Développement d'application mobile natif

Chapitre 1: Introduction au développement mobile

Hend Ben Ayed Kharrat & Marwa Chaabani

AU: 2025-2026

L3DSI

PLAN


Présentation d'Android
Langage de programmation Kotlin
Découverte de l'environnement de développement

Application native, hybride, cross-platform ou web, comment faire son choix ?

Trois types de programmation mobile :

- Native : applications codées en Java, C++, Kotlin, compilées et livrées avec leurs données sous forme d'archive Jar (fichier APK). Ce type concerne un seul système d'exploitation.

Société	**Appareils**	**Système d'exploitation**	**Plateforme de téléchargement**	**Langages de programmation**
Apple	iPhone / iPad	iOS	App Store	Objective C / Swift
Google	Samsung / Google Phone et tablette / Motorola / ...	Android	Google Play	Java, **Kotlin**
Microsoft	Windows Phone	Windows Phone	Windows Store	C#

•**Web** : applications accessibles via un navigateur internet, développées en HTML5, CSS3, JavaScript, utilisant des frameworks comme Node.js, Angular, React, ou Vue.

•**Hybride** : applications créées avec des frameworks tels qu'**Ionic**, **Apache Cordova** (anciennement PhoneGap)… Ces frameworks masquent les spécificités des systèmes, permettant à la même application de fonctionner de manière identique sur plusieurs plateformes (Android, iOS, Windows, Linux, etc.).

•**Cross-Plateformes:** des applications pour iOS et Android à partir d'une seule base de code. Contrairement aux applications natives (écrites séparément en Swift/Objective-C pour iOS et en Java/Kotlin pour Android), ou aux applications hybrides (qui utilisent un conteneur natif : WebView), les Framework multiplateformes sont conçues pour produire des applications qui s'exécutent avec des performances proches du natif , comme **Flutter , Xamarin** ..

Type d'Application	Idéal pour	Avantages	Considérations
Natives	Haute performance	Performances et expérience utilisateur optimales.	Coût et temps de développement élevés.
Cross-Plateformes	Code unique pour performances natives.	Code unique, réduction des coûts.	Accès parfois limité aux fonctions avancées de l'appareil.
Hybrides	Budget limité	Code unique, compatible avec plusieurs plateformes.	Performances inférieures, accès limité aux fonctionnalités.
Web	Accessibilité universelle	Compatibilité avec tous les navigateurs.	Dépend de la connexion Internet, accès très limité aux fonctions de l'appareil.

Tour d'horizon

- Septembre 2008 : 1ère version finale avec le téléphone HTC Dream.

Android Versions History:2008-2024 | Android Evolution

YouTube

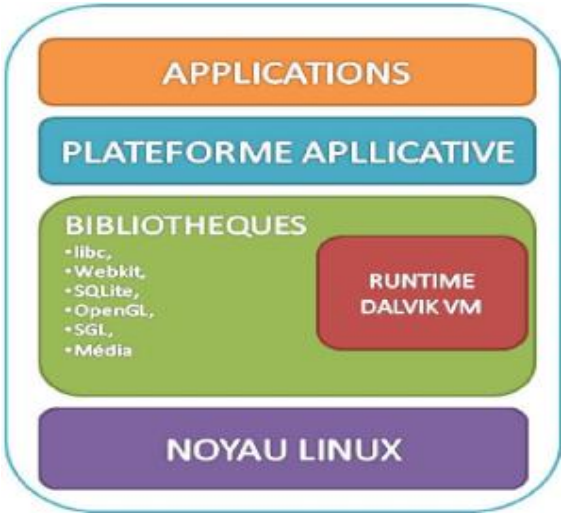
Définition de l'Android

- C'est un système d'exploitation open source, exclusivement centré sur la téléphonie mobile.
- C'est aussi un ensemble de logiciels destinés à fournir une solution clé en main pour les appareils mobiles, smartphone et tablettes tactiles. Cet ensemble comporte un système d'exploitation (comprenant un noyau Kernel Linux et une licence Apache 2.0), des applications clés telles que le navigateur web, le téléphone et le carnet d'adresses,….
- C'est aussi une plateforme de développement open-source pour la création des applications portables.

COMPOSANTS D'ANDROID

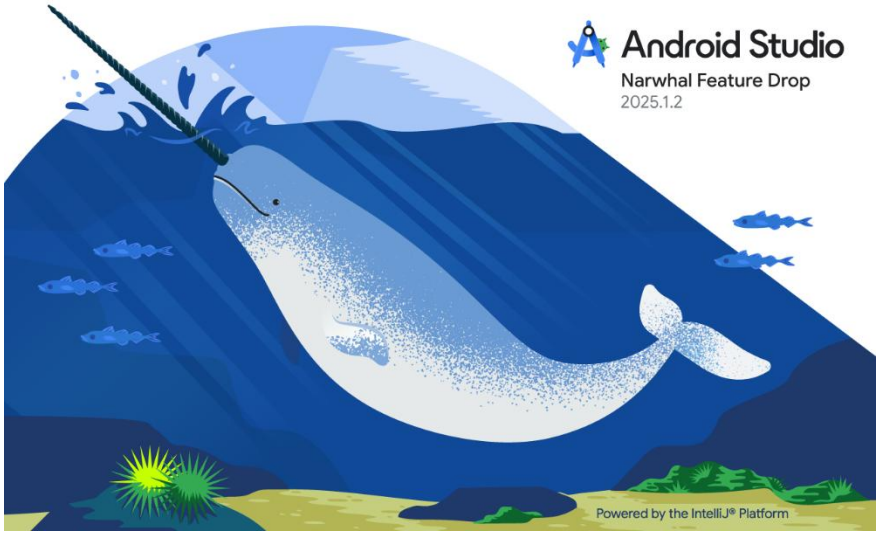
La plate-forme Android est composée de différentes couches :

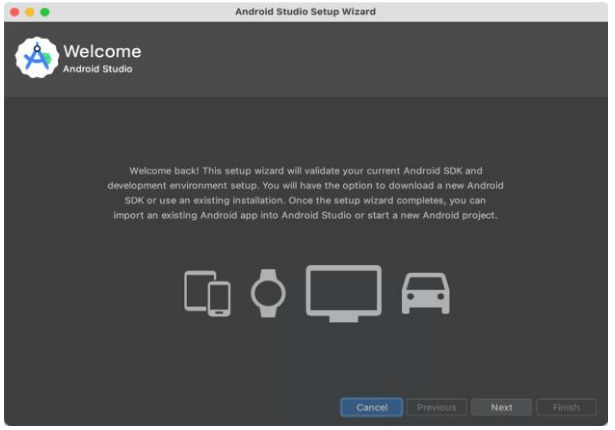
- Un noyau Linux qui lui confère notamment des caractéristiques multitâches ;
- Des bibliothèques graphiques, multimédias ;
- Une machine virtuelle Java adaptée : la « Dalvik Virtual Machine »;
- Un Framework applicatif proposant des fonctionnalités de gestion de fenêtres, de téléphonie, de gestion de contenus… ;
- Des applications dont un navigateur Web, une gestion des contacts, un calendrier…



Environnement de développement

- ✓ Installation de l'Android Studio (**Android Studio Narwhal 3 Feature Drop | 2025.1.3**
- ✓ (https://developer.android.com/studio)





Vous pouvez suivre : installer Android Studio (https://developer.android.com/studio/install?hl=fr) comme guide d'installation
Ou bien https://youtu.be/uvjMD_prpZo?si=HbVOxUZ3qOSDzYL4

PRÉREQUIS MATÉRIEL

Exigence	Minimum
OS	Microsoft Windows 10 64 bits
RAM	Studio : 8 Go Studio et émulateur : 16 Go
Processeur	Virtualisation requise (Intel VT-x ou AMD-V, activée dans le BIOS). Microarchitecture du processeur après 2017. Intel Core de 8e génération i5 / AMD Zen Ryzen (par exemple, Intel i5-8xxx, Ryzen 1xxx).
Espace disque	Studio : 8 Go d'espace libre. Studio et émulateur : 16 Go d'espace libre
Résolution d'écran	1 280 x 800

LANGAGE DE PROGRAMMATION
KOTLIN

KOTLIN

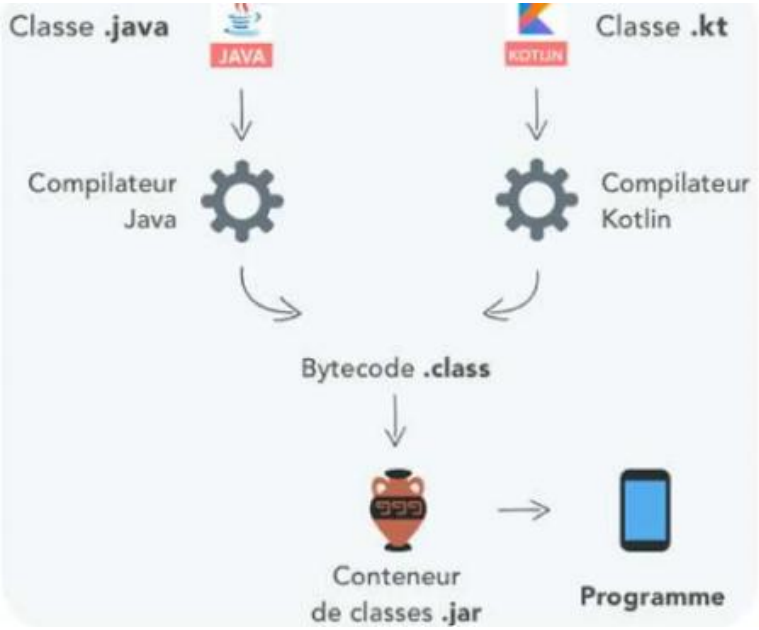
- Kotlin est un langage de programmation orienté objet, fonctionnel, avec un typage statique qui permet de compiler sur la machine virtuelle Java et JavaScript.
- Son développement provient principalement d'une équipe de programmeurs chez JetBrains basée à Saint-Pétersbourg en Russie. Le nom de ce langage vient de l'île de Kotline situé en mer Baltique, à 25 km de St. Pétersbourg.
- Google annonce pendant la conférence Google I/O 2017 que Kotlin devient le second langage de programmation officiellement pris en charge par Android après Java.

AVANTAGES

- **Typage statique :** Les types de variables sont connus et vérifiés à la compilation.
- **Concise :** Le code est plus court et plus facile à lire que celui de Java.
- **Sécurisé :** Kotlin gère la sécurité des valeurs "nulles" de manière à éviter les erreurs courantes d'exécution (NullPointerException).
- **Explicite :** Le code est clair et facile à comprendre.
- **Facile à apprendre :** La syntaxe est simple et intuitive
- **Capacités fonctionnelles et orientées objet :** combine la programmation orientée objet et les concepts de programmation fonctionnelle, ce qui le rend flexible et puissant.
- **Entièrement interopérable avec Java :** Le code Kotlin peut être utilisé dans des projets Java et inversement.
- **Multiplateforme :** des applications pour différentes plateformes, notamment les serveurs, Android, les navigateurs web (via Kotlin/JS) et les ordinateurs de bureau.
- **Gratuit et Open Source**.
- **Excellent outillage :** Il bénéficie d'un excellent support d'outils, comme Android Studio, qui améliorent l'expérience de développement.

L'INTEROPÉRABILITÉ DE KOTLIN AVEC JAVA

- Kotlin est entièrement **interopérable** avec Java. Vous pouvez accéder au code Java à partir de Kotlin et vice-versa, sans aucune difficulté.
- Il est même possible d'utiliser les deux langages dans un même projet, ce qui facilite grandement l'intégration progressive de Kotlin dans vos projets Java existants.



Kotlin Programming

Syntaxe
Variables
Fonctions
....

COMMENTAIRES

- // pour un commentaire sur une seule ligne.
- /* ... */ pour un commentaire sur plusieurs lignes.

La fonction principale (main)

Dans tout programme Kotlin, il doit y avoir une fonction nommée **main**. C'est la première fonction qui est exécutée lorsque vous lancez le programme.

Voici deux façons de la déclarer :

1.fun main() {} : C'est la déclaration la plus simple.

2.fun main(args: Array<String>) {} : Cette version permet de passer des arguments en ligne de commande. Elle est utilisée lorsque vous compilez et exécutez le programme à partir de la ligne de commande.

TYPES DE VARIABLES

Nombres (Numbers)

- Les nombres se divisent en deux catégories :
- **Entiers (Integers) :**
  - **Byte :** 1 octet, de −27 à 27−1.
  - **Short :** 2 octets, de −215 à 215−1.
  - **Int :** 4 octets, de −231 à 231−1.
  - **Long :** 8 octets, de −263 à 263−1.
- **Nombres à virgule flottante (Floating-point) :**
  - **Float :** 4 octets, avec 24 bits significatifs et 7 décimales.
  - **Double :** 8 octets, avec 53 bits significatifs et 16 décimales.

Autres types

- Le type **Boolean** représente des valeurs logiques (true ou false).
- Le type **Char** représente un seul caractère.
- Le type **String** représente une chaîne de caractères.
- Le type **Array** représente un tableau d'éléments.

LES OPÉRATEURS

- **Opérateurs arithmétiques** : + - / * %
- **Opérateurs relationnels** : > < >= <= == !=
- **Opérateurs d'affectation** : := += -= *= /= %=
- **Opérateurs unaires** : + - ++ -- !
- **Opérateurs logiques** : && ||

DÉCLARATION DES VARIABLES

- En Kotlin, on peut déclarer une variable en utilisant les mots-clés val et var.

Caractéristique	val	var
Signification	Vient de "value" (valeur)	Vient de "variable"
Mutabilité	**Immuable**	**Muable**
Modification	Ne peut pas être réaffecté après son initialisation.	Peut être réaffecté à tout moment après son initialisation.
Équivalent Java	final	Variable classique (par défaut)
Utilisation	Pour les valeurs qui ne doivent jamais changer (constantes, données fixes, etc.).	Pour les valeurs qui sont amenées à évoluer (compteurs, scores, données utilisateur modifiables, etc.).
Bonne pratique	À **privilégier** par défaut pour un code plus sûr et plus lisible.	À utiliser uniquement lorsque la modification est nécessaire.

EXEMPLE DE DECLARATION

```
fun main() {
    // Déclaration et initialisation du nom de l'utilisateur
    val nomUtilisateur = "Jean Dupont"
    // Déclaration et initialisation d'un identifiant utilisateur
    val userId = 12345
    println("Nom de l'utilisateur : $nomUtilisateur")
    println("ID de l'utilisateur : $userId")
    // Ligne suivante qui va provoquer une erreur de compilation !
    // nomUtilisateur = "Pierre Martin" // Erreur : "Val cannot be reassigned"
}
```

```
fun main() {
    // Déclaration et initialisation du nombre de vies
    var vies = 3
    println("Vies restantes : $vies")

    // Le personnage est touché, son nombre de vies diminue
    vies = vies - 1
    println("Vies restantes après avoir été touché : $vies")
    // Le personnage gagne une vie grâce à un bonus
    vies = vies + 1
    println("Vies après avoir récupéré un bonus : $vies")
}
```

LA CONVERSION DE TYPE EN KOTLIN

- En Kotlin, la conversion de type d'une variable **n'est pas implicite**, ce qui signifie que vous ne pouvez pas directement attribuer un type plus petit à un type plus grand sans une conversion explicite. Cela évite les erreurs de données.
- Pour effectuer une conversion, vous devez utiliser des fonctions spécifiques, comme :
- toByte() toShort() toInt() toLong() toFloat() toDouble()
- Par exemple, si vous voulez convertir un nombre à virgule flottante (1.0) en un entier (Int), vous devez explicitement utiliser la fonction **toInt()**. L'image ci-dessous illustre l'erreur qui se produit sans conversion et comment la corriger.

```
fun main() {
    // Mauvaise pratique : ne fonctionne pas
    // var a: Int = 1.0

    // Bonne pratique : utilisation de toInt()
    var a: Int = 1.0.toInt()
}
```

GESTION DES VALEURS NULLES (NULL SAFETY)

- En Kotlin, les variables **ne peuvent pas contenir de valeur nulle** par défaut. Cette fonctionnalité, appelée "Null Safety", a été conçue pour prévenir l'une des erreurs les plus fréquentes en programmation : la NullPointerException.

- Pour autoriser une variable à avoir une valeur nulle, vous devez le faire de manière explicite en ajoutant un **point d'interrogation** (?) après son type.

```
fun main() {
    // Mauvaise pratique : ne fonctionne pas car 'a' ne peut pas être null
    // var a: Int = null

    // Bonne pratique : le point d'interrogation autorise la valeur nulle
    var a: Int? = null
}
```

APPEL DE FONCTIONS SUR DES VARIABLES "NULLABLES"

- Lorsque vous appelez une fonction sur une variable qui peut être nulle (déclarée avec ?), vous devez utiliser l'opérateur **?** avant d'appeler la fonction. Cela garantit que la fonction ne sera exécutée que si la variable n'est pas nulle, évitant ainsi un crash du programme.

```
fun main() {
    var a: String? = "rssp"

    // Mauvaise pratique : peut causer une NullPointerException si 'a' est null
    // println(a.toUpperCase())

    // Bonne pratique : '?' vérifie si 'a' est non null avant d'appeler la fonction
    println(a?.toUpperCase())
}
```

OPÉRATEUR ELVIS (?:)

- L'opérateur Elvis, noté **?:**, est un outil puissant de la gestion de la null safety. Il vous permet de fournir une **valeur par défaut** dans le cas où une expression serait null.

Syntaxe: expression_potentiellement_nulle ?: valeur_par_défaut

```
fun findName(user: User?): String {
    // Si user est null, user.name est null.
    // L'opérateur Elvis renvoie alors "Inconnu".
    return user?.name ?: "Inconnu"
}
data class User(val name: String?)
fun main() {
    val user1 = User(name = "Alice")
    val user2 = null
    val user3 = User(name = null)
    println(findName(user1)) // Affiche : Alice
    println(findName(user2)) // Affiche : Inconnu
    println(findName(user3)) // Affiche : Inconnu
}
```

LE MOT-CLÉ CONST

- Le mot-clé **const** est utilisé conjointement avec **val** pour définir une constante dont la valeur est **connue au moment de la compilation**.
- Cette constante est généralement de portée globale.
- Sa valeur ne peut **pas être modifiée** lors de l'exécution du programme.

```
const val a: String;
```

est incorrect car la variable n'est pas initialisée. Le compilateur doit connaître la valeur de la constante dès le début, donc il y a une erreur.

```
const val a: String="rssp";
```

est correct car la variable est initialisée avec une valeur fixe ("rssp") qui est connue et peut être "gravée" dans le code binaire au moment de la compilation.

LA DIFFÉRENCE ENTRE CONST ET VAL

- On pourrait penser que const et val sont identiques, car tous les deux créent des variables immuables. Cependant, il existe une différence fondamentale dans leur fonctionnement.
- **const val** est une véritable **constante de compilation**. Sa valeur doit être connue et fixée au moment où le programme est compilé. Elle est directement "gravée" dans le code binaire.
- **val** est une **constante d'exécution**. Sa valeur est affectée au moment de l'exécution du programme. La valeur n'a pas besoin d'être connue au moment de la compilation.
- En résumé, si la valeur est une constante absolue, gravée dans le marbre avant même que le programme ne s'exécute, utilisez **const**. Pour toute autre variable immuable dont la valeur est calculée ou fournie pendant l'exécution, utilisez **val**.

LE MOT-CLÉ LATEINIT

- En Kotlin, une variable de type non primitif (une classe) doit être initialisée au moment de sa déclaration. Cependant, dans certains cas, comme le chargement d'une page ou d'un écran, vous pourriez vouloir l'initialiser plus tard.
- Si vous ne lui donnez pas de valeur immédiate, Kotlin vous renverra une erreur de compilation.
- Pour contourner ce problème, vous pouvez utiliser le mot-clé **lateinit** (pour "Late-Initialized", initialisée plus tard). Ce mot-clé indique à Kotlin que vous êtes certain que la variable sera initialisée avant d'être utilisée, même si ce n'est pas fait immédiatement.

```
// Mauvaise pratique : ne fonctionnera pas sans valeur initiale
// var a: String

// Bonne pratique : 'lateinit' indique une initialisation future
lateinit var a: String
```

FONCTIONS POUR LA SORTIE (OUTPUT)
FONCTIONS POUR L'ENTRÉE (INPUT)

Fonctions pour la sortie (Output)

•**`print()`** : Permet d'écrire une chaîne de caractères sur la console sans passer à la ligne.

•**`println()`** : Permet d'écrire une chaîne de caractères sur la console, suivie d'un saut de ligne.

Fonctions pour l'entrée (Input)

•**`readLine()`** : Lit une ligne de texte depuis la console et la retourne sous forme de chaîne de caractères (`String`).

STRUCTURES DE CONTRÔLES CONDITIONNELS

```
fun main() {
    var a = 0

    if (a < 0) {
        println("$a est négatif")
    } else if (a > 0) {
        println("$a est positif")
    } else {
        println("$a est null")
    }
}
```

```
fun main() {
    var a = 1
    when(a) {
        0 -> println("$a = zéro")
        1 -> println("$a = un")
        2 -> println("$a = deux")
        else -> println("$a = autre")
    }
}
```

les structures de contrôle conditionnelles sont des **expressions** et peuvent donc **retourner une valeur**.

```
fun main() {
    var a = -1
    val result = if (a < 0) "$a est négatif" else if (a > 0) "$a est positif" else "$a est nul"
    println(result)
}
fun main() {
    var a = -1
    println(if (a < 0) "$a est négatif" else if (a > 0) "$a est positif" else "$a est nul")
}
```

LES STRUCTURES DE CONTRÔLE RÉPÉTITIVE

Les trois types de boucles

- **for** : for(i in 1..5) pour parcourir les nombres de 1 à 5.
- **while** : while(i <= 5) répète le bloc tant que i est inférieur ou égal à 5.
- **do/while** : do { ... } while(i <= 5) exécute le code une fois puis vérifie la condition.

Les options de la boucle for

- L'image mentionne des mots-clés spécifiques qui modifient le comportement de la boucle for :
- **step** : Permet de définir un pas d'incrémentation différent de 1.
  - **Exemple** : for(i in 1..10 step 2)
- **downTo** : Permet de faire une décrémentation au lieu d'une incrémentation.
  - **Exemple** : for(i in 10 downTo 1) parcourt les nombres de 10 à 1.

Mots-clés break et continue

- **break** : Permet de sortir immédiatement d'une boucle.
- **continue** : Permet de passer directement à l'itération suivante de la boucle.

RETOUR DE FONCTION : RETURN/UNIT

- **Utilisez return et spécifiez un type de retour** si votre fonction doit renvoyer une valeur (Int, String, etc.).
- **Ne mettez rien (et le type de retour sera Unit)** si votre fonction exécute simplement une action sans rien retourner.

Fonctions avec paramètres par défaut

En Kotlin, vous pouvez définir une **valeur par défaut** pour un paramètre de fonction.

```
fun afficherEntier(A: Int = 100) {
    println("Entier = $A")
}
fun main() {
    var B = 5
    // Appel sans paramètre, la valeur par défaut est utilisée
    afficherEntier()  // Affiche "Entier = 100"

    // Appel avec un paramètre, la nouvelle valeur est utilisée
    afficherEntier(B) // Affiche "Entier = 5"
}
```

FONCTION AVEC PARAMÈTRES NOMMÉS

- Même si la fonction est déclarée dans un ordre spécifique (A puis B), vous pouvez inverser l'ordre des paramètres lors de l'appel, du moment que vous les nommez clairement.

```
//fun afficherEntier(A: Int = 100, B: Int) : String{return "A=$A & B=$B"}
fun afficherEntier(A: Int = 100, B: Int):String= "A=$A & B=$B"
fun main() {
   var A1 = 1
   var A2 = 2
// Utilisation de paramètres nommés pour changer l'ordre

   println(afficherEntier(B = A1, A = A2) )// Affiche "A=2 B=1"

// Utilisation de paramètres nommés pour ignorer la valeur par défaut
   println(afficherEntier(B = A2)) // Affiche "A=100 B=2"

}
```

LES FONCTIONS LAMBDA

- Une **fonction lambda** est une fonction anonyme, c'est-à-dire une fonction qui n'a pas de nom.

```
fun main() {
    // Définition de la fonction lambda
    var somme = {A: Int, B: Int -> A + B}

    // Assignation de la lambda à une autre variable
    var total = somme
    var A1 = 1
    var A2 = 2
    // Appel de la lambda via la variable `somme`
    println("$A1 + $A2 = ${somme(A1, A2)}")
    // Appel de la lambda via la variable `total`
    println("$A1 + $A2 = ${total(A1, A2)}")
}
```

Les classes en Kotlin : object et primary constructor

- Pour créer un objet à partir d'une classe, vous devez l'**instancier** en utilisant un **constructeur**.
- Les getter et setter sont généré automatiquement dans le code

```
// Méthode 1 :
class User(var email: String? = null, var
password: String? = null) {
    fun afficherUser() {
        println("email=$emai\n
password=$password")
    }
}
fun main() {
    var u = User()
    u.email = "rssp@ensa.ma"
    u.password = "123456"
    u.afficherUser()
// Dans cet exemple, les membres de
l'objet sont modifiés après l'instanciation
```

```
// Méthode 2 :
class User(var email: String? = null, var
password: String? = null) {
    fun afficherUser() {
        println("email=$email
\npassword=$password")
    }
}
fun main() {
    var u = User("rssp@ensa.ma", "123456")
    u.afficherUser()
}
// Dans cet exemple, les membres de l'objet
sont initialisés directement dans le
constructeur.
```

DATA CLASS

- C'est une classe conçue pour contenir des données. Sa particularité est qu'elle génère automatiquement des fonctions très pratiques pour vous, vous évitant de les coder manuellement.
- **equals() (ou ==)** : Elle compare les objets en se basant sur les **valeurs de leurs propriétés**
- **toString()** : Elle fournit une représentation textuelle claire de l'objet.
- **copy()** : Elle crée une copie de l'objet.

```
data class User(val email: String, val password: String)
fun main() {
    val ali = User("ali@exemple.com", "123456")
    // Utilisation de toString() et equals()
    println(ali) // Affiche
User(email=ali@exemple.com, password=123456)
    val ali2 = User("ali@exemple.com", "123456")
    println(ali == ali2) // Affiche true, car les valeurs
sont les mêmes
    // Utilisation de copy()
    val ahmed = ali.copy(email =
"ahmed@exemple.com")
    println(ahmed) // Affiche
User(email=ahmed@exemple.com,
password=123456)
    // Décomposition d'objet
    val (userEmail, userPassword) = ali
    println("Email: $userEmail, Mot de passe:
$userPassword")
}
```

Les classes d'énumération (enum class)

- Une **enum class** est une classe spéciale qui vous permet de définir une liste de constantes nommées. C'est très utile pour représenter un ensemble fixe de valeurs, comme les mois de l'année, les jours de la semaine ou les couleurs.

```
enum class Mois {
    JANVIER, FEVRIER, MARS, ...
}
enum class Mois(var IdMois: Int) {
    JANVIER(1), FEVRIER(2), ...
}
```

- On peut accéder à la constante (Mois.JANVIER), mais aussi à sa valeur associée (Mois.JANVIER.IdMois), qui retournera 1 dans cet exemple.

LES TABLEAUX : *ARRAY*

- Un tableau est une classe generic : **Array<Type>(size){value}**
- Pour accéder à une case dans le tableau on utilise **[]**
- La taille d'un tableau T est **T.size**
- Plusieurs méthodes pour initialiser un tableau : **arrayOf()**
- Un tableau peut contenir des éléments de différents types!!

```
fun main(){
    var myTable = Array(4){5}     //Partout la même valeur 5
    myTable[1] = 10
    for (i in myTable) println(i)  // 5 10 5 5
}
fun main(){
    var myTable = arrayOf(1,2,3,4) //pas besoin de déclarer la taille du
tableau
    myTable[1] = 10
    for (i in myTable) println(i) // 1 10 3 4
}
```

LES LISTES: ARRAYLIST

- La principale distinction avec les tableaux est que la taille d'un tableau doit être connue lors de la compilation, alors que la taille d'une liste est flexible.
- Il est possible d'initialiser une liste en utilisant la méthode **arrayListof**().
- L'index d'un élément dans une liste commence toujours à **0**.
- Acces à un element avec la méthode get() ou Utilisation de l'opérateur [] (accès par index)

Fonctions de Liste

- **add**(element): Permet d'ajouter un élément à la fin de la liste.
- **add**(index, element): Permet d'insérer un élément à un emplacement spécifique (index) dans la liste.
- **remove**(element): Supprime la première occurrence d'un élément dans la liste.
- **removeAt**(i): Supprime l'élément à l'index i.

```
fun main(){
   var myList= arrayListOf(1,1.0,"RSSP") // une liste avec les valeurs initiales 1, 1.0 et
"RSSP"
   myList.add(1,100)          // Insère l'entier 100 à l'index 1
   myList.remove(1.0)         // Supprime la première occurrence de la valeur 1.0
   myList.removeAt(2)         // Supprime l'élément qui se trouve maintenant à l'index 2
   for (i in myList) println(i)        // Affiche chaque élément de la liste
//ou bien list.forEach { println(it)} // Affiche chaque element un par un
}
```

LISTES MODIFIABLES (MUTABLELISTS)

- En Kotlin, une **MutableList** est une liste dont le contenu peut être modifié après sa création.
- C'est une **interface**, ce qui signifie qu'elle définit le comportement d'une liste modifiable (ajout, suppression, modification d'éléments).
- La manière la plus simple de créer une MutableList est d'utiliser **mutableListOf()**.

** **mutableListOf() vs arrayListOf()**

- mutableListOf() est considéré comme une meilleure pratique car vous vous concentrez sur l'**intention** (MutableList : "une liste qui peut être modifiée") plutôt que sur l'**implémentation** spécifique (ArrayList).
- Les deux sont fonctionnellement identiques pour la plupart des usages, car ArrayList implémente MutableList.
- mutableListOf() → à privilégier en Kotlin.
- arrayListOf() → utile si tu travailles avec du code Java.
- toMutableList() → utile si tu dois modifier une liste immuable (listOf) sans toucher à l'original.

HASHMAP

- Liste dont les éléments sont des paires de données, composées d'une **clé (key)** et d'une **valeur (value)**.

Fonctions de base

- **put(key, value)** : Cette fonction est utilisée pour ajouter une nouvelle paire clé-valeur. Si la clé existe déjà, la valeur associée est mise à jour.
- **get(key)** : Permet de récupérer la valeur qui correspond à une clé donnée.

```
fun main(){
    // Crée une HashMap qui peut stocker n'importe quel type de clé et de valeur
    var myHashMap=HashMap<Any, Any>()
    // Ajoute plusieurs paires clé-valeur
    myHashMap.put(1, "15")
    myHashMap.put("email", "RSSP@ensa.ma")
    myHashMap.put("Taille", 32)
    // Affiche chaque paire clé-valeur. La sortie sera dans le format "clé=valeur"
    for (i in myHashMap) println(i)
    // Affiche la valeur de chaque clé. Cela parcourt toutes les clés et récupère la valeur correspondante.
    for (i in myHashMap.keys) println(myHashMap.get(i))
    // Affiche directement la valeur associée à la clé "email"
    println(myHashMap.get("email"))
}
```

HÉRITAGE ET POLYMORPHISME

- L'héritage permet à une classe d'étendre une autre. La redéfinition permet à la classe enfant de remplacer l'implémentation d'une fonction de sa classe parente.
- La règle clé est que les classes et fonctions doivent être déclarées open pour pouvoir être héritées ou redéfinies.
- Le mot-clé open est uniquement requis si vous souhaitez **modifier le comportement par défaut de la propriété** (en surchargeant ses accesseurs) ou d'une méthode.
- Le mot-clé **super** vous permet d'accéder aux membres de la classe parente (propriétés et méthodes).
- **Super** permet d'étendre la fonctionnalité d'une méthode sans la réécrire entièrement.

```
Classe de base : Forme
open class Forme(open val nom: String) {
    // Méthode ouverte à la surcharge
    open fun calculerAire(): Double = 0.0
}
//Classes dérivées : Cercle et Carre
class Cercle(val rayon: Double) : Forme("Cercle") {
    override fun calculerAire(): Double = PI * rayon *
rayon
}
class Carre(val cote: Double) : Forme("Carré") {
    override fun calculerAire(): Double = cote * cote
}
fun main() {
    val formes = mutableListOf<Forme>()
    formes.add(Cercle(5.0))
    formes.add(Carre(4.0))
  // Appel de la bonne méthode pour chaque objet:
polymorphisme
    formes.forEach {
        println("Aire du ${it.nom} : ${it.calculerAire()}")
    }}
Résultat :Aire du Cercle : 78.53981633974483
Aire du Carré : 16.0
```

LES CLASSES ABSTRAITES

- Une **classe abstraite** est une classe qui ne peut pas être instanciée directement. Peut contenir des membres (fonctions et variables) qui ne sont pas encore implémentés.
- **Non-instanciable** : Vous ne pouvez pas créer un objet à partir d'une classe abstraite. Elle doit d'abord être héritée par une autre classe.
- **open par défaut** : Contrairement aux classes normales, une classe abstraite est déjà "ouverte" à l'héritage, vous n'avez donc pas besoin d'utiliser le mot-clé open.
- **Implémentation obligatoire** : Toute classe qui hérite d'une classe abstraite **doit obligatoirement** fournir une implémentation pour toutes les fonctions et variables déclarées comme abstract dans la classe mère.

```
// 1. Déclaration de la classe abstraite
abstract class Forme {
    // Une fonction abstraite (sans implémentation). Les classes enfants devront la redéfinir.
    abstract fun calculerSurface(): Double
    // Une fonction concrète qui peut être utilisée directement.
    fun afficherType() {
        println("Ceci est une forme.")
    }
}
// 2. Déclaration d'une classe concrète qui hérite de la classe abstraite
class Cercle(val rayon: Double) : Forme() {
// 3. Implémentation obligatoire de la fonction abstraite
    override fun calculerSurface(): Double {
        return Math.PI * rayon * rayon
    }
}
fun main() {
// val maForme = Forme() // ERREUR de compilation !
val monCercle = Cercle(rayon = 5.0)
monCercle.afficherType() // Affiche : Ceci est une forme.
val surface = monCercle.calculerSurface()
println("La surface du cercle est : $surface") // Affiche : La surface du cercle est : 78.5398
```

INTERFACE

- une **interface** en Kotlin est un contrat qui définit un ensemble de fonctions et de propriétés qu'une classe doit implémenter. Contrairement aux classes, une interface ne peut pas être instanciée, mais elle peut être implémentée par plusieurs classes.

```
interface NameOfInterface { interfaceBody }
```

DÉCOUVERTE D'ANDROID STUDIO

APPLICATION ANDROID NATIVE

- Une application Android native se compose de :
- **Sources Kotlin ou Java :** Ces fichiers sont compilés en bytecode pour la machine virtuelle Dalvik.
- **Ressources :** Ce sont des fichiers de différents formats, comme :
  - **Format XML :** pour l'interface utilisateur et les textes.
  - **Format Image :** pour les icônes, images, etc. (PNG, GIF).
  - **Format Audio :** pour les fichiers MP3.
- **Manifeste (AndroidManifest.xml) :** Ce fichier décrit le contenu de l'application. Il inclut :
  - La version minimale de l'appareil compatible.
  - La signature des fichiers d'archive.
  - Les autorisations requises par l'application.

L'ensemble du projet est géré à l'aide d'un environnement de développement intégré (IDE) appelé **Android Studio**, qui utilise le kit de développement logiciel **Android SDK** (Software Development Toolkit).

SDK & ANDROID STUDIO

- Le **Software Development Kit (SDK)** est un ensemble d'outils essentiels pour le développement d'applications. Il contient :
  - Des librairies de classes et de fonctions pour créer des logiciels.
  - Des outils de fabrication de logiciels, comme des compilateurs.
  - **AVD (Android Virtual Device) :** un émulateur pour tester les applications sur différentes tablettes virtuelles.
  - **ADB (Android Debug Bridge) :** un outil pour communiquer avec de vrais appareils Android pour le débogage.
- **Android Studio** est l'environnement de développement principal/
  - Un éditeur de sources et de ressources.
  - Des outils de compilation, comme **Gradle**.
  - Des outils de test et de débogage pour mettre au point les applications.
- Pour commencer, il suffit d'installer Android Studio. Le SDK peut être installé automatiquement avec l'IDE ou via une installation personnalisée.

CHOIX DE LA VERSION

Chaque version d'Android, dénotée par son *API level*, ex: 25, apporte des améliorations et supprime des dispositifs obsolètes.

Toute application exige un certain niveau d'API :

- **minSdkVersion** doit être inférieur pour cibler la couverture maximale des appareils Android sur lesquels l'application sera installée
- **targetSkdVersion**: est la dernière version du système d'exploitation Android sur laquelle vous souhaitez exécuter votre application pour obtenir une optimisation complète des ressources Android. l'application sera testée et marchera correctement jusqu'à ce niveau d'API,
- **compileSdkVersion** : c'est le niveau maximal de fonctionnalités API qu'on se limite à employer. Si on fait appel à quelque chose de plus récent que ce niveau, le logiciel ne se

```
minSdkVersion <= targetSdkVersion <= compileSdkVersion
```

STRUCTURE DU PROJET ANDROID

```
MyProject/
├── gradle/
│   ├── wrapper/
│   │   ├── gradle-wrapper.jar
│   │   └── gradle-wrapper.properties
│   └── (other files)
├── app/
│   ├── build.gradle(.kts)  // module-level build.gradle
│   └── src/
│       └── main/
│           ├── java/
│           ├── res/
│           └── AndroidManifest.xml
├── build.gradle(.kts)  // project-level build.gradle
├── settings.gradle(.kts) // project settings file
└── (other files)
```

Le dossier app est le cœur de votre application. C'est ici que se trouvent tout votre code, vos ressources et les configurations spécifiques à votre application.

```
app/
├── build/                    (Dossier de sortie pour les fichiers générés
├── libs/                     (Bibliothèques locales)
├── src/                      (Contient tout votre code source et vos ressources)
│   ├── main/                 (Source de l'application principale)
│   │   ├── java/             (ou kotlin/ si le langage est le Kotlin)
│   │   │   └── com.example.monprojet/  (Vos packages et vos classes)
│   │   │       └── MainActivity.kt
│   │   ├── res/              (Ressources de l'application)
│   │   │   ├── drawable/     (Images et icônes)
│   │   │   ├── layout/       (Fichiers XML pour l'interface utilisateur)
│   │   │   ├── mipmap/       (Icônes de l'application)
│   │   │   └── values/       (Strings, couleurs, styles, etc.)
│   │   └── AndroidManifest.xml   (Fichier manifest de l'application)
│   ├── androidTest/          (Tests d'instrumentation)
│   └── test/                 (Tests unitaires)
└── build.gradle.kts          (Configurations de compilation du module d'application
```

GRADLE

- Le système de compilation d'Android Studio est basé sur Gradle, un outil flexible d'automatisation de la compilation. Dans le contexte d'Android, Gradle compile le code, gère les dépendances et emballe l'application dans un fichier APK
- Lors du démarrage d'un projet ou de la modification d'une dépendance, Android Studio exécute une synchronisation Gradle. Cette action lit les fichiers .gradle et télécharge les dépendances nécessaires.
- Lorsqu'on clique sur le bouton "Run", Gradle lance une série de tâches qui incluent la compilation du code, l'assemblage des ressources et la génération du fichier APK.
- La flexibilité de Gradle permet d'effectuer des tâches complexes et personnalisées, comme la création de différentes versions d'une application ("flavors") ou l'automatisation de tests.

GRADLE FICHIERS...

Les fichiers de configuration fondamentaux de Gradle, ont un rôle précis dans la gestion du processus de construction de votre application.

- **settings.gradle / settings.gradle.kts** : point d'entrée de votre projet. Il définit la structure de votre projet et inclut les modules (par exemple, le module app) qui font partie de la compilation. C'est ici que vous indiquez à Gradle où trouver les différentes parties de votre projet.
- **build.gradle / build.gradle.kts (à la racine du projet)** : gère les configurations qui s'appliquent à tous les modules de votre projet. Définit les versions de plugins partagées, les dépôts de dépendances (comme Maven Central ou Google) et d'autres paramètres globaux.
- **build.gradle / build.gradle.kts (au niveau du module)** : spécifique à un seul module (comme votre application). Il contient les configurations uniques à ce module, telles que la version du SDK cible, l'identifiant de l'application, et surtout, la liste des dépendances spécifiques requises par ce module pour fonctionner.
- **gradle-wrapper.properties** : assure que tout le monde travaillant sur le projet utilise la même version de Gradle. Il contient l'URL de la distribution Gradle à télécharger et garantit ainsi un environnement de compilation cohérent, peu importe l'ordinateur ou le développeur.

STRUCTURE D'UN PROJET ANDROID : LE MANIFEST

- Fichier XML
- Précise l'architecture de l'application
- Chaque application doit en avoir un
- AndroidManifest.xml est dans la racine du projet
- Précise le nom du package java utilisant l'application cela sert d'identifiant unique !
- Décrit les composants de l'application
  - Liste des activités…
  - Précise les classes qui les implémentent
  - …
- Définit les permissions de l'application
  - Droit de passer des appels
  - Droit d'accéder à Internet
  - Droit d'accéder au GPS

STRUCTURE D'UN PROJET ANDROID : LE MANIFEST

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

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.MyApplication"
        tools:targetApi="34">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

EDITEUR SPÉCIFIQUE

- Les **ressources** (comme la disposition des vues, les menus, les images, les textes) sont définies à l'aide de **fichiers XML**.
- Android Studio propose des **éditeurs spécialisés** pour ces fichiers :
  - Des **formulaires** pour éditer les textes de l'interface, stockés dans le fichier **res/values/strings.xml**.
  - Des **éditeurs graphiques** pour la disposition des éléments sur l'interface, définis dans les fichiers **res/layout/*.xml**.

RECONSTRUCTION DU PROJET

Chaque modification d'un source ou d'une ressource fait **reconstruire** le projet. C'est automatique.

Dans certains cas (travail avec un gestionnaire de sources comme Subversion ou Git), il peut être nécessaire de reconstruire manuellement. Il suffit de sélectionner le menu Build/Rebuild Project.

En cas de confusion d'Android Studio (compilation directe en ligne de commande), ou de mauvaise mise à jour des sources (partages réseau), il faut parfois nettoyer le projet. Sélectionner le menu Build/Clean Project.

Ces actions lancent l'exécution de *Gradle*.

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 / WIFI
- Simuler l'application sur une tablette virtuelle *AVD* (Android Virtual Device)

Communication Android Studio

FENÊTRES ANDROID

Android Studio affiche plusieurs fenêtres utiles indiquées dans l'onglet tout en bas :

Logcat Affiche tous les messages émis par la tablette courante

Messages Messages du compilateur et du studio

Terminal Shell unix permettant de lancer des commandes dans le dossier du projet.

FENÊTRE LOGCAT

Des messages détaillés sont affichés dans la fenêtre LogCat:

- Ils sont émis par les applications : debug, infos, erreurs. . .

Pour afficher des messages dans le logcat de façon structurée.Il est commode de définir des filtres pour ne pas voir la totalité des messages de toutes les applications de la tablette :

- sur le niveau de gravité : verbose, debug, info, warn, error et assert,
- sur l'étiquette TAG associée à chaque message,
- sur le package de l'application qui émet le message

EMISSION D'UN MESSAGE VERS LOGCAT

Méthodes statiques de la classe Log

- Log.v(….), Log.d(….), Log.i(….), Log.e(…..), Log.w(…..)
- *Paramètres*: « tag », « message »

Fonctions Log.*:

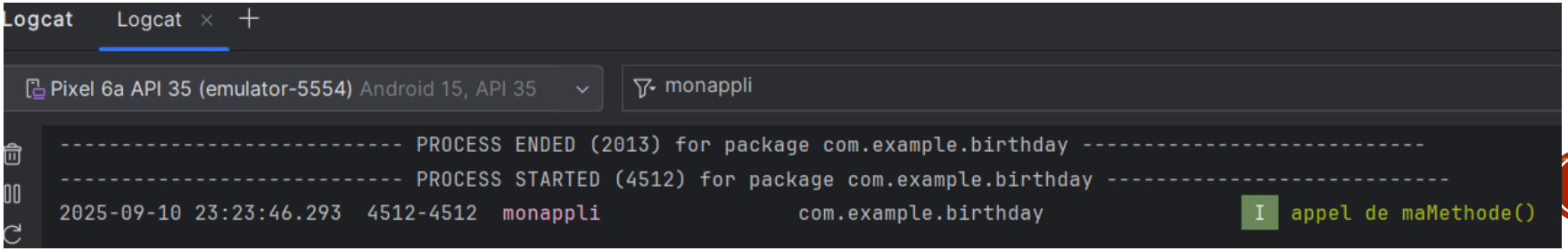
Log.i(String tag, String message) affiche une info,

Log.w(String tag, String message)affiche une alerte,

Log.e(String tag, String message)affiche une erreur.

Log.d(String tag, String message)affiche debug.

```
private val TAG = "monappli"
override fun onCreate(savedInstanceState: Bundle?) {
   super.onCreate(savedInstanceState)
....
   Log.i(TAG, "appel de maMethode()")
....
```



Création d’un paquet installable

CRÉATION D'UN PAQUET INSTALLABLE (.APK)

•Un fichier **.apk** est une archive signée et installable.

•Pour le créer, utilisez le menu **Build > Generate Signed Bundle/APK**.

•La distribution d'une application nécessite une **clé privée** stockée dans un **keystore** (fichier crypté).

•Lors de la première création, vous définissez un **alias** et un **mot de passe** pour votre clé.

•Ces mots de passe sont demandés à chaque fois que vous créez un .apk.

REFERENCES

- Développement Android en Kotlin
- Principes de base du Kotlin
- Fonctions
- Classes et objets
- Créer votre première application Android