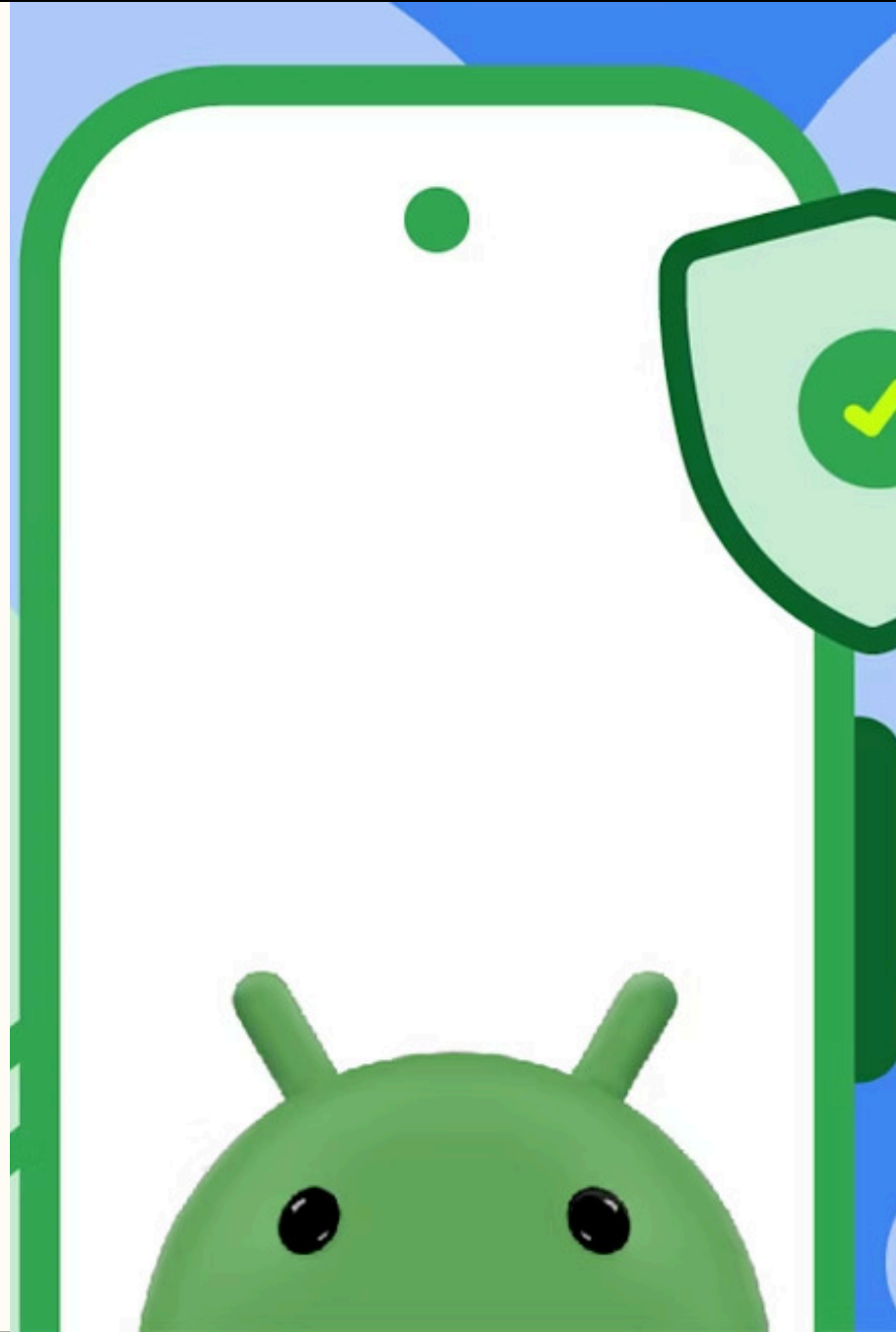
ECOLE MAROCAINE DES SCIENCES DE L'INGENIEUR

Membre de HONORIS UNITED UNIVERSITIES

Chapitre 2 : Application Android

Par: I. AITOUHANNI,S.E. BENTALBA, M. BELATAR



Chapitre 2 : Application Android

Ce chapitre est consacré au développement d'applications Android. Nous allons explorer ensemble les concepts essentiels qui vous permettront de créer, comprendre et maintenir des applications Android professionnelles. Ce chapitre vous guidera pas à pas, de la structure d'un projet jusqu'aux aspects avancés de la gestion du cycle de vie.

Objectifs du chapitre

Structure de projet

Comprendre l'organisation complète d'un projet Android et ses différents composants essentiels

Composantes principales

Identifier et maîtriser les éléments clés qui constituent une application Android moderne

Création et exécution

Apprendre à créer, exécuter et déboguer efficacement vos applications Android

Cycle de vie

Maîtriser le cycle de vie d'une activité pour gérer correctement les ressources et l'état de l'application

Structure d'un projet Android

Un projet Android est organisé selon une structure bien définie qui facilite le développement et la maintenance. Chaque dossier et fichier a un rôle spécifique dans l'architecture globale de votre application.

Cette organisation standardisée permet aux développeurs de naviguer facilement dans n'importe quel projet Android et de comprendre rapidement son fonctionnement.

01

AndroidManifest.xml

Fichier de configuration principal de l'application

02

Code Java (ou Kotlin)

Code source de votre application (/src/main/java)

03

Ressources

Layouts, images, strings, couleurs (/res)

04

Scripts Gradle

Fichiers de configuration du build et dépendances

Les ressources Android (/res)

Le dossier `/res` contient toutes les ressources non-code de votre application. Android organise ces ressources par type dans des sous-dossiers spécifiques, permettant une gestion optimisée et une adaptation automatique selon l'appareil.

/res/layout

Fichiers XML définissant les interfaces utilisateur de vos activités et fragments

/res/drawable

Images, icônes et formes vectorielles utilisées dans l'interface

/res/values

Strings, couleurs, dimensions et styles réutilisables dans toute l'application

/res/mipmap

Icônes de l'application dans différentes résolutions pour le launcher

/res/menu

Définitions des menus de l'application (barre d'action, menus contextuels)

/res/raw

Fichiers bruts (audio, vidéo, données) accessibles via les ressources

Android sélectionne automatiquement les bonnes ressources selon la configuration de l'appareil (langue, densité d'écran, orientation), simplifiant ainsi l'adaptation de votre application à une multitude de dispositifs.

Accès aux ressources Android

Android génère automatiquement une classe `R` qui contient des références à toutes vos ressources. Cette classe permet d'accéder facilement aux ressources depuis le code.

Depuis le code Java - Classe R

La classe R est générée automatiquement et organisée par type de ressource :

```
// Accès aux layouts
setContentView(R.layout.activity_main);

// Accès aux strings
String titre = getString(R.string.app_name);
TextView text = findViewById(R.id.textView);
text.setText(R.string.welcome_message);

// Accès aux images
ImageView image = findViewById(R.id.imageView);
image.setImageResource(R.drawable.logo);

// Accès aux couleurs
int couleur = ContextCompat.getColor(this, R.color.primary_blue);
```

Depuis les fichiers XML - Syntaxe @

Dans les fichiers XML, utilisez la syntaxe @type_ressource/nom_ressource :

```
<TextView
    android:text="@string/welcome_message"
    android:textColor="@color/primary_blue"
    android:background="@drawable/button_background"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content" />

<ImageView
    android:src="@drawable/logo"
    android:contentDescription="@string/logo_description" />
```

La classe `R` est générée automatiquement à chaque build et ne doit jamais être modifiée manuellement. Toute modification serait écrasée lors de la prochaine compilation de votre projet.

Détail du fichier AndroidManifest.xml

Le fichier `AndroidManifest.xml` est le cœur de votre application Android. Il contient toutes les métadonnées essentielles que le système Android doit connaître avant d'exécuter votre application. Ce fichier déclare le nom du package, les activités, les services nécessaires, les permissions requises et les intent filters qui définissent comment votre application interagit avec le système.

Package

Identifiant unique de l'application sur le Play Store

Activités & Services

Déclaration de tous les composants de l'application

Permissions

Droits d'accès nécessaires (caméra, réseau, etc.)

Intent Filters

Définit comment l'application répond aux intentions

Exemple de manifeste

```
<manifest package="com.emsi.myapp">
 <application android:label="MyApp">
 <activity android:name=".MainActivity">
 <intent-filter>
 <action android:name="android.intent.action.MAIN" />
 <category android:name="android.intent.category.LAUNCHER" />
 </intent-filter>
 </activity>
 </application>
</manifest>
```

Les principales composantes d'une application

Une application Android repose sur quatre types de composants fondamentaux. Chacun joue un rôle spécifique dans l'architecture globale et permet de construire des applications riches et interactives.

Activity

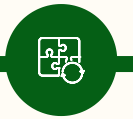
Représente un écran avec une interface utilisateur. C'est le composant visible avec lequel l'utilisateur interagit directement. Une application contient généralement plusieurs activités.

Service

Exécute des tâches en arrière-plan sans interface utilisateur. Parfait pour la lecture de musique, le téléchargement de fichiers ou les opérations longues.

Broadcast Receiver

Écoute et répond aux événements système comme les changements de batterie, de connectivité réseau ou les messages SMS reçus.



Content Provider

Gère et partage les données entre applications de manière sécurisée. Permet l'accès aux contacts, calendrier et autres données partagées.

Exemple – Une simple Activity Java

```
public class MainActivity extends AppCompatActivity {

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

Cette classe `MainActivity` hérite de `AppCompatActivity`, la classe de base pour toutes les activités modernes Android. La méthode `onCreate()` est le point d'entrée : elle initialise l'activité et définit son interface via `setContentView()`.

> 💡 **Point clé**
>
> Chaque Activity gère un écran unique de l'application. C'est le composant fondamental de l'interface utilisateur en Android.

Layout XML associé

Le fichier XML de layout définit l'apparence visuelle de votre activité. Il utilise une hiérarchie de vues pour organiser les éléments à l'écran. Android sépare ainsi la logique (Java) de la présentation (XML), facilitant la maintenance et la modification de l'interface.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  android:orientation="vertical"
  android:padding="16dp"
  android:layout_width="match_parent"
  android:layout_height="match_parent">

  <TextView
    android:text="Hello Android!"
    android:textSize="24sp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"/>

</LinearLayout>
```

LinearLayout

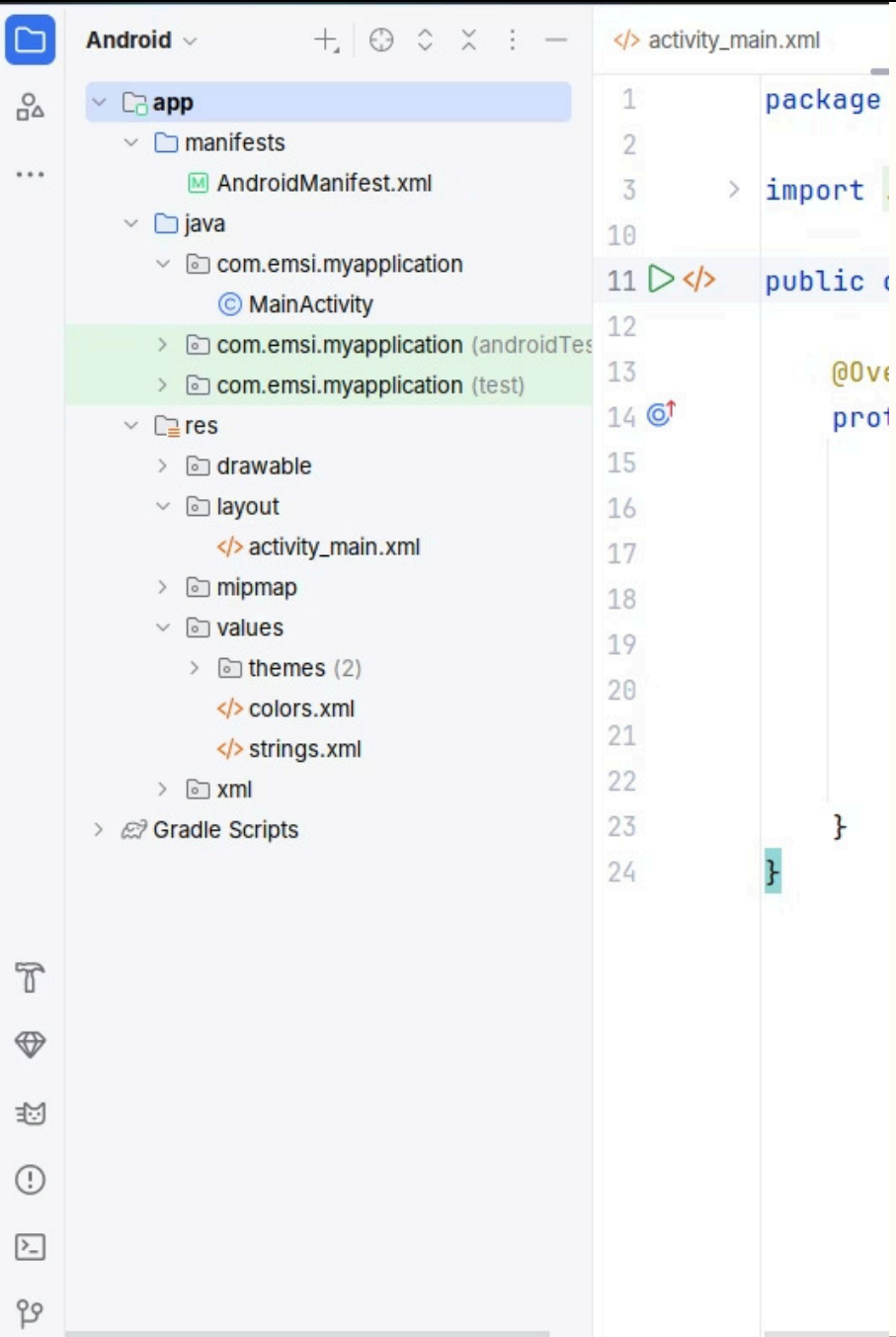
Conteneur qui organise les vues enfants de manière linéaire (verticale ou horizontale)

TextView

Composant d'affichage de texte, l'un des widgets les plus utilisés en Android

Attributs de layout

Définissent les dimensions, marges et comportement des vues à l'écran



Lien entre Activity et Layout

MainActivity.java

Contient la logique et le comportement de l'écran

setContentView()

Méthode qui établit la connexion entre code et interface

activity_main.xml

Définit l'apparence visuelle de l'écran

Le lien entre l'Activity Java et le layout XML se fait via la méthode `setContentView()` appelée dans `onCreate()`. Cette méthode prend en paramètre une référence au fichier XML : `R.layout.activity_main`. La classe `R` est générée automatiquement par Android et contient des références à toutes vos ressources. Ce système permet de modifier l'interface sans recompiler tout le code Java.

Création d'un projet Android (pas à pas)

Android Studio simplifie grandement la création d'un nouveau projet grâce à son assistant intégré. En quelques clics, vous obtenez une structure de projet complète et fonctionnelle, prête pour le développement.

Lancer Android Studio

Cliquez sur **New Project** dans l'écran d'accueil ou via le menu File

Choisir le template

Sélectionnez **Empty Views Activity** pour démarrer avec une structure minimale et claire

Configurer le projet

Saisissez le nom de l'application, le package, l'emplacement et choisissez Java comme langage

Niveau API minimum

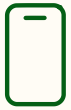
Sélectionnez la version Android minimale supportée (API 24+ recommandé pour couvrir 98% des appareils)

Finaliser

Cliquez sur **Finish** : Android Studio génère automatiquement toute la structure du projet

Android Virtual Device (AVD)

L'Android Virtual Device (AVD) est un émulateur complet qui simule un appareil Android réel directement sur votre ordinateur. Il permet de tester votre application sans avoir besoin d'un smartphone physique, ce qui est particulièrement utile pour tester différentes configurations matérielles et versions d'Android.



Simulation complète

L'AVD reproduit fidèlement le comportement d'un vrai appareil Android, incluant capteurs, GPS, caméra et connectivité réseau. Vous pouvez même simuler des appels et SMS.

Hautement configurable

Personnalisez entièrement votre émulateur : choisissez la version Android (de KitKat à la dernière), la quantité de RAM, la taille et résolution d'écran, et même l'architecture du processeur.

Tests multi-appareils

Créez plusieurs AVD pour tester votre application sur différents profils : smartphone compact, tablette, TV Android ou téléphone pliable, sans investir dans du matériel coûteux.

Debuggage Android

Utilisation de Logcat

Logcat est l'outil de débogage essentiel d'Android Studio. Il affiche en temps réel tous les messages système et applicatifs, vous permettant de suivre l'exécution de votre code et d'identifier rapidement les problèmes.

Ajouter des logs dans votre code

```
Log.d("DEBUG", "onCreate exécuté");
Log.i("INFO", "Utilisateur connecté");
Log.w("WARNING", "Batterie faible");
Log.e("ERROR", "Connexion échouée");
```

Les logs sont classés par niveau de sévérité : Debug, Info, Warning, Error et Verbose. Utilisez des tags clairs pour filtrer facilement vos messages.

Points d'arrêt (Breakpoints)

Android Studio supporte le débogage avancé avec des breakpoints. Cliquez dans la marge gauche de l'éditeur pour en placer un. L'exécution s'arrêtera à cette ligne, vous permettant d'inspecter les variables et l'état de votre application.

- **Inspecter les variables**

  Visualisez les valeurs en temps réel

- **Exécution pas à pas**

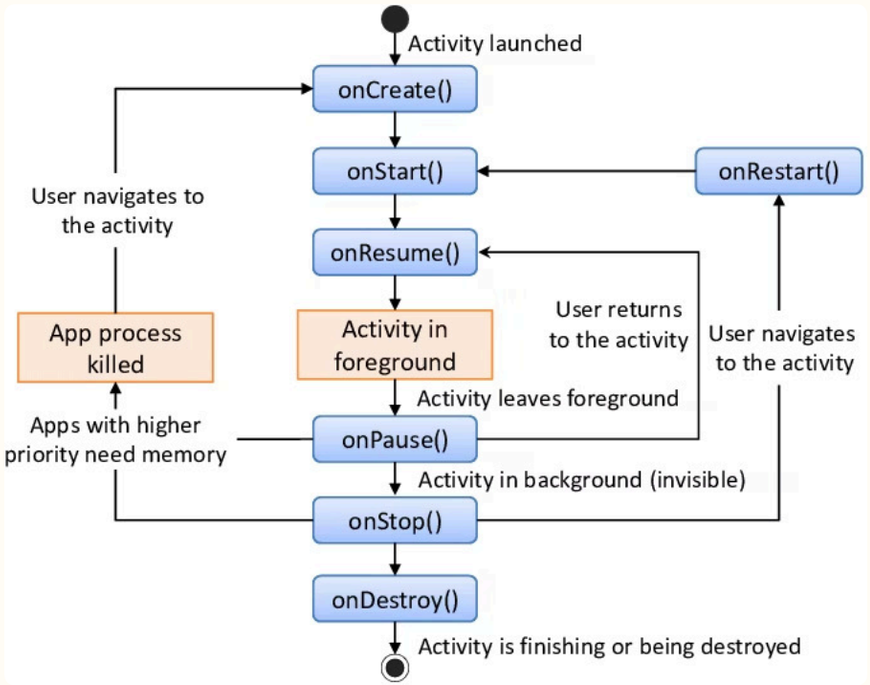
  Avancez ligne par ligne dans votre code

- **Évaluer des expressions**

  Testez du code à la volée pendant le debug

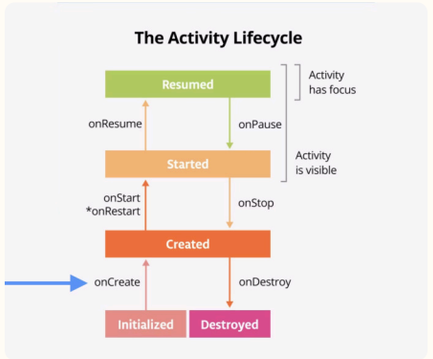
Cycle de vie d'une activité

Ce diagramme illustre le flux complet du cycle de vie d'une Activity. Les flèches indiquent les transitions possibles entre chaque état. Notez que certaines transitions peuvent être déclenchées par l'utilisateur (clic sur le bouton Retour) ou par le système (mémoire faible, rotation d'écran). Chaque méthode du cycle de vie est appelée dans un ordre prévisible, vous permettant de gérer correctement les ressources et l'état de votre application.



Les transitions principales sont :

- **onStop → onCreate :** L'activité est complètement détruite par le système (mémoire faible) puis recréée.
- **onStop → onRestart :** L'utilisateur revient à l'activité qui était en arrière-plan.
- **onPause → onResume :** Retour rapide à l'activité (fermeture d'un dialogue, notification).
- **onPause → onCreate :** Destruction rapide de l'activité puis recréation (rotation d'écran).



Transitions du cycle de vie - Cas pratiques

Comprendre quand et pourquoi Android passe d'un état à l'autre est crucial pour anticiper le comportement de votre application.

onStop → onCreate (Destruction complète)

Quand : Le système manque de mémoire ou l'utilisateur n'utilise pas l'app depuis longtemps

Exemple : Vous ouvrez 10 applications lourdes, Android détruit les plus anciennes pour libérer la RAM

Ce qui se passe : L'activité est complètement supprimée de la mémoire, puis recréée from scratch

onStop → onRestart → onStart → onResume (Retour normal)

Quand : L'utilisateur revient à une activité qui était simplement en arrière-plan

Exemple : Vous passez à WhatsApp puis revenez à votre app via le bouton multitâche

Ce qui se passe : L'activité était préservée en mémoire, elle redémarre rapidement

onPause → onResume (Interruption légère)

Quand : Une interruption temporaire sans quitter complètement l'activité

Exemple : Notification qui s'affiche, dialogue système, appel entrant qui se termine rapidement

Ce qui se passe : L'activité reste visible partiellement, reprend immédiatement le focus

onPause → onCreate (Destruction rapide)

Quand : Changement de configuration ou contrainte système immédiate

Exemple : Rotation de l'écran, changement de langue, mémoire critique

Ce qui se passe : Android détruit et recrée l'activité pour s'adapter à la nouvelle configuration

Exemple de code complet du cycle de vie

Voici une implémentation complète montrant toutes les méthodes du cycle de vie avec des logs pour visualiser l'ordre d'exécution. En exécutant ce code et en observant Logcat, vous comprendrez concrètement quand chaque méthode est appelée.

Création et démarrage

```
  @Override
  protected void onCreate(Bundle
savedInstance) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.d("Cycle", "onCreate");
  }

  @Override
  protected void onStart() {
    super.onStart();
    Log.d("Cycle", "onStart");
  }

  @Override
  protected void onResume() {
    super.onResume();
    Log.d("Cycle", "onResume");
  }
```

Pause et arrêt

```
@Override
protected void onPause() {
  super.onPause();
  Log.d("Cycle", "onPause");
}

@Override
protected void onStop() {
  super.onStop();
  Log.d("Cycle", "onStop");
}

@Override
protected void onRestart() {
  super.onRestart();
  Log.d("Cycle", "onRestart");
}
```

Destruction

```
  @Override
  protected void onDestroy() {
    super.onDestroy();
    Log.d("Cycle", "onDestroy");
  }
}
```

💡 **Exercice pratique :** Copiez ce code, exécutez-le et observez Logcat pendant que vous naviguez : passez à une autre app puis revenez (pour voir onRestart), ou faites pivoter l'écran. Vous verrez l'ordre exact des appels.

Pourquoi comprendre le cycle de vie ?

Maîtriser le cycle de vie des activités n'est pas optionnel : c'est une compétence essentielle pour créer des applications Android robustes et efficaces. Une mauvaise gestion du cycle de vie entraîne des bugs subtils, des crashs inattendus et une expérience utilisateur dégradée.

Gestion des ressources

Libérez correctement les ressources coûteuses comme la caméra, les capteurs, les connexions réseau ou les fichiers ouverts. Ne pas le faire provoque des fuites de ressources qui ralentissent ou bloquent le système.

Sauvegarde de l'état

Android peut détruire et recréer votre activité à tout moment (rotation d'écran, mémoire faible). Sans sauvegarde appropriée, l'utilisateur perd ses données saisies ou sa progression, créant une expérience frustrante.

Éviter les fuites mémoire

Les références non libérées empêchent le garbage collector de fonctionner correctement. Résultat : l'application consomme de plus en plus de mémoire jusqu'au crash inévitable (OutOfMemoryError).

Sauvegarde de l'état

Android peut détruire votre activité à tout moment pour libérer de la mémoire ou lors d'un changement de configuration (rotation d'écran). Sans mécanisme de sauvegarde, toutes les données temporaires sont perdues. La méthode `onSaveInstanceState()` vous permet de sauvegarder l'état avant destruction.

Exemple de sauvegarde

```
@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);
  outState.putString("message", "Hello");
  outState.putInt("compteur", compteurValue);
  outState.putBoolean("connexion", isConnected);
}
```

Le `Bundle` est une structure clé-valeur qui stocke vos données. Vous pouvez y mettre des types primitifs (int, boolean, String) et des objets Parcelable. Ces données survivent à la destruction de l'activité et sont restaurées automatiquement.

Données temporaires uniquement

Ne sauvegardez que l'état de l'interface, pas des données métier volumineuses

Limitation de taille

Le Bundle a une limite (~500KB), privilégiez les données essentielles

Restauration de l'état

Une fois l'état sauvegardé, vous devez le restaurer pour que l'utilisateur retrouve l'application dans l'état où il l'avait laissée. Deux méthodes permettent cette restauration : `onCreate()` et `onRestoreInstanceState()`.

Méthode 1 : Dans onCreate()

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

  if (savedInstanceState != null) {
    String msg = savedInstanceState.getString("message");
    int compteur = savedInstanceState.getInt("compteur");
    Log.d("Restore", "Message: " + msg);
  }
}
```

Méthode 2 : Dans onRestoreInstanceState()

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

  String msg = savedInstanceState.getString("message");
  int compteur = savedInstanceState.getInt("compteur");
  Log.d("Restore", "Message: " + msg);
}
```

> 💡 **Quelle méthode choisir ?** `onRestoreInstanceState()` est appelée après `onStart()` et uniquement si un état a été sauvegardé, vous n'avez donc pas besoin de vérifier si le Bundle est null. C'est généralement plus propre pour la restauration.

Grâce à ce mécanisme, l'utilisateur ne remarque même pas que l'activité a été détruite et recréée. Son texte saisi, sa position de scroll et toutes les données temporaires sont préservées automatiquement.

Bonnes pratiques Android

Pour conclure ce chapitre, voici les bonnes pratiques essentielles qui distinguent un développeur Android débutant d'un développeur professionnel. Respectez ces principes pour créer des applications performantes, stables et appréciées des utilisateurs.

1

Ne bloquez jamais le thread principal

Le thread UI (main thread) gère l'affichage et les interactions utilisateur. Toute opération longue (réseau, base de données, calculs) doit s'exécuter dans un thread secondaire ou via des coroutines. Bloquer l'UI thread provoque des freezes et l'erreur ANR (Application Not Responding).

2

Libérez les ressources dans onPause() / onStop()

Fermez les connexions, arrêtez les capteurs, relâchez la caméra dès que l'activité n'est plus visible. Ne pas le faire draine la batterie, consomme inutilement des ressources système et peut empêcher d'autres applications de fonctionner correctement.

3

Utilisez Logcat pour traquer les anomalies

Ajoutez des logs pertinents à chaque étape critique de votre code. Logcat est votre meilleur allié pour comprendre le comportement de votre application, détecter les erreurs silencieuses et optimiser les performances. Utilisez des tags cohérents pour faciliter le filtrage.

4

Respectez la hiérarchie du cycle de vie

Appelez toujours `super.onMethode()` en premier dans vos surcharges de méthodes de cycle de vie. Cette pratique garantit que le comportement de base d'Android s'exécute correctement avant votre code personnalisé. Ne pas le faire peut causer des bugs subtils et des crashs difficiles à diagnostiquer.