République Tunisienne
Ministère de
l'Enseignement
Supérieur
et de la Recherche
Scientifique

الجمهورية التونسية
وزارة التعليم العالي
والبحث العلمي

Compte rendu TP Sérialisation

Application de calcul de l'IMG

Auteur
Karim Dhafer

Matière
Développement Mobile sur Android

Classe
L3-DSI3

Année universitaire
2024-2025

Sommaire

Liste des figures

Introduction

Cet atelier se concentre sur l'implémentation de la persistance dans l'application mobile Android ce qui permet de mémoriser les entrées de l'utilisateur afin de faciliter l'utilité de l'application.

Objectifs

- Comprendre le phénomène de persistance.
- Codage de la classe de serialisation.
- Implémentation de la sérialisation dans les autres couches de l'application.
- Comprendre comment Android gère les fichiers internes d'une application comme le fichier de sérilaisation.

1. Comprendre la Persistance des Données sur Android

Dans cette partie, nous allons explorer la problématique de ce phénomène, puis sa notion de base et, enfin, les différentes solutions de persistance que l'on peut implémenter sur Android.

1.1 Introduction à la problématique

Lorsque les applications récupèrent des données à partir des entrées des utilisateurs, ces données restent accessibles uniquement pendant que l'application est en cours d'exécution. Dés que l'application est fermée, toutes les données créées pendant l'exécution de l'application sont perdues. D'où l'importance de gérer les données insérées par les utilisateurs pour assurer une expérience d'utilisation fluide.

1.2 La notion de persistance des données

La persistance des données permet de conserver l'état et les informations de l'application, même après sa fermeture ou le redémarrage du système. Grâce à la persistance, ces données peuvent rester stockées tant que l'application est installée sur le smartphone. Dans ce contexte, la plateforme Android offre différentes solutions pour implémenter la persistance.

1.3 Les méthodes de persistance de données sur Android

Android propose plusieurs possibilités pour persister les données d'une application, voici quelques-unes :

- **Sérialisation des objets :** Permet de convertir des objets en un format binaire ou textuel pour les stocker dans des fichiers, facilitant ainsi leur récupération ultérieure. C'est la méthode que nous allons utiliser dans notre cas.

- **Base de Données SQLite :** Permet de stocker des données structurées comme les contacts sous un système de base de données intégré.
- **Base de données externe :** Permet de stocker des données sur un serveur distant, offrant une gestion centralisée des données, utilisée pour un accès partagé ou à grande échelle.
- **Stockage Partagé (Shared Preferences) :** Permet de sauvegarder des données légères comme les paramètres de l'utilisateur sous forme de paires clé-valeur dans des fichiers XML.
- **Room Database :** Une couche d'abstraction sur SQLite qui simplifie la manipulation de la base de données et assure la sécurité des types.
- **Stockage de Fichiers :** Permet de stocker des fichiers tels que des images, des documents ou d'autres types de données volumineuses. Cela peut inclure :
  - **Stockage Interne :** Stockage réservé à l'application, accessible uniquement par elle-même.
  - **Stockage Externe :** Stockage sur des périphériques accessibles par plusieurs applications, comme un répertoire public ou une carte SD.
- **Stockage sur le Cloud :** Permet de conserver des données à distance pour la synchronisation entre plusieurs appareils.

2. Codage de la classe de serialisation

La sérialisation est un mécanisme permettant de convertir un objet Java en une séquence d'octets afin de le stocker dans des fichiers binaires ou textuels. Nous allons utiliser cette méthode afin de persister des objets et permettre leur récupération ultérieure. Dans cette section, nous détaillons l'implémentation de la classe **Serializer** pour gérer cette sérialisation et désérialisation des données.

2.1 Création de la classe Serializer

Tout d'abord, on ajoute un nouveau package **outils** (ou **utils** en anglais), dont l'interêt est d'organiser et de séparer les fonctionlités utilitaires de la logique métier de l'application. Puis, on ajoute une classe abstraite Java nommée **'Serializer'**.
Cette classe contient principalement 2 méthodes statiques, une pour la mémorisation des objets et l'autre pour la récupération de ces derniers lors de l'exécution de l'application.

```
1  public abstract class Serializer {
2      public static void serialize(String filename, Object object,
           Context context) {...}
3      public static Object deSerialize(String filename, Context
           context) {...}
4  }
```

2.2 La méthode 'serialize()'

2.2.1 Les paramètres de la fonction 'serialize()'

Les paramètres de la méthode `serialize()` sont définis comme suit :

1. **String filename** : Le nom du fichier dans lequel l'objet doit être sérialisé et stocké. Il permet d'identifier le fichier dans le système de fichiers internes de l'application.

2. **Object object** : L'objet à sérialiser. Il doit être une instance d'une classe qui implémente l'interface `Serializable`, permettant ainsi de le convertir en un flux de données.

3. **Context context** : Le contexte de l'application, qui fait référence au contexte de l'application, généralement un Activity ou un Service.

2.2.2 Fonctionnement de la méthode 'serialize()'

Cette méthode permet de mémoriser les données selon la logique suivante :

1. Ouvrir un flux de fichier en mode privé pour l'écriture.

2. Créer un **ObjectOutputStream** pour permettre la sérialisation de l'objet.

3. Sérialiser l'objet et l'écrire dans le fichier à l'aide de la méthode `writeObject(object)`.

4. Vider le flux et fermer l'**ObjectOutputStream** pour s'assurer que toutes les données sont écrites.

5. Gérer les exceptions liées à l'accès aux fichiers ou à la sérialisation.

2.2.3 Le code de la méthode 'serialize()'

```
1   public static void serialize(String filename, Object object,
        Context context) {
2       try {
3           FileOutputStream file = context.openFileOutput(filename,
                Context.MODE_PRIVATE);
4           ObjectOutputStream oos;
5           try {
6               oos = new ObjectOutputStream(file);
7               oos.writeObject(object);
8               oos.flush();
9               oos.close();
10          } catch (IOException e) {
11              e.printStackTrace();
12          }
13      } catch (FileNotFoundException e) {
14          e.printStackTrace();
15      }
16  }
```

2.3 La méthode 'deSerialize()'

2.3.1 Les paramètres de la fonction 'deSerialize()'

Les paramètres de la méthode `deSerialize()` sont définis comme suit :

1. **String filename :** Le nom du fichier à partir duquel l'objet doit être désérialisé. Il permet d'identifier le fichier dans le système de fichiers internes de l'application.
2. **Context context :** Le contexte de l'application, généralement un *Activity* ou un *Service*, qui est utilisé pour accéder aux ressources internes de l'application.

2.3.2 Fonctionnement de la méthode 'deSerialize()'

Cette méthode permet de récupérer un objet sérialisé précédemment selon la logique suivante :

1. Ouvrir un flux de fichier pour la lecture.
2. Créer un **ObjectInputStream** pour permettre la désérialisation de l'objet.
3. Lire l'objet à partir du fichier en utilisant la méthode `readObject()`.
4. Fermer l'**ObjectInputStream** après la lecture pour libérer les ressources.
5. Gérer les exceptions liées aux fichiers, à la désérialisation, ou aux classes non trouvées.
6. Retourner l'objet souhaité.

2.3.3 Le code de la méthode 'deSerialize()'

```
1   public static Object deSerialize(String filename, Context context
        ) {
2       try {
3           FileInputStream file = context.openFileInput(filename);
4           ObjectInputStream ois;
5           try {
6               ois = new ObjectInputStream(file);
7               try {
8                   Object object = ois.readObject();
9                   ois.close();
10                  return object;
11              } catch (ClassNotFoundException e) {
12                  e.printStackTrace();
13              }
14          } catch (StreamCorruptedException e) {
15              e.printStackTrace();
16          } catch (IOException e) {
17              e.printStackTrace();
18          }
19      } catch (FileNotFoundException e) {
20          e.printStackTrace();
```

```
        }
        return null;
    }
```

Attention

→ Il ne faut pas oublier d'implémenter l'interface **Serializable** dans la classe **Profil** qui est l'objet de la sérialisation dans notre cas.

3. Implémentation de la logique de sérialisation dans l'application

Maintenant, nous allons exploiter les fonctions statiques définies dans la classe **'Serializer'** par la couche controleur et vue de l'application.

3.1 Implémentation de la sérialisation dans le controleur

1. Déclarer un attribut de type chaine de caractères qui reflète le nom du fichier à utiliser dans la sérialisation.

```
    private static String nomFic = "saveprofil";
```

2. Ajouter la sérialisation du profil lors de sa création dans la méthode **creerProfil** de la classe **Controle**, en spécifiant le nom du fichier déclaré au-dessus, le profil crée et l'activité principale comme contexte qui sera passé en paramètre à partir de l'activité elle-même.

```
    public void creerProfil(Integer poids, Integer taille,
       Integer age, Integer sexe, Context contexte) {
        profil = new Profil(poids, taille, age, sexe);
        Serializer.serialize(nomFic, profil, contexte);
    }
```

3. Ajouter une fonction de désérialisation **recupSerialize** qui récupère les données sérialisés du fichier.

```
    private static void recupSerialize(Context contexte) {
        profil = (Profil) Serializer.deSerialize(nomFic,
           contexte);
    }
```

4. Appler cette dernière dans la méthode de création de l'instance car les données désérialisées doivent être récupérées dès le lancement de l'application.

```
    public static final Controle getInstance(Context contexte) {
        if (Controle.instance == null) {
            Controle.instance = new Controle();
            recupSerialize(contexte);
        }
        return Controle.instance;
    }
```

5. Changer la méthode **'recupSerialize'** et les variables qu'elle utilise en mode statique pour qu'elles soient reconnaisables par la classe **'Controle'**.

3.2 Implémentation de la sérialisation dans la couche vue

1. Fixer la méthode **afficheResult()** pour qu'elle envoie la classe **MainActivity** comme contexte, dans l'appel de la méthode du controleur **creerProfil**, vers la méthode qui exploite la sérialisation dans le controleur.

```
1     private void afficheResult(Integer poids, Integer taille,
          Integer age, Integer sexe) {
2         this.controle.creerProfil(poids, taille, age, sexe, this
              );
3         ...
4     }
```

2. Ajouter une variable pour le bouton radio du femme pour l'exploiter dans le code ultérieurement.

```
1     private RadioButton rdFemme;
2     ...
3     private void init() {
4     ...
5     rdFemme = (RadioButton) findViewById(R.id.rdFemme);
6     ...
7     }
```

3. Créer une méthode **recupProfil()** dans **MainActivity** qui vérifiera si un profil a été sérilaisé à partir de la fonction **recupSerialize()** qui se déclenche dès le lancement de l'applciation. Puis cette méthode remplira les champs de l'interface utilisateur avec les données récupérées.

```
1     private void recupProfil() {
2         if (controle.getPoids() != null) {
3             txtPoids.setText(controle.getPoids().toString());
4             txtTaille.setText(controle.getTaille().toString());
5             txtAge.setText(controle.getAge().toString());
6             rdFemme.setChecked(true);
7             if (controle.getSexe() == 1) {
8                 rdHomme.setChecked(true);
9             }
10            findViewById(R.id.btnCalc).performClick();
11        }
12    }
```

4. Appeler la méthode précédente dans la fonction **init()** de la classe **MainActivity** aprés la création de l'instance, pour qu'elle se déclenche lors du lancement de l'application tout de suite de la récupération des données sérialisées et elle affiche alors les données dans l'interface.

```
1     private void init() {
2         ...
```

```
3           this.controle = Controle.getInstance(this);
4           recupProfil();
5           ecouteCalcul();
6       }
```

4. Résultat et vérification de la sérialisation

Maintenant il suffit de lancer l'application, d'entrer des données, puis d'afficher le résultat. Ensuite, en relançant l'application, les mêmes valeurs saisies avant sa fermeture devraient apparaître. D'après cela, on peut conclure que la platforme Android gère ce type de persistance d'une manière spécifique.

4.1 Gestion des données internes d'une application sur Android

Android sauvegarde les données utilitaires d'une application pour une expérience utilisateur plus fluide. Il enregistre ces données et fichiers internes dans un répértoire privé et propriétaire à l'application nommée `cache`, garantissant que seules l'application elle-même et le système Android peuvent y accéder, sauf en cas d'autorisations spécifiques accordées à d'autres applications.
Le répertoire qui contient les fichiers de cache d'une application se trouve généralement à cet emplacement : **/data/data/<nom_du_paquet_de_l'application>/cache**, Ce dernier, connu sous le nom de *sandbox directory* en anglais), est un espace privé à l'application, offrant ainsi une confidentialité. L'utilisateur peut toutefois effacer manuellement le cache via les paramètres de stockage du smartphone pour libérer de l'espace ou résoudre des problèmes de performance.

4.2 Vérification de la sérialisation

Il existe plusieurs méthodes pour confirmer que l'implémentation de la sérialisation a été correctement réalisée. Dans cette section, nous allons en présenter deux.

4.2.1 Vérification à travers les paramètres de stockage

Pour vérifier de manière simple que la sérialisation est opérationnelle, il est possible de consulter les paramètres de stockage de l'application. Pour cela, suivez le chemin suivant : ***Paramètres > Applications > [Nom de l'application] > Stockage***, Ceci affiche des détails sur la taille des données stockées par l'application, y compris les éléments tels que Data et Cache. La taille des données (Data) doit être supérieure à 0 octet pour indiquer que les données de l'application ont bien été sauvegardées.

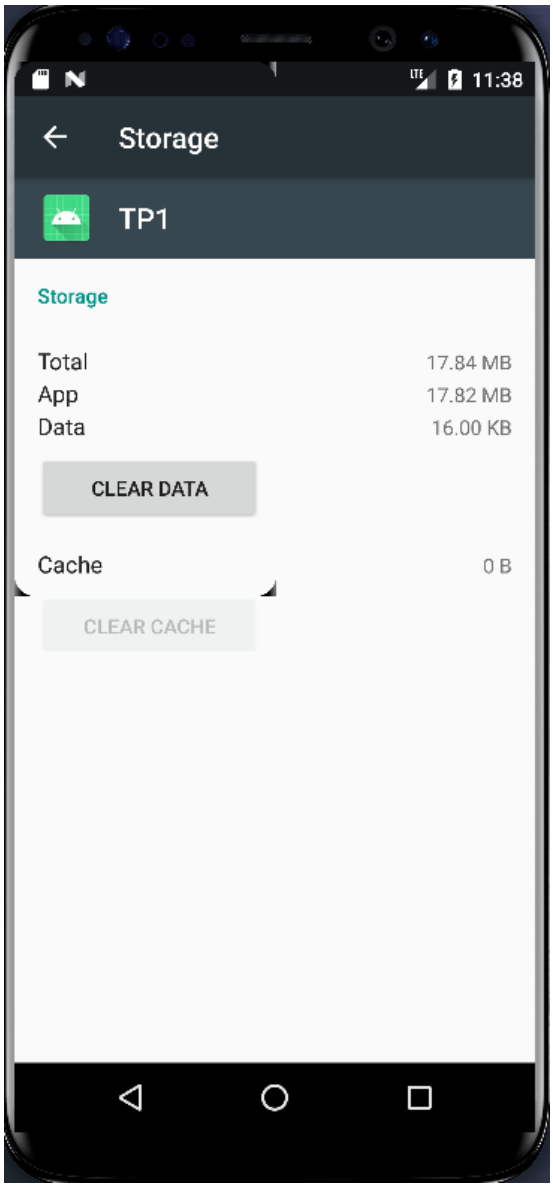

FIG. 1 : Détails du Stockage de l'Application

4.2.2 Vérification à Travers le Device Explorer dans Android Studio

Pour vérifier la présence du fichier de sérialisation lui-même, une méthode plus approfondie consiste à utiliser le `Device Explorer` dans Android Studio. Cette approche permet de naviguer dans le système de fichiers de l'appareil ou de l'émulateur afin de localiser et de visualiser directement le fichier stocké par l'application. Ceci sont les étapes nécessaires pour visualiser ce fichier.

1. Tout d'abord, ouvrir Android Studio et lancer l'émulateur.
2. Accéder ensuite à l'onglet Device Explorer dans la barre d'outils à partir de *View > Tool Windows > Device Explorer*, comme cette image l'indique.

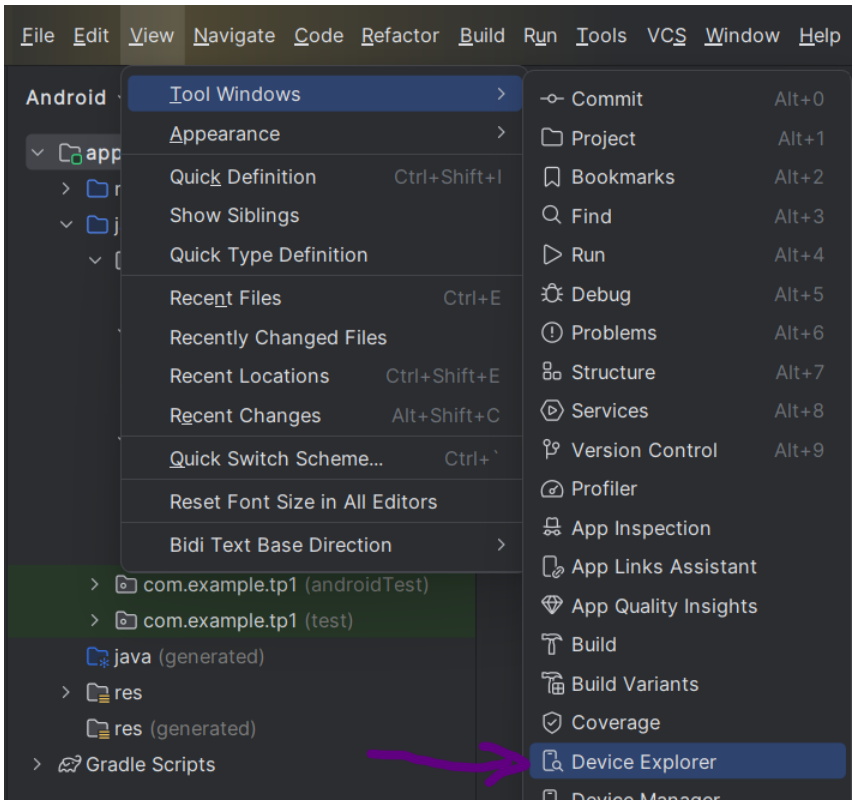

FIG. 2 : Accès à l'outil Device Explorer

3. À ce stade, l'outil doit apparaître en bas à gauche.

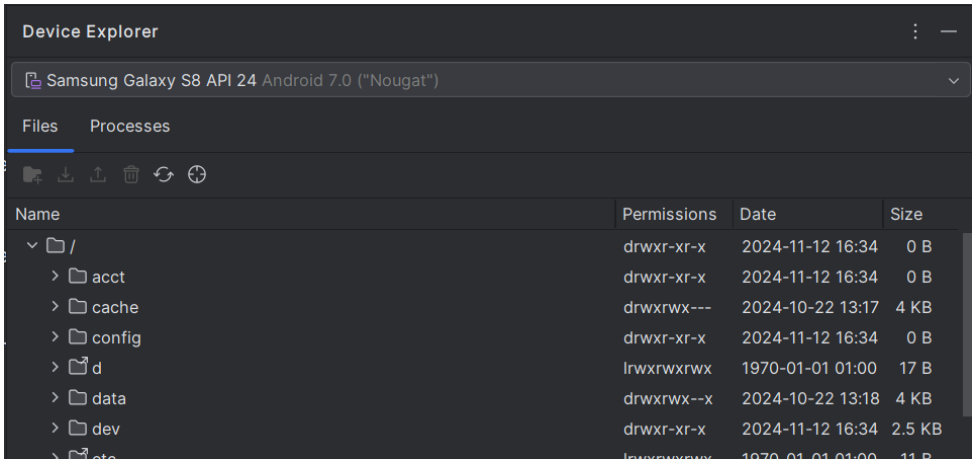

FIG. 3 : l'outil Device Explorer

4. Maintenan, il faut naviguer vers le fichier créé à partir de ce chemin : */data/data/<nom_du_paquet_de_l'application>/files*

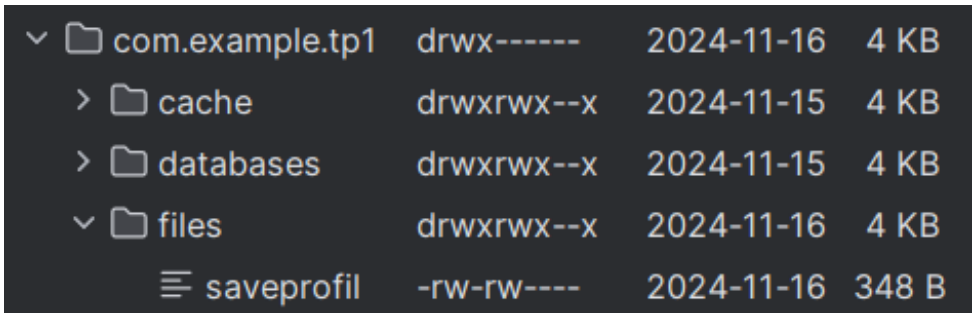

FIG. 4 : Emplacement du fichier

Nous pouvons télécharger le fichier en locale sur l'ordinateur par un clic droit sur celui-ci, puis en sélectionnant l'option *save as*. Ensuite, il suffit de choisir le dossier de destination souhaité.

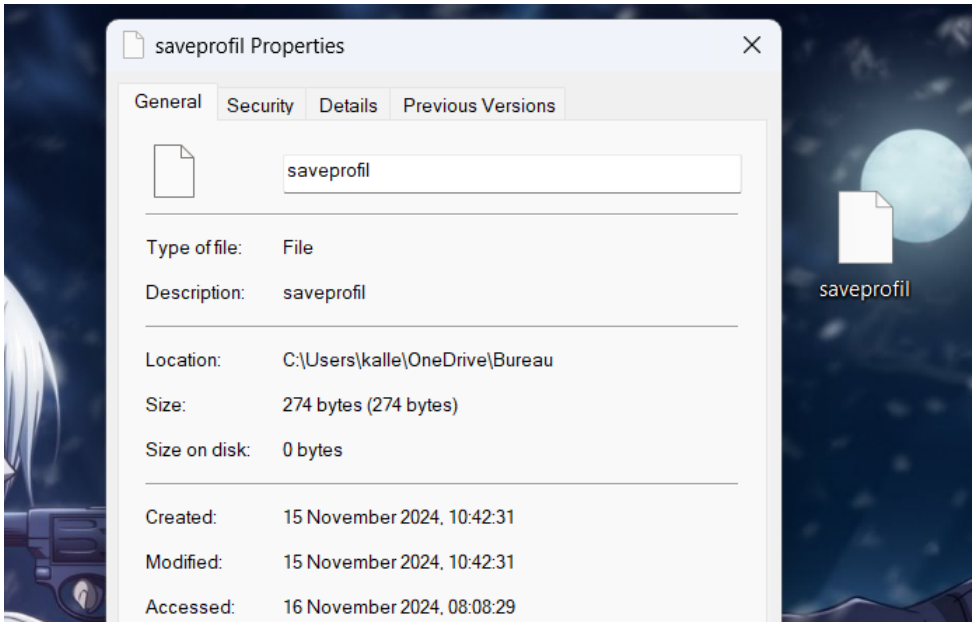

FIG. 5 : Fichier téléchargé en locale

Conclusion

Ce document a illustré l'implémentation de la sérialisation des objets pour sauvegarder et restaurer les données des utilisateurs, permettant ainsi une persistance des données afin de garantir une expérience utilisateur fluide.