous le souhaitez. Itérez rapidement des aperçus en direct et bénéficiez de la compatibilité totale avec Android Studio.

Performances

Créez de superbes applications avec un accès direct aux API de la plate-forme Android et une compatibilité intégrée avec Material Design, le thème sombre, les animations et plus encore.

Introduction à Jetpack Compose

Présentation

Compiler sur plusieurs appareils avec Compose

Jetpack Compose vous permet de créer des applications de haute qualité sur tous les appareils, que ce soit sur un téléphone, une tablette, un pliable, Chrome OS ou Wear OS.

Compose pour les grands écrans

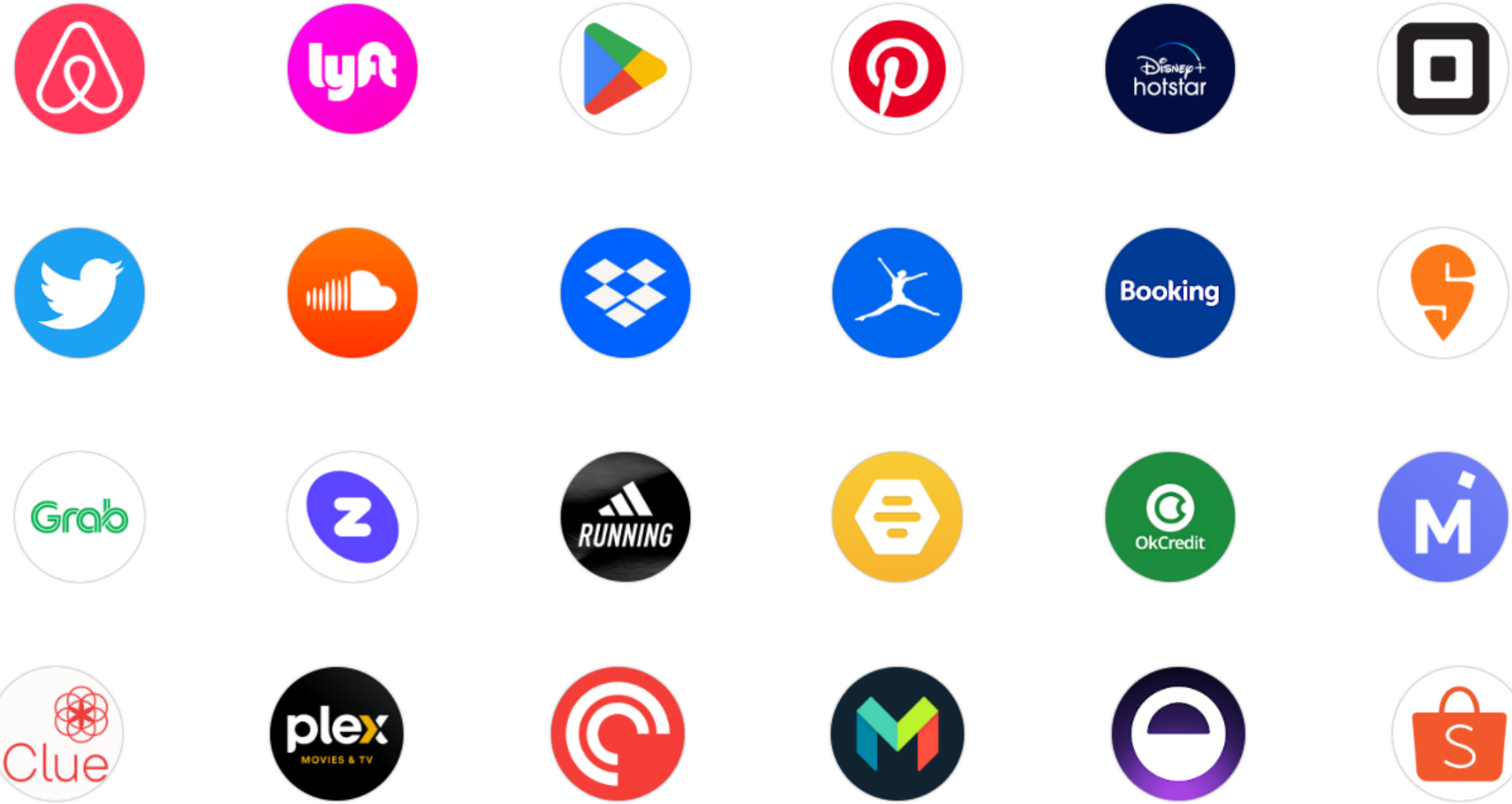
L'UI de votre application doit s'adapter aux différentes tailles d'écran, différentes orientations et différents facteurs de forme. Une mise en page adaptative change en fonction de l'espace d'écran disponible.

Compose pour Wear OS

Compose pour Wear OS permet de créer des applications portables sur le poignet plus facilement, plus rapidement et de manière plus intuitive. Ce guide vous présente les similitudes et les différences entre la version standard de Compose et Compose pour Wear OS.

Introduction à Jetpack Compose

Applications créées avec Compose



Introduction à Jetpack Compose

Installation et configuration

Pour ajouter Jetpack Compose à un projet existant, suivez les étapes suivantes :

1. Assurez-vous que vous utilisez Android Studio 4.0 ou une version ultérieure.
2. Ouvrez le fichier build.gradle de votre projet.
3. Ajoutez les dépendances suivantes à la section dependencies du fichier build.gradle :

```
dependencies {
    // Jetpack Compose
    implementation 'androidx.compose.ui:ui:1.1.0'
    implementation 'androidx.compose.material:material:1.1.0'
    implementation 'androidx.compose.runtime:runtime-livedata:1.1.0'
}
```

Introduction à Jetpack Compose

Installation et configuration

4. Ajoutez les plugins suivants à la section plugins du fichier build.gradle :

```
plugins {
    id 'org.jetbrains.kotlin.android' version '1.5.31'
    id 'kotlin-parcelize'
    id 'com.android.application'
    id 'com.google.gms.google-services'
    id 'kotlin-android'
    id 'kotlin-android-extensions'
}
```

Introduction à Jetpack Compose

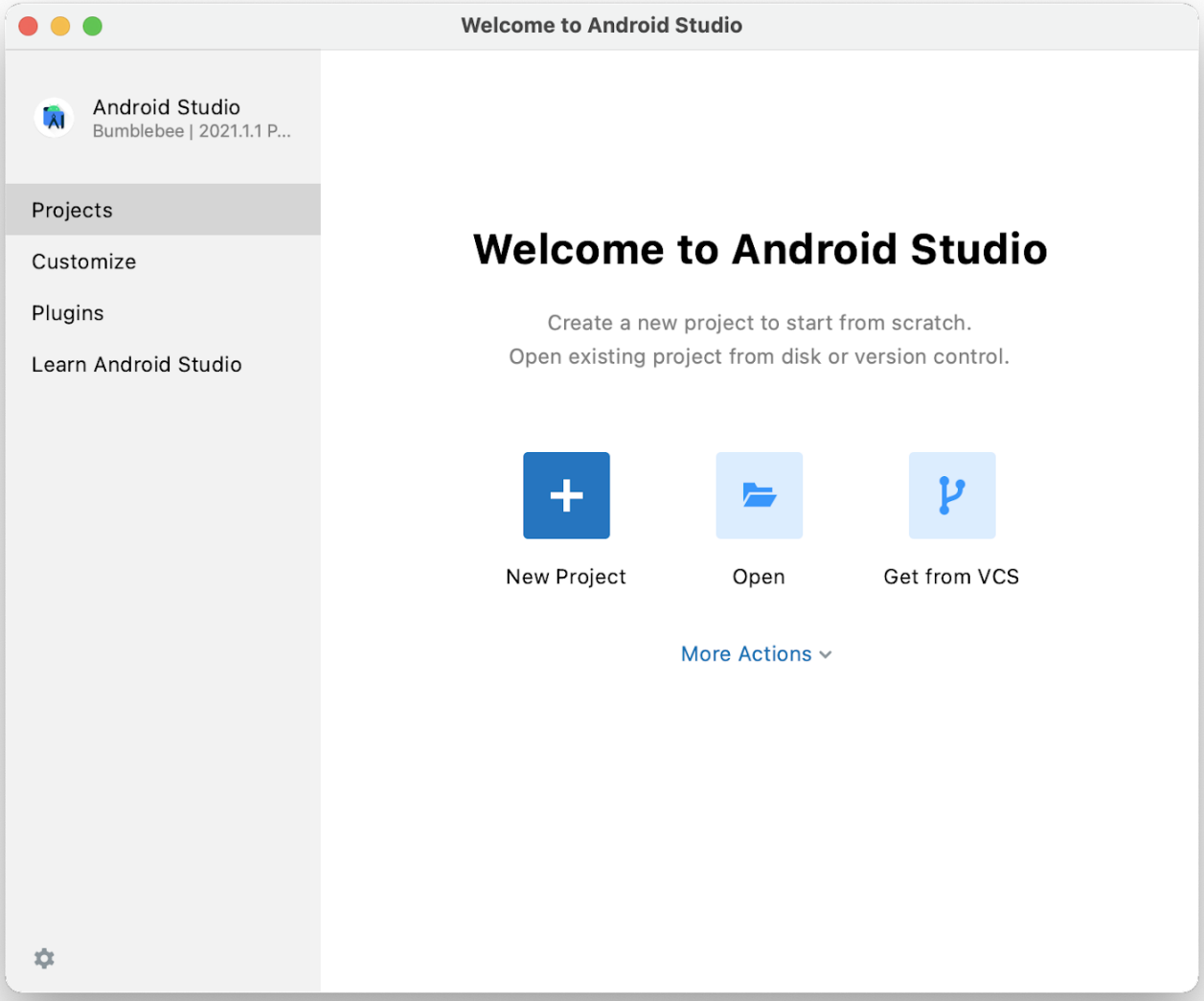
Installation et configuration

5. Synchronisez le fichier build.gradle en cliquant sur Sync Now.
Après avoir ajouté Jetpack Compose à votre projet, vous pouvez commencer à utiliser les widgets et les fonctions de Jetpack Compose pour créer des interfaces utilisateur modernes et déclaratives.

Remarque: N'oubliez pas que Jetpack Compose est une technologie relativement nouvelle et qu'il est important de vérifier régulièrement les mises à jour pour vous assurer que vous utilisez la dernière version et que vous bénéficiez des dernières fonctionnalités et améliorations.

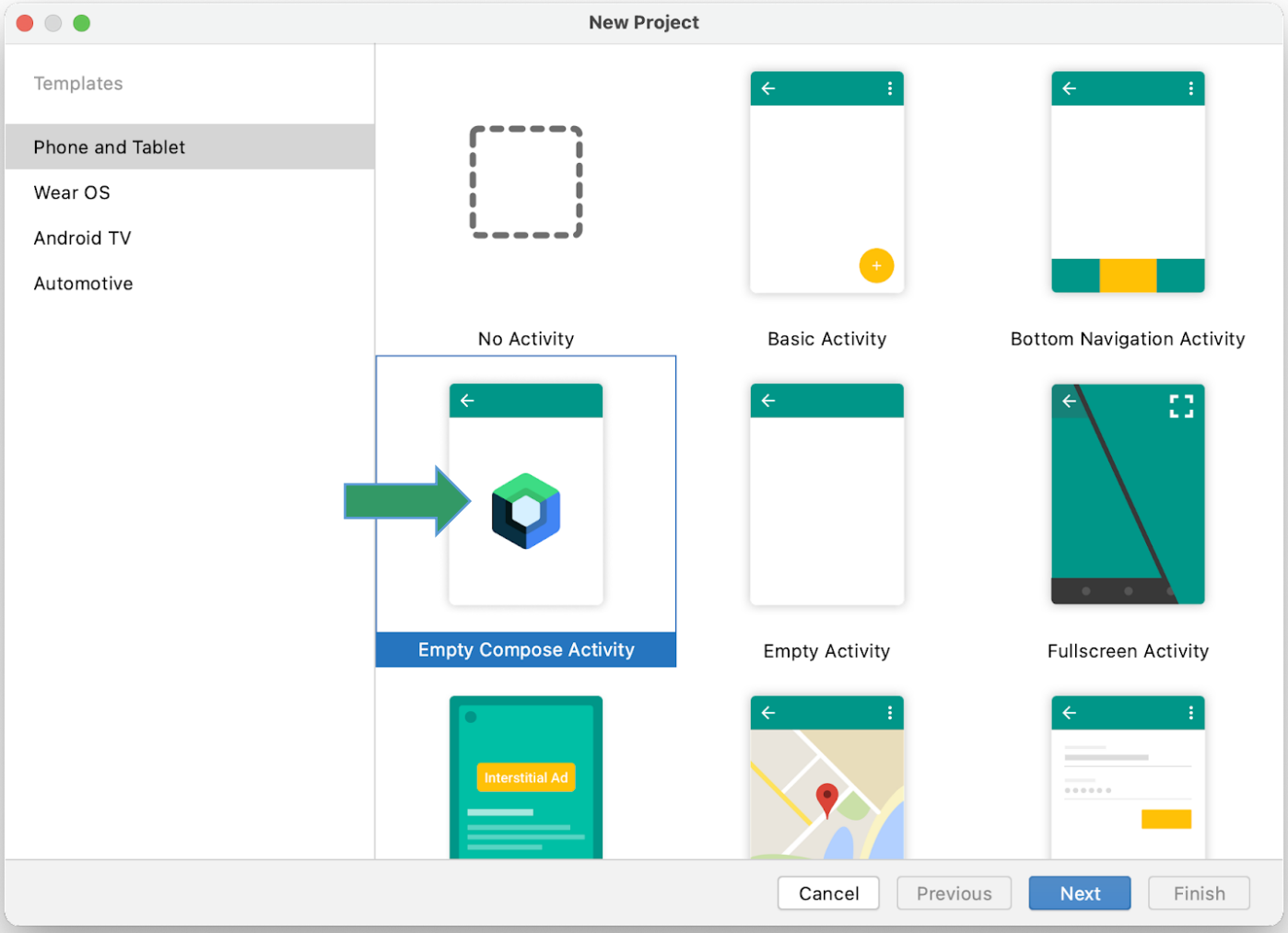
Introduction à Jetpack Compose

Création d'une application Android avec compose



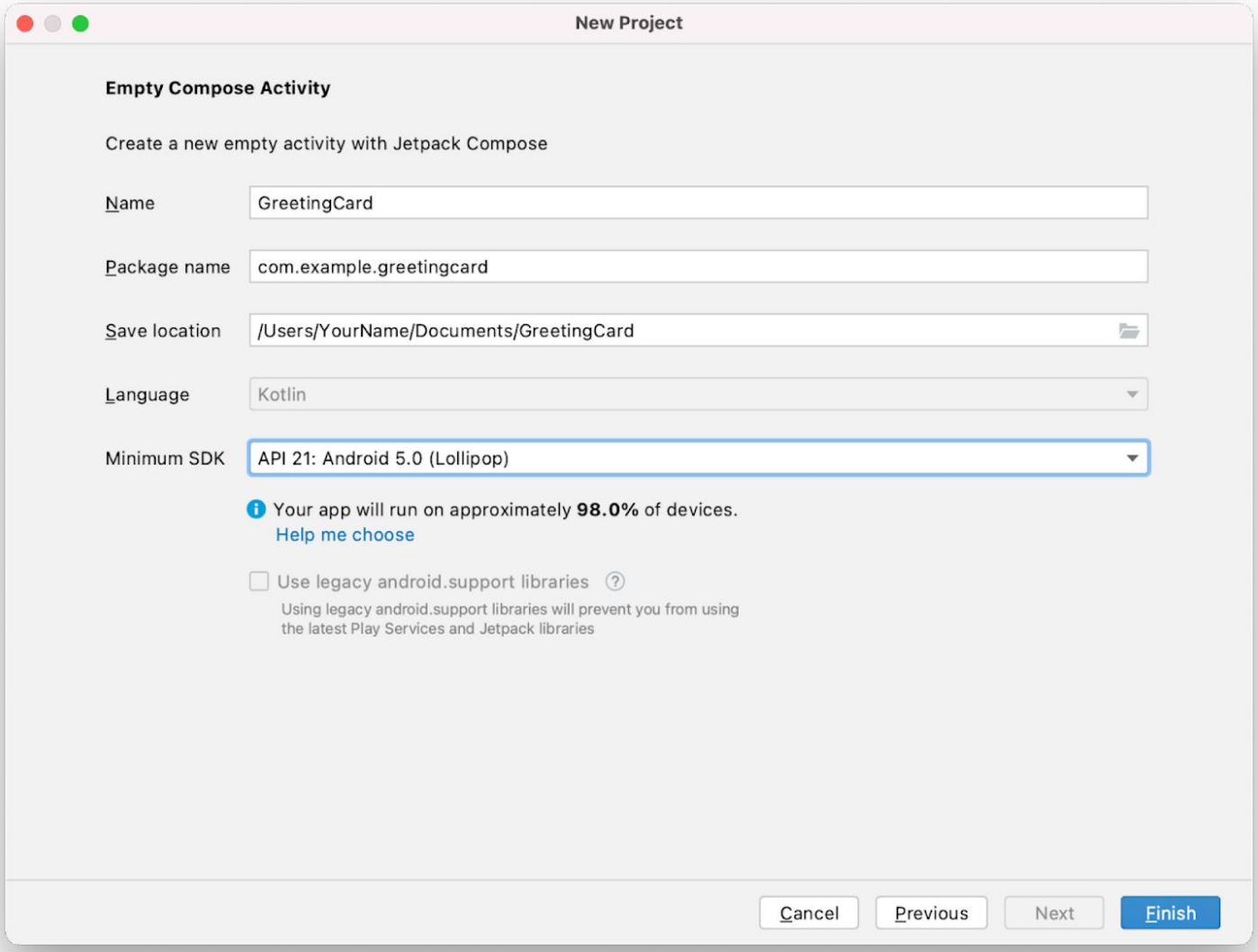
Introduction à Jetpack Compose

Création d'une application Android avec compose



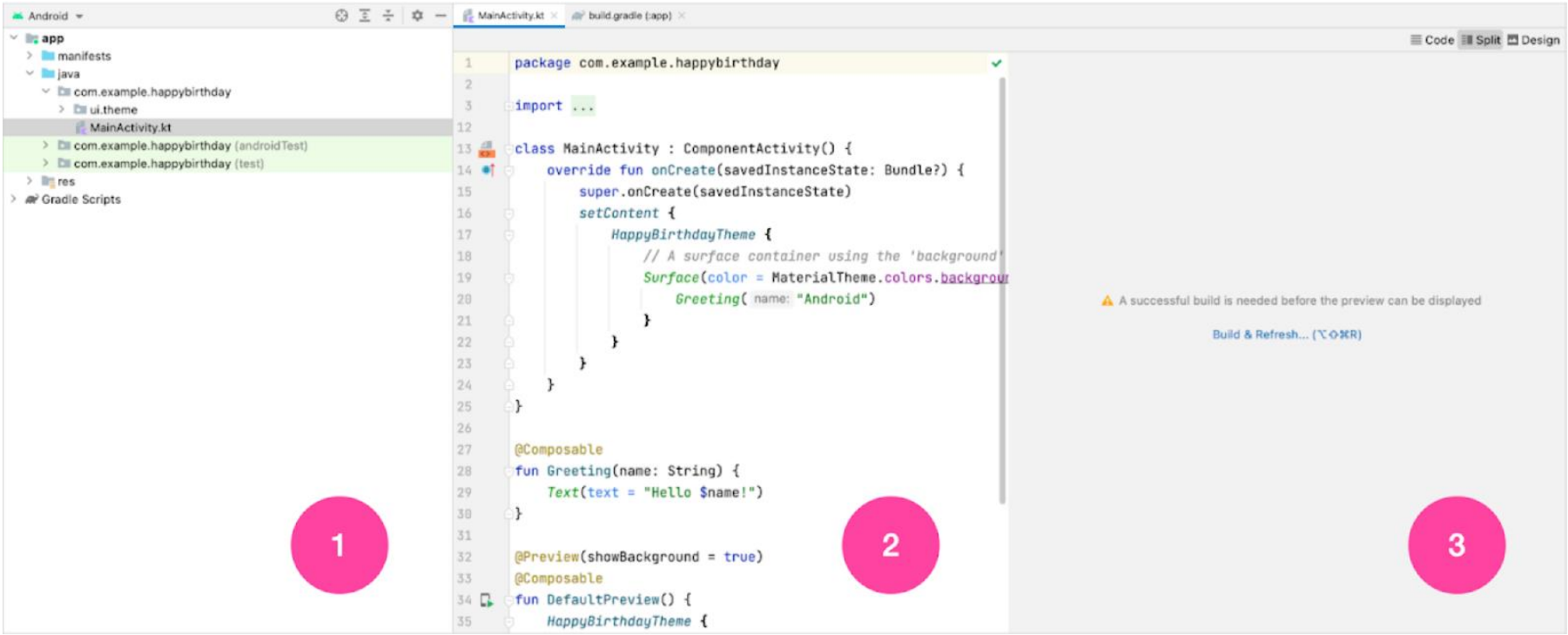
Introduction à Jetpack Compose

Création d'une application Android avec compose



Introduction à Jetpack Compose

Création d'une application Android avec compose



- La vue **Projet** (1) affiche les fichiers et les dossiers de votre projet.
- La vue **Code** (2) est la zone dans laquelle vous pouvez modifier le code.
- La vue **Conception** (3) vous permet d'afficher un aperçu de votre application.

Introduction à Jetpack Compose

Présentation de la structure du code

Le code généré par défaut contient des fonctions générées automatiquement, et en particulier **onCreate**() et **setContent**().

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            GreetingCardTheme {
                // A surface container using the 'background' color from the theme
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colors.background
                ) {
                    Greeting("Android")
                }
            }
        }
    }
}
```

Introduction à Jetpack Compose

Présentation de la structure du code

- La fonction **onCreate**() est le point d'entrée de cette application et elle appelle d'autres fonctions pour créer l'interface utilisateur. Dans les programmes Kotlin, la fonction main() est le point précis où démarre le compilateur Kotlin dans votre code. Dans les applications Android, c'est la fonction onCreate() qui remplit ce rôle.

- La fonction **setContent**() à l'intérieur de la fonction onCreate() permet de définir votre mise en page au moyen de fonctions modulables. Toutes les fonctions annotées avec **@Composable** peuvent être appelées à partir de la fonction setContent() ou d'autres fonctions modulables. L'annotation indique au compilateur Kotlin que cette fonction est utilisée par Jetpack Compose pour générer l'interface utilisateur.

Introduction à Jetpack Compose

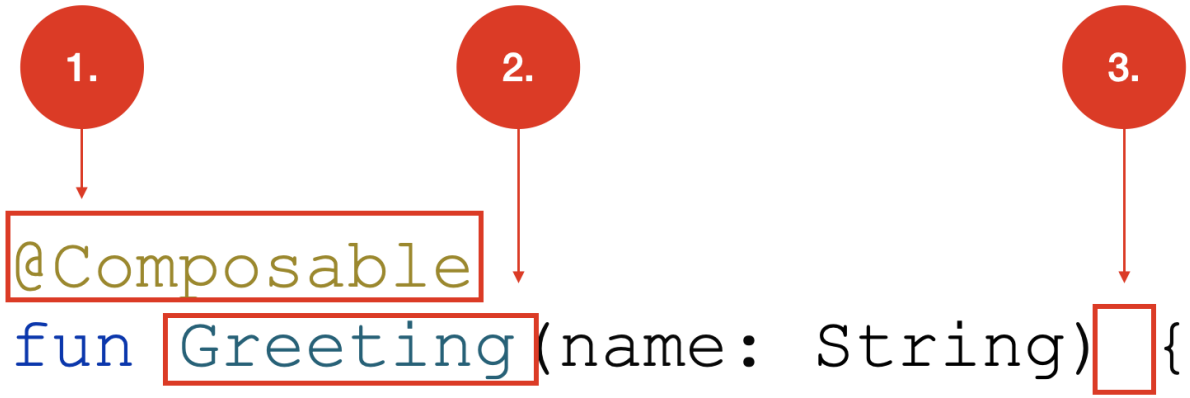
Présentation de la structure du code

- Examinez ensuite la fonction **Greeting**(). Elle est une fonction modulable ; remarquez l'annotation @Composable indiquée au-dessus. Une fonction modulable reçoit des entrées et génère ce qui s'affiche à l'écran.

```
@Composable
fun Greeting(name: String) {
    Text(text = "Hello $name!")
}
```

Introduction à Jetpack Compose

Présentation de la structure du code



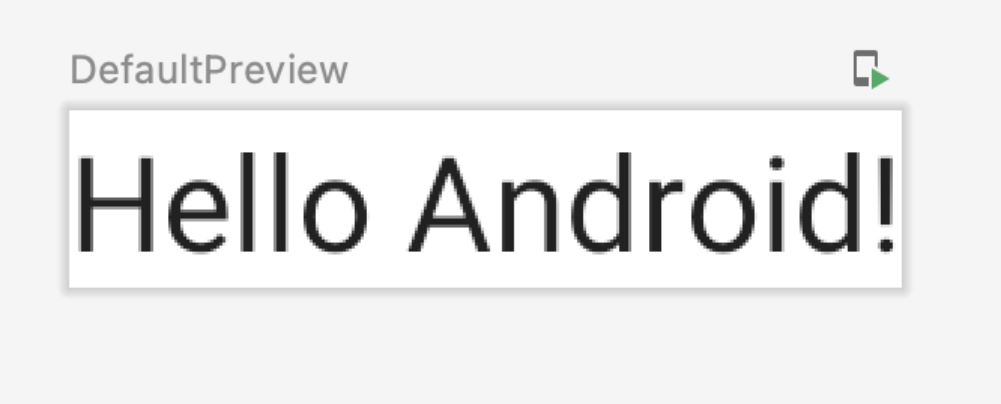
IMPORTANT :

- La **première** lettre d'un nom de fonction @Composable est mise en **majuscule**.
- L'annotation @Composable est ajoutée avant la fonction.
- Les fonctions @Composable ne peuvent rien renvoyer.

Introduction à Jetpack Compose

Présentation de la structure du code

- La fonction **DefaultPreview()** affiche un aperçu de la fonction composable.
- En cliquant sur **Compiler et actualiser**. La compilation peut prendre un certain temps. Une fois qu'elle est terminée, l'aperçu affiche Hello Android! dans la zone de texte.



- L'activité Compose vide contient tout le code nécessaire pour créer cette application.

Introduction à Jetpack Compose

Présentation de la structure du code

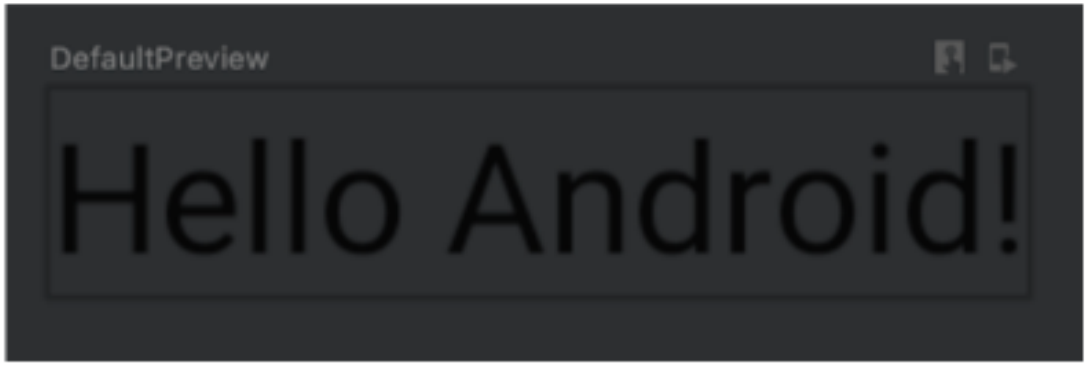
- **DefaultPreview()** est une fonctionnalité particulièrement utile qui vous permet de voir à quoi ressemble votre application sans avoir à la compiler entièrement. Pour qu'il s'agisse d'une fonction d'aperçu, vous devez ajouter une annotation **@Preview**.
- Comme vous pouvez le constater, l'annotation @Preview accepte un paramètre appelé showBackground. Si showBackground est défini sur true, un arrière-plan est ajouté à l'aperçu de votre application.
- Par défaut, Android Studio utilise un thème clair pour l'éditeur. Il peut donc être difficile de voir la différence entre showBackground = true et showBackground = false. Vous pouvez toutefois voir la différence ci-dessous avec un thème sombre. Notez que l'arrière-plan blanc de l'image est défini sur "true".

Introduction à Jetpack Compose

Présentation de la structure du code

showBackground = true

showBackground = false



Etats de base de Jetpack Compose

Présentation des états de base

Jetpack Compose utilise des concepts de base pour créer des interfaces utilisateur déclaratives. Voici quelques-uns des concepts clés :

1. **État (State)** : L'état est une variable qui peut être modifiée et qui détermine l'apparence et le comportement d'un composant. Lorsque l'état est modifié, le composant est redessiné automatiquement pour refléter le nouveau état.

2.Fonctions (Functions) : Les fonctions sont utilisées pour décrire l'apparence et le comportement des composants d'interface utilisateur. Elles prennent en entrée des paramètres et renvoient un composant à afficher.

Etats de base de Jetpack Compose

Présentation des états de base

3. **Compositions (Composables)** : Les compositions sont des fonctions qui utilisent d'autres fonctions pour décrire une partie de l'interface utilisateur. Les compositions peuvent être imbriquées pour créer des hiérarchies d'interface utilisateur plus complexes.
4. Widgets : Les widgets sont les composants de base utilisés pour créer l'interface utilisateur. Ils peuvent être des boutons, des champs de texte, des images, des listes, des tableaux, etc.
5. Modifier (Modifier) : Les modificateurs sont des fonctions qui modifient l'apparence et le comportement des widgets. Ils peuvent être utilisés pour définir des couleurs, des tailles, des polices, des événements, etc.
En utilisant ces concepts de base, Jetpack Compose permet de créer des interfaces utilisateur modernes et déclaratives, qui sont plus faciles à comprendre et à modifier que les méthodes traditionnelles basées sur le code XML.

Les Widgets de Jetpack Compose

Notion de mise en page

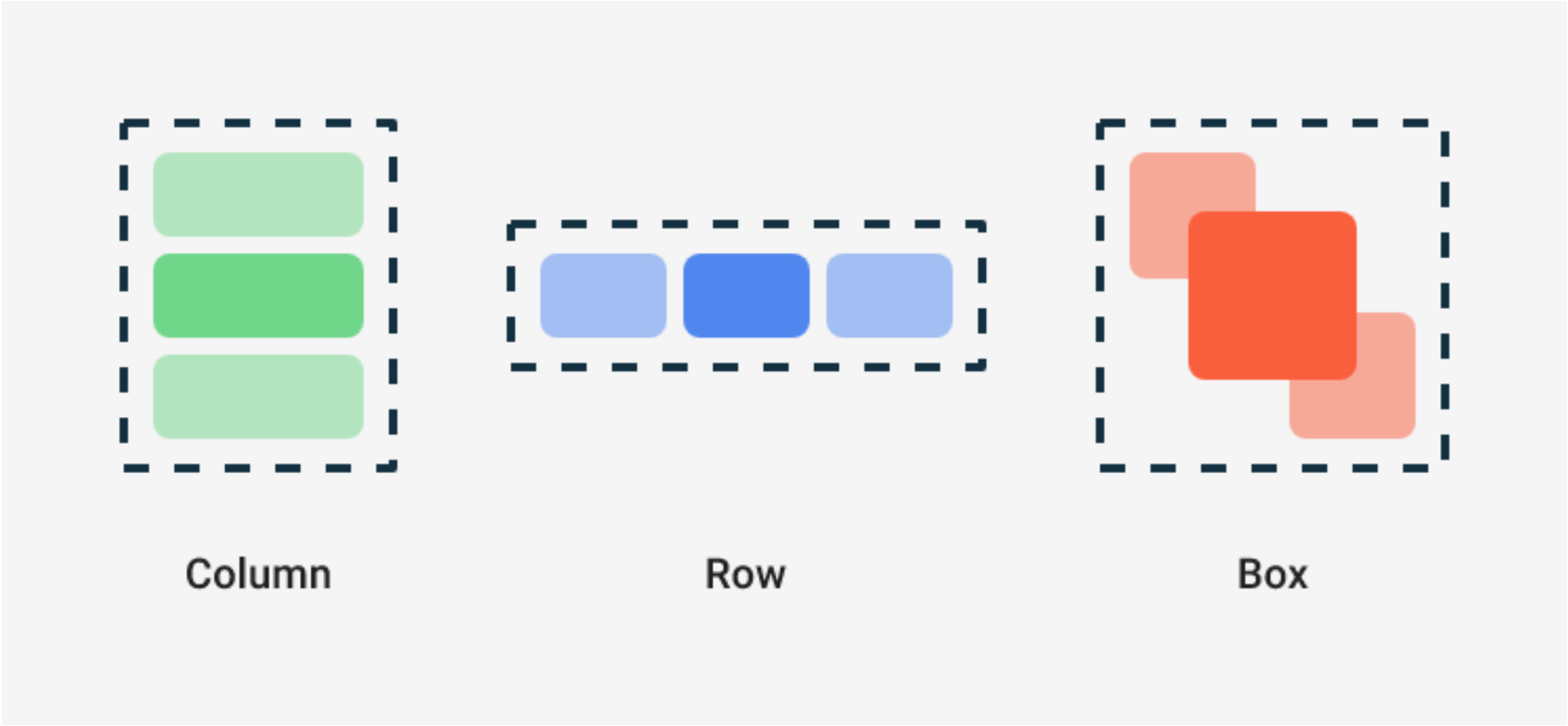
Les éléments de l'interface utilisateur sont organisés de façon hiérarchique, avec des éléments contenus dans d'autres éléments. Dans Compose, vous créez une hiérarchie d'interface utilisateur en appelant des fonctions modulables à partir d'autres fonctions modulables.

Dans Compose, les trois éléments de mise en page standards de base sont :

1. **Column** :Cet élément de mise en page organise les widgets verticalement les uns au-dessus des autres dans une colonne.
2. **Row** : Cet élément de mise en page organise les widgets horizontalement côte à côte dans une ligne.
3. **Box** :Cet élément de mise en page organise les widgets les uns au-dessus des autres en utilisant des coordonnées x et y

Les Widgets de Jetpack Compose

Notion de mise en page



Les Widgets de Jetpack Compose

Notion de mise en page

```
Column(
 //modificateur
modifier=Modifier.
) {
   // les Widgets
}

Row(
//modificateur
) {
    // les Widgets
}

Box(
//modificateur
) {
    // les Widgets
}
```

Les Widgets de Jetpack Compose

Notion de mise en page

Les modificateurs sont utilisés pour ajouter des fonctionnalités ou des comportements supplémentaires aux widgets et aux éléments de mise en page. Les modificateurs sont chaînés à l'aide de l'opérateur `.` , ce qui permet de les combiner pour créer des mises en page plus complexes. Voici quelques exemples de modificateurs couramment utilisés en Jetpack Compose :

- **background**: pour définir la couleur de fond d'un widget.
- **border**: pour ajouter une bordure à un widget.
- **padding**: pour ajouter un espace autour d'un widget.
- **clickable**: pour rendre un widget cliquable.
- **fillMaxSize**: pour remplir toute la taille disponible dans le conteneur parent.
- **align**: pour aligner un widget à l'intérieur de son conteneur parent.
- **weight**: pour définir la répartition de l'espace disponible entre plusieurs widgets dans un conteneur.

Les Widgets de Jetpack Compose

Notion de mise en page

```
Row(
modifier = Modifier.fillMaxWidth(),
horizontalArrangement = Arrangement.SpaceEvenly
) {

}

Column(
        modifier = Modifier
            .fillMaxSize()
            .padding(16.dp)
            .verticalScroll(rememberScrollState())
    ) {

}
```

Les Widgets de Jetpack Compose

Présentation de Widgets

- Un **widget** est un composant d'interface utilisateur d'une application.

- Dans le contexte de Jetpack Compose, un **widget** est un composant d'interface utilisateur de base qui peut être utilisé pour créer des interfaces utilisateur personnalisées.

- Les widgets de Jetpack Compose sont des éléments de construction qui peuvent être combinés pour créer des interfaces utilisateur complexes.

Les Widgets de Jetpack Compose

Présentation de Widgets

Text: utilisé pour afficher du texte. Il prend en charge diverses propriétés de texte telles que la couleur, la taille, le style, etc.

```
Text (
    text = "Je suis un texte",
    style = TextStyle(
        fontSize = 24.sp,
        fontWeight = FontWeight.Normal,
        color = Color.Blue
    )
)
```

Les Widgets de Jetpack Compose

Présentation de Widgets

Image: utilisé pour afficher une image. Il prend en charge diverses propriétés d'image telles que la forme, le redimensionnement, etc.

```
Image(
    painter = painterResource(R.mipmap.ic_launcher),
    contentDescription = "My Image",
    modifier = Modifier.size(100.dp)
)
```

Les Widgets de Jetpack Compose

Présentation de Widgets

Button : utilisé pour créer un bouton cliquable. Il prend en charge diverses propriétés de bouton telles que la couleur, la forme, la taille, etc.

```
Button(
    onClick = { /* Do something */ },
    modifier = Modifier
        .padding(16.dp)
        .fillMaxWidth(),
    colors = ButtonDefaults.buttonColors(
        backgroundColor = MaterialTheme.colors.primary,
        contentColor = MaterialTheme.colors.onPrimary
    )
) {
    Text(text = "Button")
}
```

Les Widgets de Jetpack Compose

Présentation de Widgets

TextField: utilisé pour collecter du texte entré par l'utilisateur. Il prend en charge diverses propriétés de champ de texte telles que la couleur, la forme, la taille, etc

```
TextField(
    value = "",
    onValueChange = { /* Action à effectuer lors de la saisie de texte */ },
    label = {
            Text("Entrez votre texte ici")
    },
    modifier = Modifier.fillMaxWidth()
)
```

Les Widgets de Jetpack Compose

Présentation de Widgets

Icon: est un widget de base de Jetpack Compose qui permet d'afficher une icône à l'écran.

```
Icon(
    Icons.Default.Favorite,
    contentDescription = "Favorite Icon",
    tint = Color.Red,
    modifier = Modifier.size(32.dp)
)
```

Les Widgets de Jetpack Compose

Présentation de Widgets

IconButton: est un widget de base de Jetpack Compose qui permet d'afficher une icône avec un bouton cliquable.

```
IconButton(
    onClick = { /* Action à effectuer lors du clic sur l'icône */ },
    modifier = Modifier.size(64.dp)
) {
    Icon(
        Icons.Filled.Search,
        contentDescription = "Apple Icon",
        tint = Color.Red,
        modifier = Modifier.size(32.dp)
    )
}
```

Les Widgets de Jetpack Compose

Présentation de Widgets

Chekbox: est un widget de base de Jetpack Compose qui permet aux utilisateurs de sélectionner ou de désélectionner une option. Il est souvent utilisé pour les choix binaires, tels que les cases à cocher "Oui" ou "Non" ou les options "Activer" ou "Désactiver. Il aussi utilisé pour collecter une valeur booléenne (vrai/faux) à partir de l'utilisateur. Il prend en charge diverses propriétés de case à cocher telles que la couleur, la forme, la taille, etc.

```
var isChecked by remember { mutableStateOf(false) }

Checkbox(
    checked = isChecked,
    onCheckedChange = { isChecked = it },
    colors = CheckboxDefaults.colors(
        checkedColor = Color.Red,
        uncheckedColor = Color.Gray
    ),
    modifier = Modifier.padding(8.dp)
)
```

Les Widgets de Jetpack Compose

Présentation de Widgets

RadioButton: utilisé pour collecter une valeur unique à partir d'une liste d'options. Il prend en charge diverses propriétés de bouton radio telles que la couleur, la forme, la taille, etc.

```
val options = listOf("Option 1", "Option 2", "Option 3")
var selectedOption by remember { mutableStateOf(options[0]) }

Column {
    options.forEach { option ->
        Row(
            modifier = Modifier
                .fillMaxWidth()
                .padding(vertical = 8.dp),
            verticalAlignment = Alignment.CenterVertically
        ) {
            RadioButton(
                selected = selectedOption == option,
                onClick = { selectedOption = option },
                modifier = Modifier.padding(horizontal = 8.dp)
            )
            Text(
                text = option,
                style = MaterialTheme.typography.body1,
                modifier = Modifier.padding(start = 16.dp)
            )
        }
    }
}
```

Les Widgets de Jetpack Compose

Présentation de Widgets

Switch : utilisé pour collecter une valeur booléenne (vrai/faux) en basculant entre deux états (activé et désactivé). Il prend en charge diverses propriétés de commutateur telles que la couleur, la forme, la taille, etc.

```
var switchState by remember { mutableStateOf(false) }
Column(
    modifier = Modifier.padding(16.dp)
) {
    Text(
        text = "Activer la fonctionnalité",
        style = MaterialTheme.typography.subtitle1
    )
    Switch(
        checked = switchState,
        onCheckedChange = { switchState = it },
        modifier = Modifier.padding(top = 8.dp),
        colors = SwitchDefaults.colors(
            checkedThumbColor = MaterialTheme.colors.primary,
            checkedTrackColor = MaterialTheme.colors.primary.copy(alpha = 0.5f)
        )
    )
}
```

Les Widgets de Jetpack Compose

Présentation de Widgets

ProgressBar : utilisé pour afficher la progression d'une tâche. Il prend en charge diverses propriétés de barre de progression telles que la couleur, la forme, la taille, etc.

```
var progress by remember { mutableStateOf(0.5f) }

Column {
    LinearProgressIndicator(
        progress = progress,
        modifier = Modifier.fillMaxWidth(),
        color = MaterialTheme.colors.primary
    )
    Spacer(modifier = Modifier.height(16.dp))
    Button(onClick = {
        progress += 0.1f
        if (progress >= 1f) progress = 0f
    }) {
        Text(text = "Increase progress")
    }
}
```

Les Widgets de Jetpack Compose

Présentation de Widgets

Spacer : est un widget de base de Jetpack Compose qui est utilisé pour ajouter de l'espace entre les éléments d'une interface utilisateur. La taille de l'espace peut être définie à l'aide de modificateurs

```
Spacer(modifier = Modifier.height(16.dp))
```

Les Widgets de Jetpack Compose

Présentation de Widgets

Card : est un widget de Jetpack Compose qui vous permet d'afficher du contenu dans une carte avec une ombre douce et un fond arrondi. Il est souvent utilisé pour afficher des informations sous forme de cartes.

```
Card(
    modifier = Modifier
        .fillMaxWidth()
        .padding(16.dp),
    elevation = 4.dp
) {
    Column(
        modifier = Modifier.padding(16.dp)
    ) {
        Text(
            text = "Card Title",
            style = MaterialTheme.typography.h6
        )
        Spacer(modifier = Modifier.height(8.dp))
        Text(
            text = "Card Content",
            style = MaterialTheme.typography.body1
        )
    }
}
```

Les Widgets de Jetpack Compose

Présentation de Widgets

LazyColumn : est un widget de Jetpack Compose qui permet d'afficher une liste verticale de manière efficace. Contrairement à Column, LazyColumn ne crée que les éléments qui sont actuellement visibles à l'écran, ce qui permet de réduire l'utilisation de la mémoire et d'optimiser les performances de l'application.

```
val names = listOf("Alice", "Bob", "Charlie", "David", "Emily", "Frank", "Grace", "Henry", "Ivy", "John")
val scrollState = rememberLazyListState()

LazyColumn(
    modifier = Modifier
        .fillMaxSize()
        .padding(16.dp),
    verticalArrangement = Arrangement.spacedBy(8.dp),
    horizontalAlignment = Alignment.CenterHorizontally,
    contentPadding = PaddingValues(vertical = 16.dp),
    reverseLayout = true,
    state = scrollState,
) {
    items(names) { name ->
        Text(
            text = name,
            modifier = Modifier.fillMaxWidth(),
            textAlign = TextAlign.Center,
            style = MaterialTheme.typography.h6
        )
    }
}
```

Les Widgets de Jetpack Compose

Présentation de Widgets

LazyRow : est un widget de Jetpack Compose qui permet d'afficher une horizontale de manière efficace. Contrairement à Row, LazyRow ne crée que les éléments qui sont actuellement visibles à l'écran, ce qui permet de réduire l'utilisation de la mémoire et d'optimiser les performances de l'application.

```
LazyRow(
    modifier = Modifier.fillMaxWidth(),
    horizontalArrangement = Arrangement.spacedBy(16.dp)
) {
    items(10) { index ->
        Box(
            modifier = Modifier
                .width(100.dp)
                .height(100.dp)
                .background(Color.Blue),
            contentAlignment = Alignment.Center
        ) {
            Text(text = "Item $index", color = Color.White)
        }
    }
}
```

Les Widgets de Jetpack Compose

Scaffold

- **Scaffold** est un widget de mise en page pratique qui fournit une disposition de base pour les écrans courants tels que les écrans d'accueil, les écrans de détails, les formulaires, etc.
- Le widget Scaffold est construit sur le modèle de conception "top-down" qui permet aux développeurs de se concentrer sur la création des éléments de contenu plutôt que sur la création des éléments de l'interface utilisateur tels que les barres d'outils, les tiroirs de navigation, les boutons d'action flottants, etc.
- Les éléments de Scaffold sont: topBar, bottomBar, floatingActionButton et content (contenu de la page)

Les Widgets de Jetpack Compose

Scaffold

```
Scaffold(
        topBar = {
        //bar d'outils
        },
        bottomBar = {
        //bar de navigation
        },
        floatingActionButton = {
        //bouton flotante
        },
        content = {
         //contenu de la page
        }
)
```

Les Widgets de Jetpack Compose

topBar

Les éléments possibles de **topBar** sont:

- **title** : Le titre de la barre d'applications.
- **subtitle** : Le sous-titre de la barre d'applications.
- **navigationIcon** : L'icône de navigation à gauche de la barre d'applications. On peut utiliser un IconButton avec une icône de menu.
- **actions** : Les actions à droite de la barre d'applications. On peut utiliser les IconButton avec des icônes de recherche, de favoris et de partage. Etc..
- **elevation** : L'élévation de la barre d'applications, qui ajoute une ombre pour donner une impression de profondeur.
- **backgroundColor** : La couleur de fond de la barre d'applications.
- **contentColor** : La couleur de texte des éléments de la barre d'applications.
- **navigationContentColor** : La couleur de texte de l'icône de navigation

Les Widgets de Jetpack Compose

topBar

```
topBar={TopAppBar(
title = { Text("My App") },
subtitle = { Text("Subtitle") },
navigationIcon = {
    IconButton(onClick = { /* TODO */ }) {
        Icon(Icons.Default.Menu)
    }
},
actions = {
    IconButton(onClick = { /* TODO */ }) {
        Icon(Icons.Default.Search)
    }
    IconButton(onClick = { /* TODO */ }) {
        Icon(Icons.Default.Favorite)
    }
    IconButton(onClick = { /* TODO */ }) {
        Icon(Icons.Default.Share)
    }
},
elevation = 8.dp,
backgroundColor = Color.Blue,
contentColor = Color.White,
navigationContentColor = Color.White,
modifier = Modifier.height(56.dp)
)
}
```

Les Widgets de Jetpack Compose

bottomBar

BottomNavigation peut contenir les éléments suivants :

- **backgroundColor** : La couleur de fond de la barre de navigation inférieure.
- **contentColor** : La couleur de texte et d'icône des éléments de la barre de navigation inférieure.
- **elevation** : L'élévation de la barre de navigation inférieure, qui ajoute une ombre pour donner une impression de profondeur.
- **modifier** : Définit la hauteur de la barre de navigation inférieure.

Les Widgets de Jetpack Compose

bottomBar

```
BottomNavigation(
    backgroundColor = Color.White, // set the background color to blue
    elevation = 8.dp // set the elevation to 8dp
) {
    BottomNavigationItem(
        selected = true,
        onClick = { /* TODO */ },
        icon = { Icon(Icons.Filled.Home, contentDescription = "Home") },
        label = { Text("Home") },
        selectedContentColor = Color.Blue,
        unselectedContentColor = Color.Gray
    )
    BottomNavigationItem(
        selected = false,
        onClick = { /* TODO */ },
        icon = { Icon(Icons.Filled.Search, contentDescription = "Search") },
        label = { Text("Search") },
        selectedContentColor = Color.Blue,
        unselectedContentColor = Color.Gray
    )
    BottomNavigationItem(
        selected = false,
        onClick = { /* TODO */ },
        icon = { Icon(Icons.Filled.Settings, contentDescription = "Settings") },
        label = { Text("Settings") },
        selectedContentColor = Color.Blue,
        unselectedContentColor = Color.Gray
    )

}
```

Les Widgets de Jetpack Compose

FloatingButton

- **onClick** : La fonction à exécuter lorsqu'on clique sur le bouton.
- **backgroundColor** : La couleur de fond du bouton.
- **contentColor** : La couleur de l'icône et du texte sur le bouton.
- **content** : Le contenu du bouton, qui peut être un texte, une icône ou tout autre élément composé.
- **elevation** : L'élévation du bouton, qui ajoute une ombre pour donner une impression de profondeur.
- **modifier** : Les modificateurs qui définissent la position et la taille du bouton.

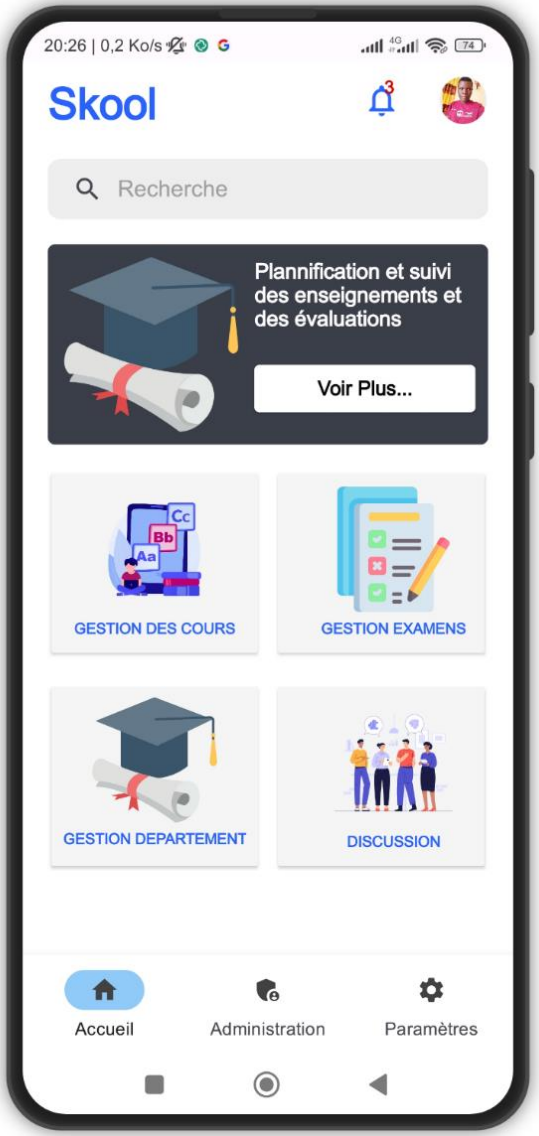
Les Widgets de Jetpack Compose

floatingActionButton

```
FloatingActionButton(
onClick = { /* action à exécuter lors du clic */ },
backgroundColor = Color.Red,
contentColor = Color.White,
content = {
    Icon(Icons.Filled.Add, contentDescription = "Ajouter")
},
elevation = FloatingActionButtonDefaults.elevation(8.dp),
modifier = Modifier.padding(16.dp)
)
```

Exercice

Reproduire avec Jetpack compose les interfaces ci-dessous



Thèmes & styles

- Les thèmes et les styles sont des concepts clés dans le développement Android, car ils permettent de personnaliser l'apparence de l'interface utilisateur.
- Le **thème** d'une application est un ensemble de styles prédéfinis pour différents éléments d'interface utilisateur, tels que les boutons, les textes, les arrière-plans, etc. Les thèmes sont souvent utilisés pour donner une apparence cohérente à l'ensemble de l'application.
- Les **styles** sont des ensembles de propriétés qui définissent l'apparence d'un élément spécifique d'interface utilisateur, tel qu'un bouton ou un texte. Les styles peuvent être définis de manière globale pour l'ensemble de l'application ou de manière spécifique pour un élément d'interface utilisateur individuel.

Thèmes & styles

- Jetpack Compose utilise également des thèmes et des styles pour personnaliser l'apparence de l'interface utilisateur. Cependant, la manière de définir les thèmes et les styles est différente de celle des anciennes versions d'Android, car Jetpack Compose utilise une approche plus déclarative.
- Pour définir un thème dans Jetpack Compose, vous pouvez utiliser la fonction **MaterialTheme**, qui définit un ensemble de styles de matériaux. Vous pouvez également définir vos propres styles personnalisés en utilisant la fonction androidx.compose.material.Typography pour les styles de texte, androidx.compose.material.Shapes pour les formes, et androidx.compose.ui.graphics.Color pour les couleurs.

Thèmes & styles

Voici un exemple de définition d'un thème dans Jetpack Compose :

```
val myTheme = darkColors(
    primary = Color(0xFFBB86FC),
    background = Color(0xFF121212)
)

@Composable
fun MyApp() {
    MaterialTheme(colors = myTheme) {
        // l'interface utilisateur de l'application
    }
}
```

Thèmes & styles

- Pour définir un style personnalisé dans Jetpack Compose, vous pouvez utiliser la fonction **style** et la propriété **Modifier** pour appliquer le style à un élément d'interface utilisateur spécifique.

Voici un exemple de définition d'un style pour un bouton :

Thèmes & styles

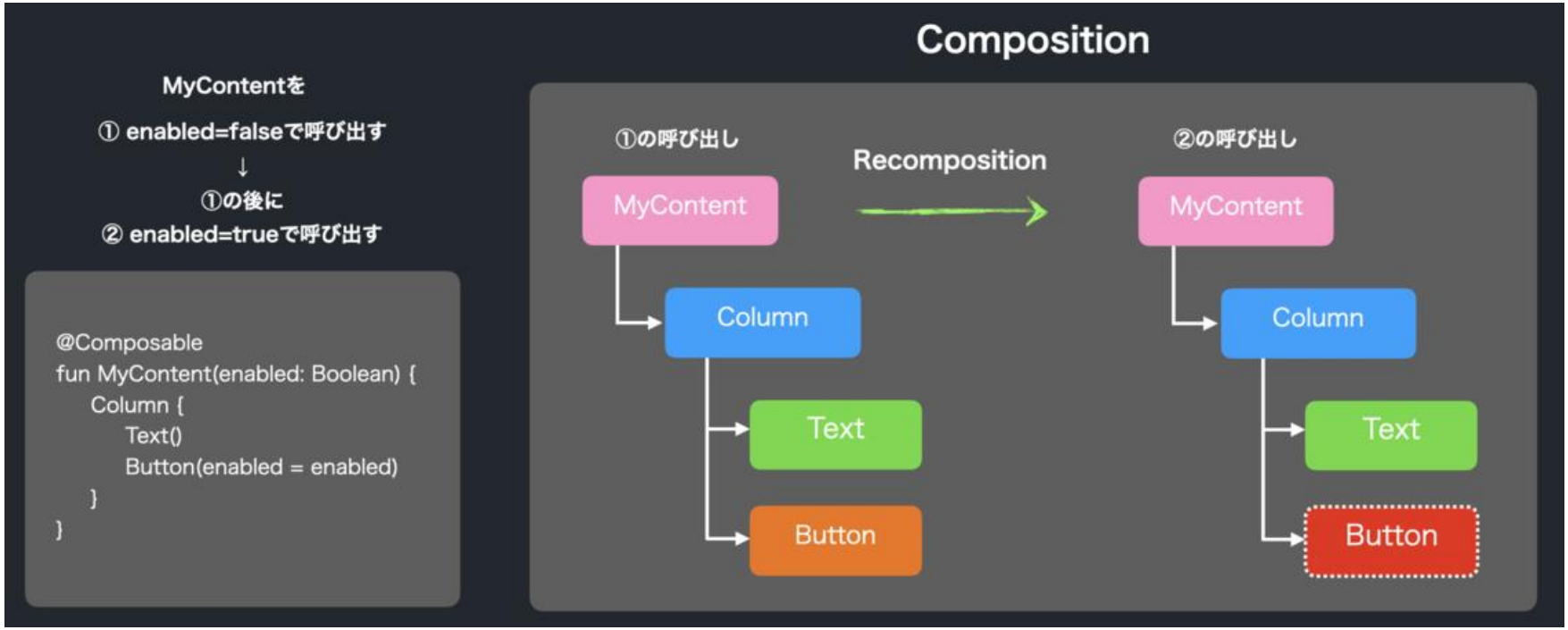
```
val TextStyle = ButtonDefaults.textButtonColors(
    backgroundColor = Color(0xFFBB86FC),
    contentColor = Color.White,
)

Button(
    onClick = { /* handle button click */ },
    modifier = Modifier
        .padding(16.dp)
        .height(48.dp)
        .width(200.dp),
    colors = TextStyle
) {
    Text("Cliquez ici")
}

}
```

Composition & recomposition

- La composition et la recomposition sont deux concepts clés de Jetpack Compose.
- Jetpack Compose utilise une approche déclarative pour la construction de l'interface utilisateur, ce qui signifie que l'état de l'interface utilisateur est décrit dans une fonction qui crée une représentation visuelle de l'interface utilisateur en fonction de cet état.
- Cette fonction est appelée chaque fois que l'état change et la représentation visuelle de l'interface utilisateur est recomposée pour refléter les nouveaux états.
- Le processus de composition et de recomposition est géré par le framework lui-même, il n'y a donc pas besoin pour les développeurs de gérer manuellement ce processus

Composition & recomposition

Composition & recomposition



La gestion des états

remember

La gestion de l'état est une partie importante de la création d'interface utilisateur avec Jetpack Compose. Jetpack Compose fournit plusieurs outils pour gérer l'état de l'application, notamment :

1. **mutableStateOf** : c'est une fonction qui permet de créer un état mutable, c'est-à-dire une variable qui peut être modifiée et qui déclenche automatiquement une recomposition de l'interface utilisateur lorsque sa valeur est mise à jour. Voici un exemple :

```
var counter by remember { mutableStateOf(0) }

Button(onClick = { counter++ }) {
    Text("Cliquez moi $counter fois")
}
```

La gestion des états

remember

2. remember : c'est une fonction qui permet de créer une variable dont la valeur est mémorisée (ou "rappelée"), c'est-à-dire qu'elle n'est pas recréée à chaque fois que la composition est recomposée. Voici un exemple

```
val list = remember { mutableListOf<String>() }

list.add("Nouvel élément")
```

Dans cet exemple, nous avons créé une liste list qui est mémorisée à l'aide de la fonction remember. Nous avons ensuite ajouté un nouvel élément à la liste. Chaque fois que la composition est recomposée, list est rappelée avec les mêmes éléments qu'auparavant, ce qui garantit la stabilité de l'interface utilisateur.

La gestion des états

viewModel

3. viewModel : c'est une classe qui permet de stocker et de gérer l'état de l'application de manière centralisée. Elle est généralement utilisée pour stocker les données qui doivent être partagées entre plusieurs composants de l'interface utilisateur. Voici un exemple :

```
class MyViewModel : ViewModel() {
    val counter = mutableStateOf(0)
}

@Composable
fun MyScreen(viewModel: MyViewModel = viewModel()) {
    Button(onClick = { viewModel.counter.value++ }) {
        Text("Cliquez moi ${viewModel.counter.value} fois")
    }
}
```

Navigation avec Jetpack compose

Composant de navigation

- **Navigation** est une bibliothèque Jetpack qui permet de naviguer d'une destination à une autre dans votre application.

Pour utiliser cette bibliothèque de navigation Jetpack compose, on il tout d'abord ajouter la dépendance en procédant comme suit:
ouvrez le fichier de compilation de l'application, qui se trouve ici : app/build.gradle. Dans la section des dépendances, ajoutez la dépendance navigation-compose.

```
implementation 'androidx.navigation:navigation-compose:2.5.3'
```

Navigation avec Jetpack compose

Composant de navigation

Voici les composants de navigation les plus couramment utilisés en Jetpack Compose:

- **NavHost** - C'est le conteneur qui héberge les destinations de navigation. Il est utilisé pour définir la zone dans laquelle la navigation doit être effectuée.
- **NavController** - C'est un objet qui gère la navigation entre les destinations. Il permet de naviguer vers une destination en fonction de l'état actuel de l'application.
- **NavGraph** - C'est une représentation visuelle de la structure de navigation de l'application. Il définit les différentes destinations et les chemins de navigation qui les relient.
- **NavDestination** - C'est un élément de la hiérarchie de navigation qui représente une destination à laquelle l'utilisateur peut naviguer

Navigation avec Jetpack compose

Navigation entre écrans de compose

1. Ajouter la dépendance de navigation à votre projet
2. Ecrire les codes des différentes écrans de votre application
3. Créer le graphe de navigation : vous pouvez créer les fichiers suivants dans le package de navigation: **Screen.kt** et **Navgraph.kt**

Ouvrez le fichier Screen.kt puis ajouter le code :

```
sealed class Screen(val route : String){
    object Home :Screen("home_screen")
    object Detail :Screen("detail_screen")
    object Login:Screen("login_screen")
}
```

Navigation avec Jetpack compose

Navigation entre écrans de compose

Ouvrez le fichier NavGraph.kt puis ajoutez ce code:

```
@Composable
fun NavGraph (navController: NavHostController){
   NavHost(
      navController = navController,
      startDestination = Screen.Login.route)
   {
      composable(route = Screen.Login.route){
         LoginScreen()
      }
      composable(route = Screen.Home.route){
         HomeScreen()
      }
      composable(route = Screen.Detail.route){
         DetailScreen()
      }
   }
}
```

Navigation avec Jetpack compose

Navigation entre écrans de compose

4. Maintenant, allez dans notre MainActivity.kt, ici vous devez appeler votre NavGraph à l'intérieur de setContent / NavigationTheme / Surface.

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            NavigationTheme {
                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colors.background
                ) {

                    val navController = rememberNavController()
                    NavGraph(navController = navController)
                }
            }
        }
    }
}
```

Navigation avec Jetpack compose

Navigation entre écrans de compose

- Ici, nous devons créer le **navController** qui sera utilisé dans l'application.

- Tout d'abord, nous créons une nouvelle variable navController en appelant rememberNavController().

- Cette fonction remember retourne toujours le même objet et nous permet de ne pas perdre cet objet si nous apportons des modifications de configuration, telles que changer l'orientation de l'écran.

Navigation avec Jetpack compose

Navigation entre écrans de compose

5. Jusqu'à présent, tout va bien... Maintenant, nous devons ajouter la navigation aux boutons, pour passer d'un écran à un autre.

Allez dans le fichier **NavGraph.kt**, puis ajouter le paramètre **navController** à tous les écrans qui utiliserons la navigation. Par exemple on peut ajouter à LoginScreen et HomeScreen, pour leur permettre d'utiliser le NavGraph lorsque nous cliquons sur le bouton.

Le code du fichier NavGraph.kt devient donc :

Navigation avec Jetpack compose

Navigation entre écrans de compose

```
@Composable
fun NavGraph (navController: NavHostController){
    NavHost(
        navController = navController,
        startDestination = Screen.Login.route)
    {
        composable(route = Screen.Login.route){
            LoginScreen(navController)
        }
        composable(route = Screen.Home.route){
            HomeScreen(navController)
        }
        composable(route = Screen.Detail.route){
            DetailScreen()
        }
    }
}
```

Navigation avec Jetpack compose

Navigation entre écrans de compose

Ensuite, ouvrez les fichiers LoginScreen.kt et HomeScreen.kt et ajoutez le navController en tant que paramètre pour les deux.

```
@Composable
fun LoginScreen(navController: NavController)
```

```
@Composable
fun HomeScreen(navController: NavController)
```

Navigation avec Jetpack compose

Navigation entre écrans de compose

6. Maintenant, à l'intérieur des codes sources de vos écrans, vous pouvez appelez le navigation vers un autre écran à l'intérieur de l'événement onClick du bouton , d'un texte, ou d'un autre widget:

```
onClick = {
   //TODO: Navigate to Home Screen
   navController.navigate(Screens.Home.route)
}
```

Navigation avec Jetpack compose

Passage d'arguments lors de la navigation

1.Dans votre écran source, vous pouvez passer la valeur de "name" à l'aide de la méthode navigate() du NavController :

```
composable(
"destination/{name}",
arguments = listOf(navArgument("name") { type =
NavType.StringType })
) { backStackEntry ->
    DestinationScreen(
        name = backStackEntry.arguments?.getString("name")
    )
}
```

Navigation avec Jetpack compose

Passage d'arguments lors de la navigation

2.Pour envoyer des données à un écran lors de la navigation, vous pouvez utiliser les arguments de navigation.

```
Button(
onClick = {
    navController.navigate("destination/Jean")
}
) {
    Text("Aller à la destination avec le nom Jean")
}
```

Navigation avec Jetpack compose

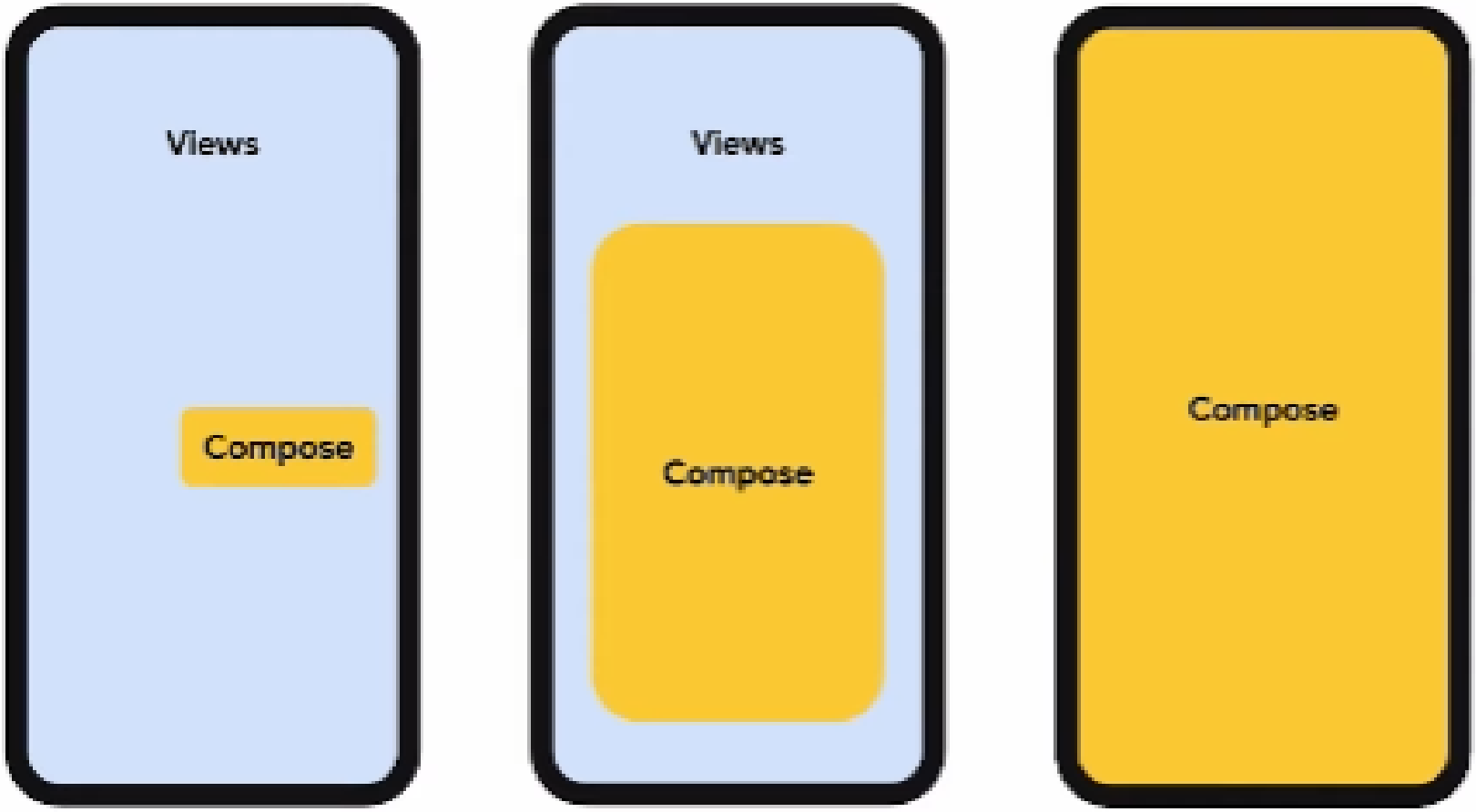
Passage d'arguments lors de la navigation

3. Dans votre écran de destination, vous pouvez récupérer la valeur de "name" en utilisant le constructeur de la destination

```
@Composable
fun DestinationScreen(name: String?) {
    // Utilisez la valeur de "name" ici
}
```

XML & Jetpack compose

Interopérabilité entre XML et Jetpack compose



XML & Jetpack compose

Interopérabilité entre XML et Jetpack compose

Jetpack Compose et XML peuvent coexister dans une même application Android. Les développeurs peuvent utiliser Jetpack Compose pour créer des parties de l'interface utilisateur, tout en utilisant XML pour d'autres parties de l'interface utilisateur.

L'interopérabilité entre Jetpack Compose et XML est possible grâce à l'utilisation de View composables. Les View composables permettent d'intégrer des éléments de l'interface utilisateur basés sur XML dans une interface utilisateur Jetpack Compose.

XML & Jetpack compose

ComposeView

ComposeView est une vue Android qui permet d'afficher et de gérer le contenu Jetpack Compose dans une application Android. Il s'agit d'un élément d'interface utilisateur personnalisé qui peut être ajouté à une mise en page XML ou créé dynamiquement dans le code.

L'utilisation de ComposeView permet d'intégrer progressivement Jetpack Compose dans une application Android existante, en permettant de créer des parties de l'interface utilisateur à l'aide de Jetpack Compose tout en conservant le reste de l'application en XML.

XML & Jetpack compose

ComposeView

ComposeView est une vue Android qui permet d'afficher et de gérer le contenu Jetpack Compose dans une application Android. Il s'agit d'un élément d'interface utilisateur personnalisé qui peut être ajouté à une mise en page XML ou créé dynamiquement dans le code.

L'utilisation de ComposeView permet d'intégrer progressivement Jetpack Compose dans une application Android existante, en permettant de créer des parties de l'interface utilisateur à l'aide de Jetpack Compose tout en conservant le reste de l'application en XML.

XML & Jetpack compose

ComposeView

Voici les étapes à suivre pour installer ComposeView et Jetpack Compose :

1. Ouvrez le fichier build.gradle de votre module d'application.
2. Ajoutez la dépendance à Jetpack Compose dans la section des dépendances :

```
dependencies {
    implementation 'androidx.compose.ui:ui:1.1.0'
    implementation 'androidx.compose.material:material:1.1.0'
    implementation 'androidx.compose.runtime:runtime-livedata:1.1.0'
}
```

XML & Jetpack compose

ComposeView

3. Assurez-vous que la version de **compose.ui, compose.material** et **compose.runtime-livedata** correspond à la version de Jetpack Compose que vous souhaitez utiliser.
4. Synchronisez votre projet avec Gradle pour télécharger les dépendances.
5. Après avoir installé Jetpack Compose, vous pouvez utiliser **ComposeView** dans votre code en important la classe androidx.compose.ui.platform.ComposeView. Vous pouvez ensuite créer une instance de ComposeView dans votre code et lui attribuer une fonction Composable pour afficher le contenu Jetpack Compose.

XML & Jetpack compose

ComposeView

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

<androidx.compose.ui.platform.ComposeView
        android:id="@+id/myComposeView"
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"/>

</LinearLayout>
```

XML & Jetpack compose

ComposeView

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        // inflate the XML layout
        setContentView(R.layout.activity_main)

        // reference the ComposeView using findViewById
        val composeView = findViewById<ComposeView>(R.id.myComposeView)

        // set up the Composable function for the ComposeView
        composeView.setContent {
            // define the user interface using Jetpack Compose
        }
    }
}
```

SÉANCE 13

SQLITE

Le cours de cette semaine porte sur les bases de données SQLite .

- Création d'une base de données SQLite
- Requêtes SQL en Kotlin

Introduction à SQLite

Présentation de SQLite

- SQLite est un système de gestion de base de données relationnelle open-source qui a été développé par Richard Hipp en 2000.
- Il est intégré dans la plupart des systèmes d'exploitation, y compris Android, iOS et Windows, et est largement utilisé pour stocker des données dans les applications mobiles, les navigateurs web, les systèmes de gestion de contenu, etc.
- SQLite est un système de gestion de base de données relationnelle léger qui ne nécessite pas de configuration de serveur.
- Les données sont stockées dans un seul fichier de base de données qui peut être transféré facilement d'un emplacement à un autre.

Introduction à SQLite

Présentation de SQLite

- Il est également rapide et efficace, avec des performances comparables à celles d'autres systèmes de gestion de base de données relationnelle tels que MySQL et PostgreSQL.

- En plus d'être léger et rapide, SQLite offre une gamme de fonctionnalités telles que la prise en charge de transactions ACID (Atomicité, Cohérence, Isolation, Durabilité), la gestion des contraintes de clé primaire et étrangère, et la prise en charge de nombreuses fonctions SQL courantes.

- Il est également compatible avec de nombreux langages de programmation, y compris Kotlin, Java, C et Python.

Introduction à SQLite

Kotlin et SQLite

Kotlin simplifie la gestion de la base de données SQLite en offrant des fonctionnalités spécifiques qui facilitent la gestion des données, réduisent le code nécessaire pour effectuer des opérations sur la base de données et améliorent la sécurité du code.

Voici quelques exemples de la façon dont Kotlin simplifie la gestion de la base de données SQLite :

- **Sécurité de type et nullabilité**: Kotlin offre une sécurité de type qui permet aux développeurs de détecter les erreurs de type à la compilation plutôt qu'à l'exécution. Cela aide à prévenir les erreurs courantes telles que les exceptions de pointeur null (NullPointerException) et les erreurs de type lors de la manipulation de données dans la base de données SQLite.

Introduction à SQLite

Kotlin et SQLite

- **Fonctions d'extension** : Kotlin permet aux développeurs d'ajouter des fonctions d'extension à des classes existantes. Cela permet de simplifier le code et d'améliorer la lisibilité en encapsulant des fonctionnalités courantes dans des fonctions réutilisables qui peuvent être utilisées à plusieurs endroits dans le code.
- **Fonctions lambda** : Kotlin prend en charge les fonctions lambda, ce qui facilite la création de requêtes SQL complexes. Les fonctions lambda permettent aux développeurs d'écrire des requêtes SQL de manière plus concise et de les rendre plus faciles à lire et à comprendre.

Introduction à SQLite

Kotlin et SQLite

- **Prise en charge de l'interopérabilité** : Kotlin est conçu pour être compatible avec Java et peut facilement interagir avec du code Java existant. Cela facilite l'utilisation de bibliothèques et d'outils existants pour la gestion de la base de données SQLite dans Kotlin.

Base de données SQLite

Création de BD et de ses tables

Pour créer une base de données SQLite avec Kotlin dans une application Android, vous devez suivre les étapes suivantes :

1. Importez la bibliothèque SQLite dans votre projet Kotlin. Vous pouvez le faire en ajoutant la dépendance suivante dans votre fichier build.gradle :

```
dependencies {
    implementation 'androidx.sqlite:sqlite:2.1.0'
}
```

2. Créez une classe de gestionnaire de base de données qui étend SQLiteOpenHelper. Cette classe gérera la création de la base de données et la mise à niveau de la base de données au fil du temps. Voici un exemple de code :

Base de données SQLite

Création de BD et de ses tables

```
class DatabaseHelper(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME, null,
DATABASE_VERSION) {

    override fun onCreate(db: SQLiteDatabase?) {
        db?.execSQL(CREATE_TABLE)
    }

    override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int, newVersion: Int) {
        db?.execSQL("DROP TABLE IF EXISTS $TABLE_NAME")
        onCreate(db)
    }

    companion object {
        private const val DATABASE_VERSION = 1
        private const val DATABASE_NAME = "my_database"
        private const val TABLE_NAME = "my_table"
        private const val COLUMN_ID = "id"
        private const val COLUMN_NAME = "name"
        private const val CREATE_TABLE = "CREATE TABLE $TABLE_NAME ($COLUMN_ID INTEGER
PRIMARY KEY AUTOINCREMENT, $COLUMN_NAME TEXT)"
    }
}
```

Base de données SQLite

Création de BD et de ses tables

3. Utilisez la classe de gestionnaire de base de données pour créer une instance de base de données dans votre code Kotlin. Vous pouvez le faire en utilisant le code suivant :

```
val dbHelper = DatabaseHelper(this)
val db = dbHelper.writableDatabase
```

4. Utilisez la méthode execSQL pour exécuter une requête SQL pour créer une table dans la base de données :

```
val createTableQuery = "CREATE TABLE my_table (id INTEGER PRIMARY KEY
AUTOINCREMENT, name TEXT)"
db.execSQL(createTableQuery)
```

Base de données SQLite

Clé primaire et clé étrangère

Pour ajouter une contrainte de clé primaire à une colonne, vous pouvez ajouter l'attribut PRIMARY KEY à la définition de la colonne lors de la création de la table. Par exemple, pour ajouter une clé primaire à la colonne "id" de la table "users" :

```
override fun onCreate(db: SQLiteDatabase?) {
   db?.execSQL("CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT,
email TEXT)")
}
```

Base de données SQLite

Clé primaire et clé étrangère

Pour ajouter une contrainte de clé étrangère à une colonne, vous pouvez ajouter l'attribut FOREIGN KEY à la définition de la colonne lors de la création de la table. Par exemple, pour ajouter une clé étrangère à la colonne "user_id" de la table "posts" :

```
override fun onCreate(db: SQLiteDatabase?) {
   db?.execSQL("CREATE TABLE posts (id INTEGER PRIMARY KEY, title TEXT, content TEXT,
user_id INTEGER, FOREIGN KEY(user_id) REFERENCES users(id))")
}
```

Base de données SQLite

Manipulation des données

La manipulation des données dans SQLite concerne les opérations suivantes:

- l'ajout des enregistrements
- La mise à jour des données
- La sélection ou l'affichage des données
- La suppression des données

Base de données SQLite

Ajouter les enregistrements

Pour insérer des données dans une table vous pouvez utiliser la méthode **insert** de la classe SQLiteDatabase.
Voici un exemple de code pour insérer un nouvel utilisateur avec un nom et une adresse e-mail :

```
val dbHelper = DatabaseHelper(this)
val db = dbHelper.writableDatabase

val values = ContentValues().apply {
    put("name", nom.text.toString())
    put("email", mail.text.toString())
}
val newRowId = db.insert("users", null, values)

if (newRowId == -1L) {
    // L'insertion a échoué
    Toast.makeText(this,"Erreur lors de l'insertion!",Toast.LENGTH_SHORT).show()
} else {
    // L'insertion a réussi
    Toast.makeText(this,"Insertion réussie avec l'ID $newRowId",Toast.LENGTH_SHORT).show()
}
```

Base de données SQLite

Affichage des données

Pour afficher les données stockées dans une base de données SQLite avec Kotlin, vous pouvez utiliser une requête SQL **SELECT** pour récupérer les données souhaitées. Vous pouvez ensuite parcourir le curseur résultant et afficher les données.

Voici un exemple de code qui montre comment afficher toutes les données de la table "users" dans une ListView :

Base de données SQLite

Affichage des données

```
val dbHelper = DatabaseHelper(this)
val db = dbHelper.readableDatabase

val cursor = db.rawQuery("SELECT * FROM users", null)

val userList = ArrayList<String>()
if (cursor.moveToFirst()) {
    do {
        val name = cursor.getString(cursor.getColumnIndex("name"))
        val email = cursor.getString(cursor.getColumnIndex("email"))
        val userInfo = "Nom : $name | email : $email "
        userList.add(userInfo)
    } while (cursor.moveToNext())
}

val listView = findViewById<ListView>(R.id.listview)
val adapter = ArrayAdapter(this, android.R.layout.simple_list_item_1, userList)
listView.adapter = adapter
```

Base de données SQLite

Affichage des données

- Dans cet exemple, nous récupérons toutes les données de la table "users" à l'aide de la méthode **rawQuery**, qui prend une requête SQL en paramètre et renvoie un curseur contenant les résultats.
- Nous parcourons ensuite le curseur à l'aide d'une boucle **while** et récupérons les valeurs de chaque colonne à l'aide des méthodes **getString** du curseur.
- Nous stockons ensuite les informations de chaque utilisateur dans une chaîne de caractères et les ajoutons à une liste d'utilisateurs.
- Enfin, nous utilisons un ArrayAdapter pour afficher la liste d'utilisateurs dans une ListView.
- Notez que dans cet exemple, nous avons utilisé une méthode **readableDatabase** pour obtenir une référence à la base de données. Si vous avez besoin de modifier les données de la base de données, vous devrez utiliser la méthode **writableDatabase** à la place.

Base de données SQLite

Mises à jour des données

Pour mettre à jour des données existantes dans une table, vous pouvez utiliser la méthode **update** de la classe SQLiteDatabase. Voici un exemple de code pour mettre à jour l'adresse e-mail de l'utilisateur ayant un ID spécifique :

```
val values = ContentValues().apply {
    put("email", "newemail@example.com")
}
val selection = "id = ?"
val selectionArgs = arrayOf("1")
val count = db.update("users", values, selection, selectionArgs)
```

Base de données SQLite

Suppression des données

Pour supprimer des données de la table "users", vous pouvez utiliser la méthode **delete** de la classe SQLiteDatabase. Voici un exemple de code pour supprimer un utilisateur ayant un ID spécifique :

```
val selection = "id = ?"
val selectionArgs = arrayOf("1")
val deletedRows = db.delete("users", selection, selectionArgs)
```

Base de données SQLite

Exercice

Exercice :

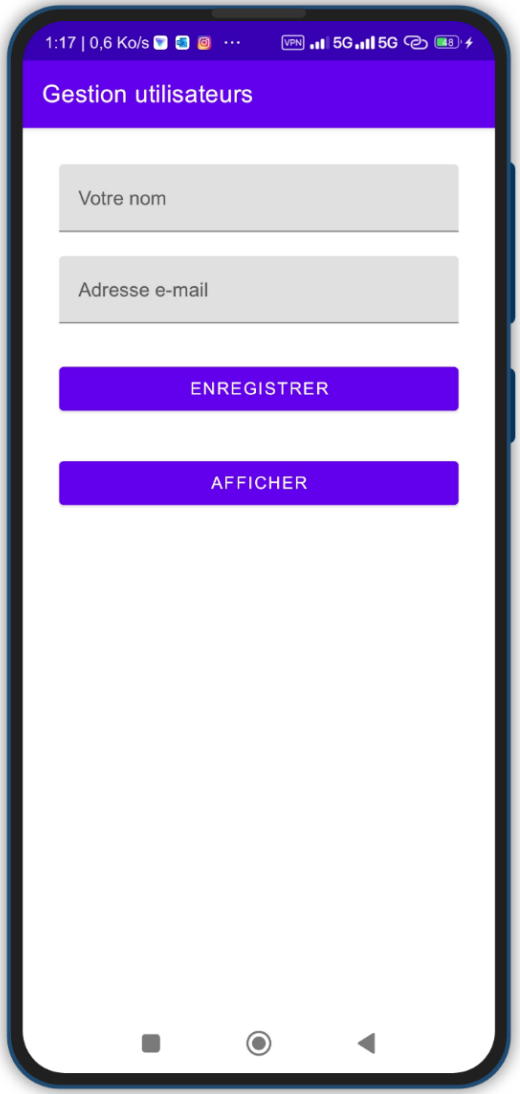
Créer une application Android qui permettant de gérer les utilisateurs d'un site internet.

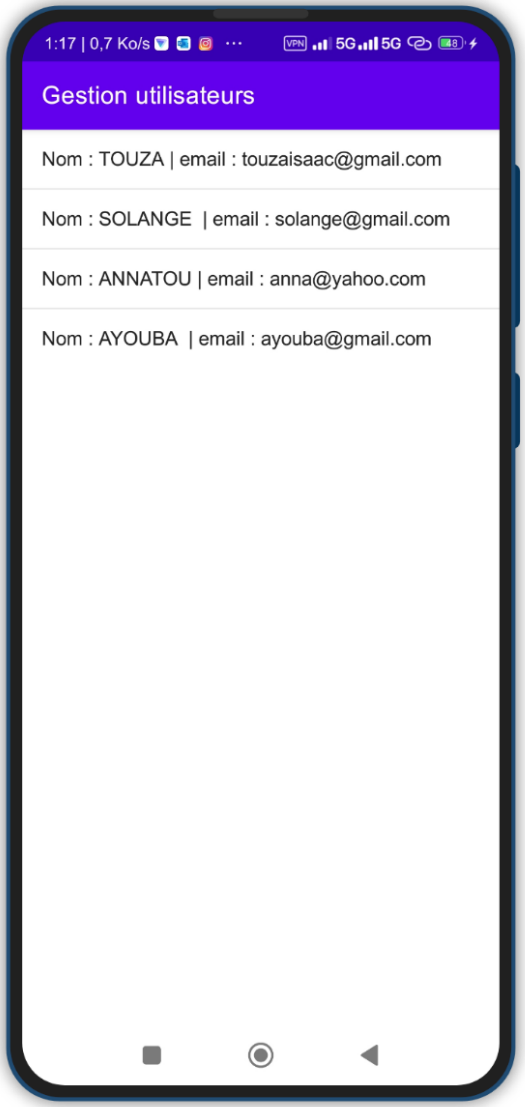
L'application doit permettre d'enregistré, de modifier de supprimer et de mettre à jour les informations sur les utilisateurs.

Un utilisateur dispose d'un identifiant, d'un nom , d'un mot de passe, d'un adresse e-mail, d'un numéro de telephone et d'une adresse.

Base de données SQLite

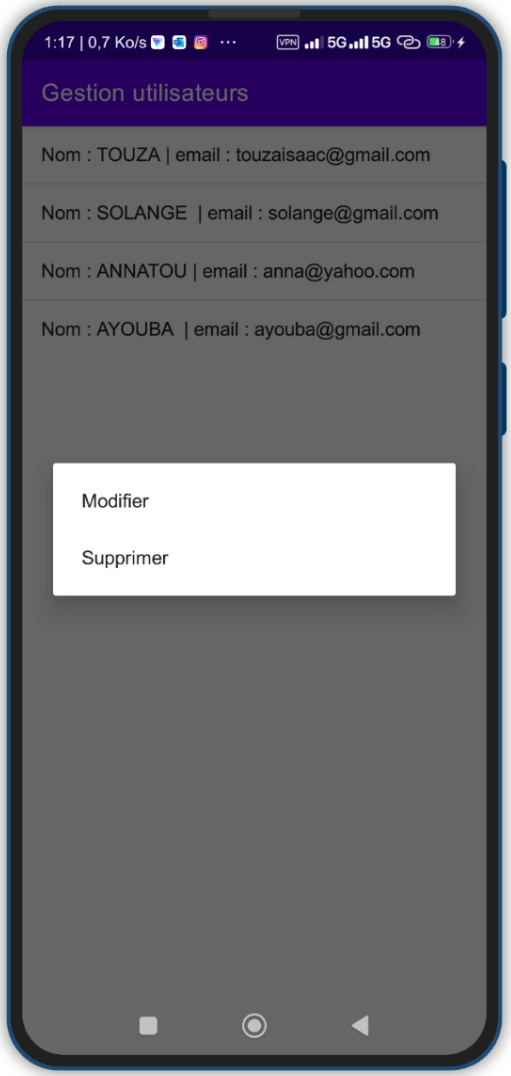
Exercice

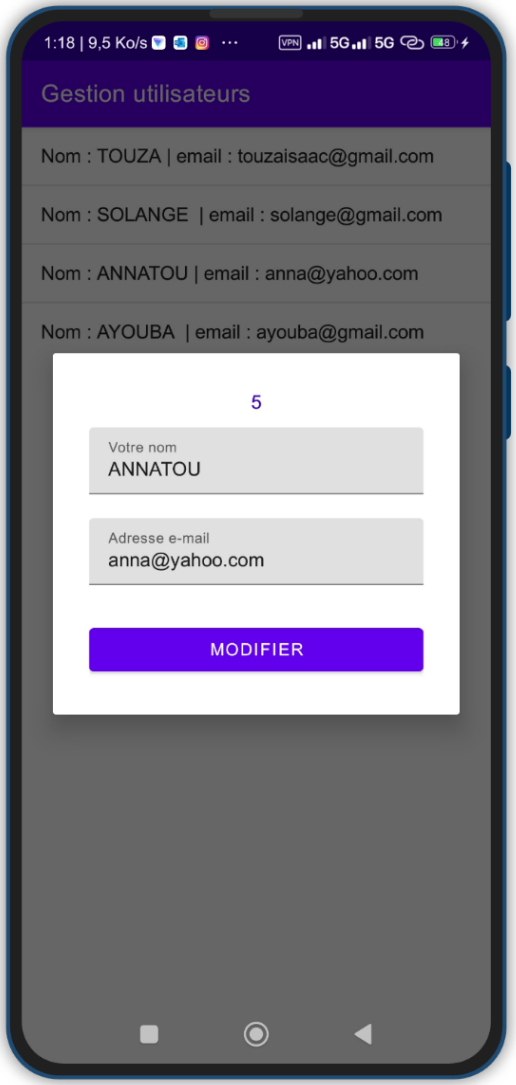




Base de données SQLite

Exercice





Base de données SQLite

La jointure

Une jointure (ou "join" en anglais) est une opération SQL permettant de combiner les données de deux ou plusieurs tables en une seule.

En général, une jointure est effectuée en spécifiant une ou plusieurs colonnes communes entre les tables, qui sont utilisées pour associer les lignes des tables.

Pour effectuer une jointure SQLite en Kotlin, vous pouvez utiliser la méthode **rawQuery**() du **SQLiteDatabase**

Base de données SQLite

La jointure

Exemple:

Supposons que nous avons deux tables : **users** et **orders**. La table users contient des informations sur les utilisateurs, notamment leur nom et leur adresse électronique, et la table orders contient des informations sur les commandes passées par ces utilisateurs, notamment la date de commande et le montant total de la commande. Les deux tables sont liées par l'identifiant de l'utilisateur, qui est une clé étrangère dans la table orders.

Nous voulons obtenir les informations sur les commandes et les utilisateurs associées

Base de données SQLite

La jointure

```
val dbHelper = DatabaseHelper(this)
val db = dbHelper.readableDatabase

val projection = arrayOf("users.name", "orders.product")
val selection = "users.id = orders.user_id"
val sortOrder = "users.name DESC"

val cursor = db.query(
    "users INNER JOIN orders ON users.id = orders.user_id", // Tables et jointure
    projection, // Colonnes à retourner
    selection, // Clause WHERE
    null, // Arguments de la clause WHERE
    null, // GROUP BY
    null, // HAVING
    sortOrder // Clause ORDER BY
)

val userList = ArrayList<String>()
if (cursor.moveToFirst()) {
    do {
        val name = cursor.getString(cursor.getColumnIndex("name"))
        val product = cursor.getString(cursor.getColumnIndex("product"))
        val userInfo = "Nom : $name | Produit : $product"
        userList.add(userInfo)
    } while (cursor.moveToNext())
}
```

Base de données SQLite

La jointure

Dans cet exemple, nous utilisons la méthode **query**() pour effectuer une jointure entre les tables **users** et **orders** en utilisant une clause **INNER JOIN** avec la condition **users.id = orders.user_id**. Nous spécifions également les colonnes à retourner (projection), la clause WHERE (selection) et la clause ORDER BY (sortOrder).

Base de données SQLite

Exercice

Exercice pratique: 1/3

Supposons que vous développez une application Android pour une entreprise qui vend des produits électroniques. L'application permettra aux clients de parcourir les produits disponibles, de les ajouter à leur panier et de passer des commandes. Vous êtes chargé de créer la base de données SQLite pour cette application.

Voici les informations que vous devez stocker pour chaque table :

- La table "Produits" doit stocker les informations suivantes pour chaque produit : l'identifiant du produit (entier), le nom du produit (chaîne), la description du produit (chaîne), le prix unitaire du produit (décimal), la quantité disponible en stock (entier).

Base de données SQLite

Exercice

Exercice pratique : 2/3

- La table "Clients" doit stocker les informations suivantes pour chaque client : l'identifiant du client (entier), le nom du client (chaîne), l'adresse e-mail du client (chaîne), le numéro de téléphone du client (chaîne).
- La table "Commandes" doit stocker les informations suivantes pour chaque commande : l'identifiant de la commande (entier), la date de la commande (date/heure), l'identifiant du client qui a passé la commande (entier), l'état de la commande (en attente, expédiée, livrée, annulée).
- La table "CommandeProduits" doit stocker les informations suivantes pour chaque produit inclus dans une commande : l'identifiant de la commande (entier), l'identifiant du produit (entier), la quantité commandée (entier).

Base de données SQLite

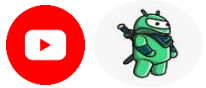
Exercice

Exercice pratique : 3/3

En utilisant Kotlin pour Android, écrivez le code nécessaire pour créer la base de données SQLite avec ces tables et leurs relations. Assurez-vous d'utiliser les contraintes de clé primaire et étrangère pour garantir l'intégrité des données.

Et C’est tout !

C’est fini avec ce cours
Rendez vous sur :



https://www.youtube.com/@StevdzaSan

https://www.youtube.com/@AndroidDevelopers

Pour plus des contenus sur jetpack compose