

CR Électif PMR - Séquence 1

Qianyan WANG & Baoding LIU

1. Introduction

Cette séquence a pour objectif de réaliser une application de *ToDoList* dans un premier pas, qui permet de réaliser des fonctionnements en base d'une telle application.

Cette application va permettre aux différents utilisateurs de se connecter à leur compte personnel, de créer des listes de Todo (PMR, WEB par exemple), et de créer dans chaque liste des items Todo (TEA, TD, TP par exemple).

Elle permet aussi de stocker les données d'utilisateur, en utilisant *SharedPreference*, dans le mémoire intérieur de l'application.

2. Fonctionnement

Pour réaliser le fonctionnement de l'application, on a créé 4 activités :

- MainActivity

Le *MainActivity* est l'entrée de l'application. On va demander l'utilisateur de saisir son pseudo (dans un *TextView*) et de déterminer s'il veut que sa préférence soit mémorisée (par un *CheckBox*). Elle permet aussi de modifier les préférences.

Le pseudo vide ne sera pas accepté et l'utilisateur sera demandé de saisir à nouveau. Si le pseudo est valide, on cherche les données correspondantes dans le mémoire interne. On le charge s'il y en a, le crée sinon.

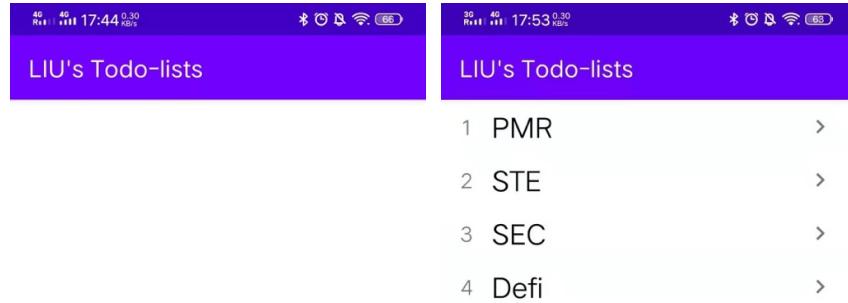
En cliquant sur le bouton OK, on passe à l'activité *ChoixListActivity*. En cliquant sur « Préférences », on passe à l'activité *SettingActivity*.



- ChoixListActivity

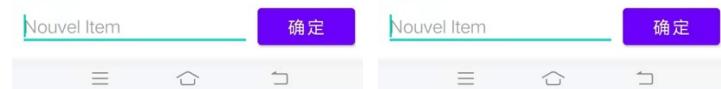
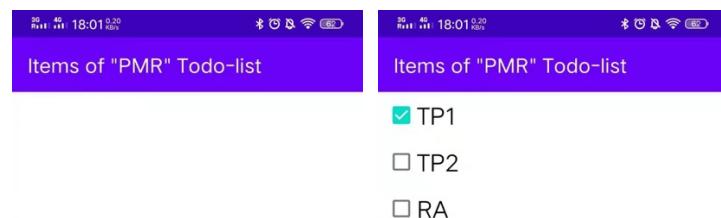
Cette activité permet d'actualiser les listes de Todo. Elle va porter l'information de pseudo d'utilisateur dans le titre. Elle permet d'ajouter des nouvelles listes Todo dans le compte. On peut ajouter une nouvelle liste dont le titre est saisi dans le TextView (un titre vide ne sera pas accepté) et en cliquant sur le bouton OK. Pour être plus joli, on ajoute les nombres d'ordre et une petite flèche à droite.

En cliquant sur une des listes, on va passer à l'activité ShowListActivity.



● ShowListActivity

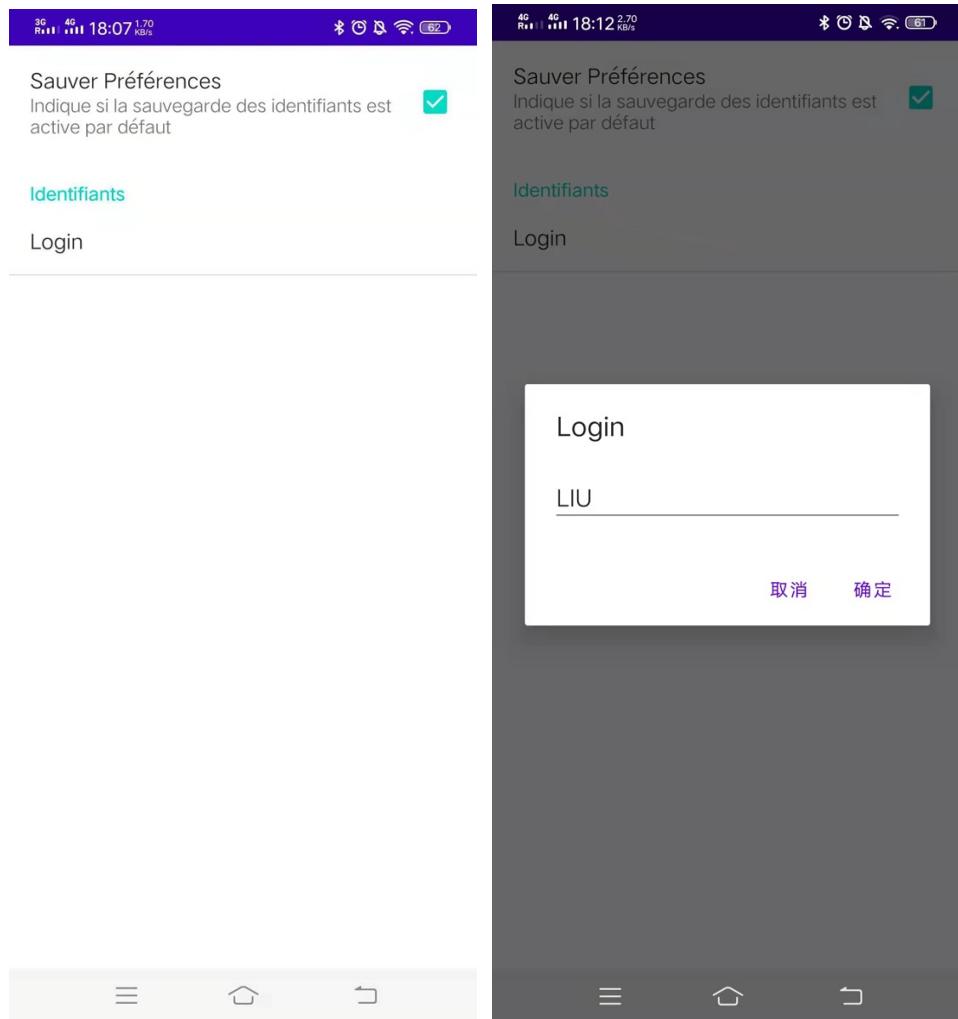
Cette activité permet de visualiser les items Todo de la liste cliquée. Les items sont présentés par un CheckBox, pour indiquer s'ils sont déjà faits ou pas encore.



Le titre de cette page porte de l'information sur le nom de la liste. On peut ajouter un nouvel item dont la description est saisie dans le TextView (une description vide ne sera pas acceptée) et en cliquant sur le bouton OK. Les nouveaux items affichent de la même manière.

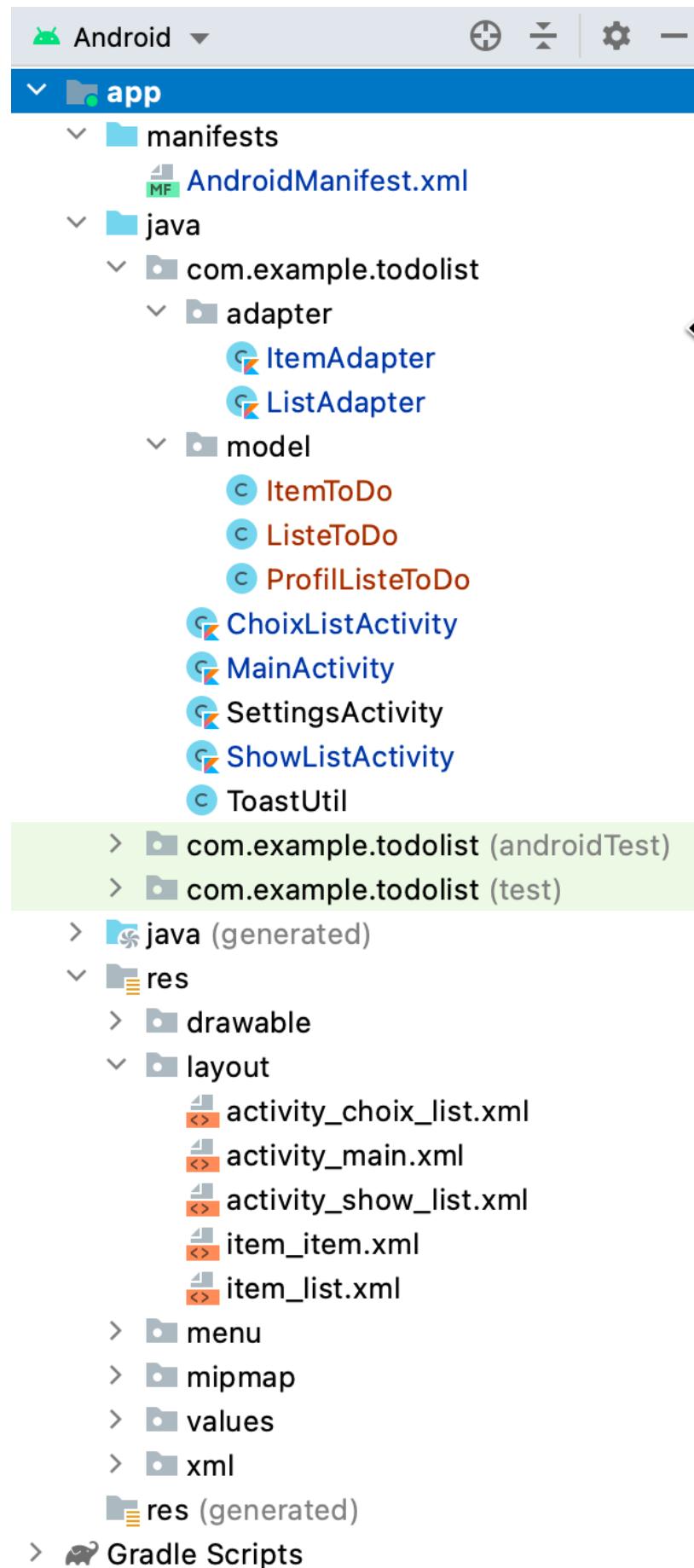
● SettingActivity

Cette activité est l'activité des préférences, qui est entrée par le MainActivity. Pour l'instant on n'a que deux préférences : le pseudo et se souvenir de soi ou pas. Quand on change les préférences ici, ceux sur la page Main changent aussi et vice versa.



3. Réalisation

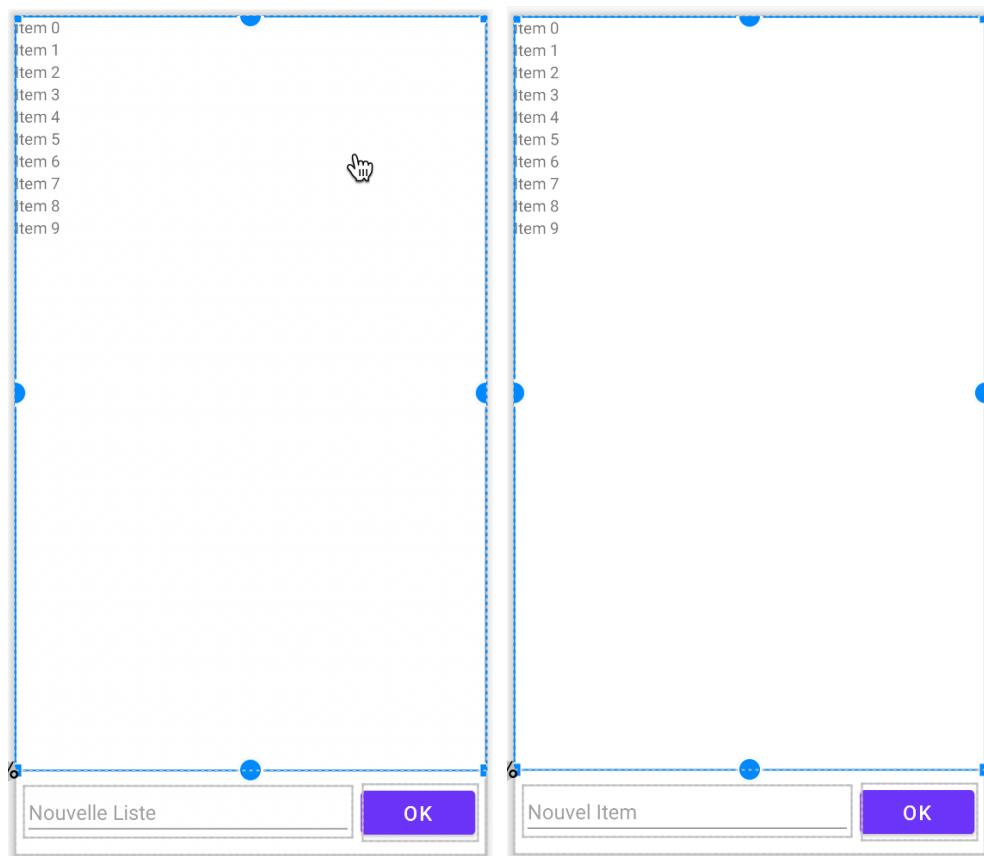
Pour le réaliser plus correctement, on a beaucoup d'autres classes à part de ces quatre activités comme un projet dans Android Studio, dont le langage est mixte de Java et Kotlin.



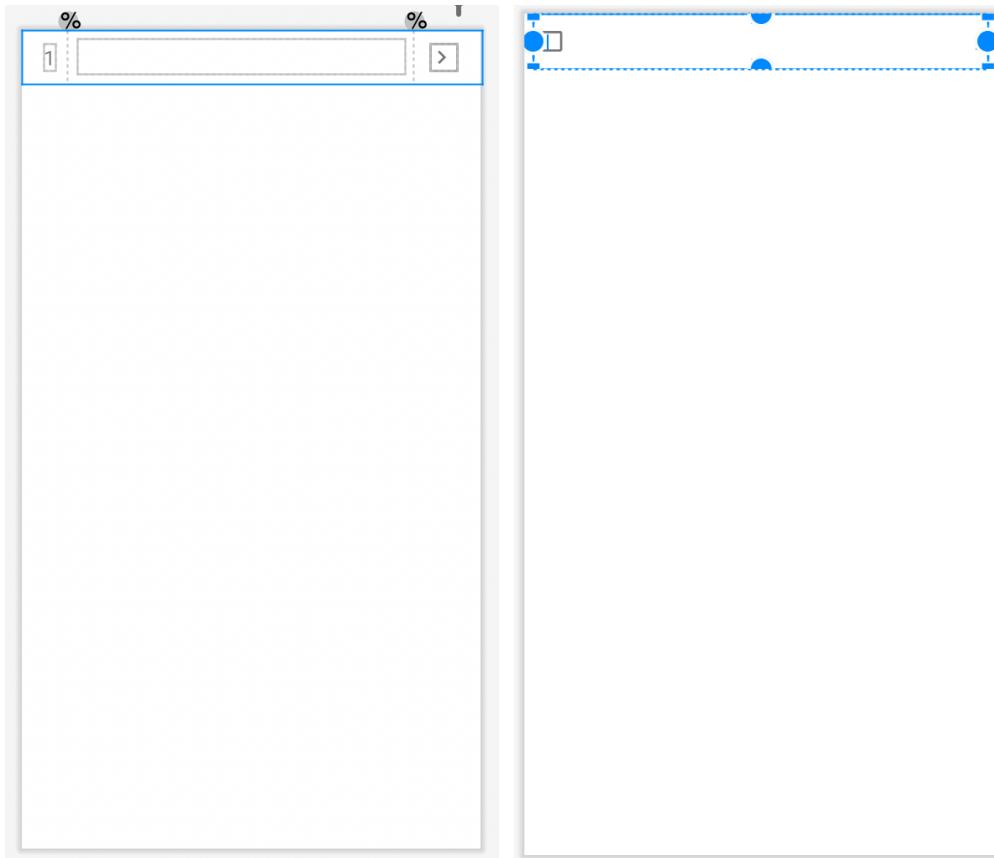
- Adapter

Pour afficher des listes et des items Todo, on utilise le *RecyclerView* et l'adapter.

RecyclerView :



Dans l'adapter, on a besoin d'un modèle de l'affichage des items et des listes, qui sont définis par un fichier XML :



On peut aussi définir les actions à faire quand on clique sur les items et les listes.
Clique sur une liste : passer dans l'activité *ShowListActivity*.

```

53     val adapter = ListAdapter(lists)
54
55     // start ShowListActivity
56     val intent = Intent(packageContext: this, ShowListActivity::class.java)
57     intent.putExtra(name: "pseudo", pseudo)
58
59     // action when click on the list
60     adapter.setOnItemClickListener(object : ListAdapter.OnItemClickListener {
61         override fun onItemClick(position: Int) {
62             val listName = lists[position].titreListeToDo
63             ToastUtil.newToast(context, content: "this is $listName")
64             // pass the clicked list
65             intent.putExtra(name: "profil", profil)
66             intent.putExtra(name: "whichList", position)
67             startActivityForResult(intent, requestCode)
68         }
69     })
70
71     recyclerView.adapter = adapter
72     recyclerView.layoutManager = LinearLayoutManager(context: this, RecyclerView.VERTICAL, reverseLayout: false)

```

Clique sur un item : changer le statut du *CheckBox* et la propriété *isFait* de l'item.

```

23     override fun onBindViewHolder(holder: ItemViewHolder, position: Int) {
24         holder.bind((dataset[position]))
25         holder.cbView.setOnCheckedChangeListener(null)
26         holder.cbView.setChecked(checkStatus.get(position) == true);
27         holder.cbView.setOnCheckedChangeListener(CompoundButton.OnCheckedChangeListener { buttonView, isChecked ->
28             checkStatus.put(position, isChecked)
29             if (isChecked) {
30                 ToastUtil.showToast(holder.cbView.context, content: holder.cbView.text.toString() + " has been done")
31                 dataset[position].isFait = true // update ItemToDo.isFait
32             } else {
33                 ToastUtil.showToast(holder.cbView.context, content: holder.cbView.text.toString() + " to do")
34                 dataset[position].isFait = false // update ItemToDo.isFait
35             }
36         })
37     }

```

Dans le *ItemAdapter*, on a besoin de mémoriser le *isFait* de chaque item, on utilise donc un *HashMap* pour stocker.

```

class ItemAdapter(private val dataset: MutableList<ItemToDo>): RecyclerView.Adapter<ItemAdapter.ItemViewHolder>() {
    val checkStatus: HashMap<Int, Boolean> = HashMap() // store the check status for all checkboxes
    val CAT: String = "TODO_ITEM"

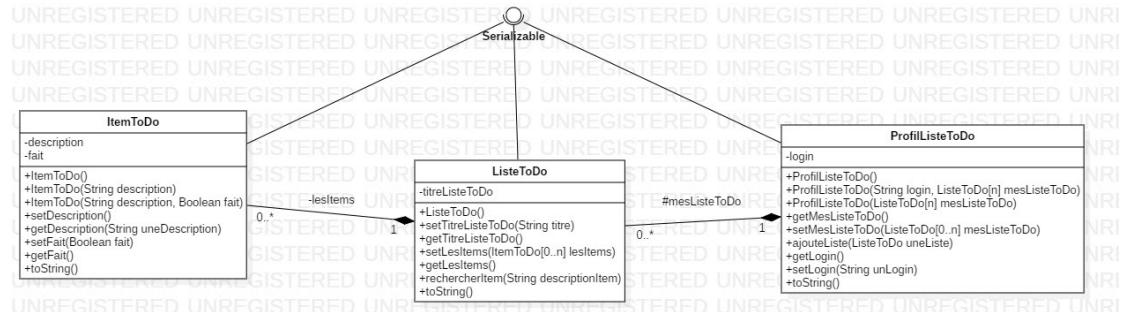
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ItemViewHolder {
        val inflater = LayoutInflater.from(parent.context)
        return ItemViewHolder(itemView = inflater.inflate(R.layout.item_item, parent, attachToRoot: false))
    }
}

```

Il nous permet de garder tous les *isFait* corrects en glissant le *RecyclerView* ou en passant les données parmi des activités.

● Model

Pour réaliser le fonctionnement, on a besoin des structures adéquates des données pour stocker les données d'un compte. On organise les modèles de données suite au diagramme de classe suivant.



ItemToDo :

```
1 package com.example.todolist.model;
2
3 import java.io.Serializable;
4
5 public class ItemToDo implements Serializable {
6     private static final long serialVersionUID=3L;
7     private String description;
8     private boolean fait;
9
10    public ItemToDo() {
11    }
12
13    public ItemToDo(String description, boolean fait) {
14        this.description = description;
15        this.fait = fait;
16    }
17
18    public ItemToDo(String description) { this(description, false); }
19
20    public String getDescription() { return description; }
21
22    public void setDescription(String description) { this.description = description; }
23
24    public boolean isFait() { return fait; }
25
26    public void setFait(boolean fait) { this.fait = fait; }
27
28    @Override
29    public String toString() {
30        return "\"" + description + ":" + fait;
31    }
32}
```

ListeToDo :

```

1 package com.example.todolist.model;
2
3 import ...
4
5
6 public class ListeToDo implements Serializable {
7     private static final long serialVersionUID=2L;
8     private String titreListeToDo;
9     private ArrayList<ItemToDo> lesItems;
10
11    public ListeToDo() {
12    }
13
14    public ListeToDo(String titreListeToDo, ArrayList<ItemToDo> lesItems) {
15        this.titreListeToDo = titreListeToDo;
16        this.lesItems = lesItems;
17    }
18
19    public ListeToDo(String titreListeToDo) {
20        this.titreListeToDo = titreListeToDo;
21        this.lesItems = new ArrayList<ItemToDo>();
22    }
23
24    public String getTitreListeToDo() { return titreListeToDo; }
25
26    public void setTitreListeToDo(String titreListeToDo) { this.titreListeToDo = titreListeToDo; }
27
28    public ArrayList<ItemToDo> getLesItems() { return lesItems; }
29
30    public void setLesItems(ArrayList<ItemToDo> lesItems) { this.lesItems = lesItems; }
31
32    public int rechercherItem(String descriptionItem) {
33        for (int i = 0; i < lesItems.size(); i++) {
34            if (descriptionItem == lesItems.get(i).getDescription()) {
35                return i;
36            }
37        }
38        return -1;
39    }
40
41    public void ajouteItem(ItemToDo unItem) { this.lesItems.add(unItem); }
42
43
44    @Override
45    public String toString() {
46        return "ListeToDo{"
47            + "titreListeToDo='" + titreListeToDo.toString() + '\''
48            + ", lesItems=" + lesItems.toString()
49            + '}';
50    }
51}
52
53
54 ProfilListeToDo :

```

```

1  package com.example.todolist.model;
2
3  import ...
4
5
6  public class ProfilListeToDo implements Serializable {
7      private static final long serialVersionUID=1L;
8      private String login;
9      protected ArrayList<ListeToDo> mesListeToDo;
10
11     public ProfilListeToDo() {
12     }
13
14     public ProfilListeToDo(String login, ArrayList<ListeToDo> mesListeToDo) {
15         this.login = login;
16         this.mesListeToDo = mesListeToDo;
17     }
18
19     public ProfilListeToDo(ArrayList<ListeToDo> mesListeToDo) { this.mesListeToDo = mesListeToDo; }
20
21     public ProfilListeToDo(String login) {
22         this.login = login;
23         this.mesListeToDo = new ArrayList<ListeToDo>();
24     }
25
26
27     public ArrayList<ListeToDo> getMesListeToDo() { return mesListeToDo; }
28
29     public void setMesListeToDo( ArrayList<ListeToDo> mesListeToDo) {
30         this.mesListeToDo = mesListeToDo;
31     }
32
33     public String getLogin() { return login; }
34
35     public void setLogin(String login) { this.login = login; }
36
37     public void ajouteListe(ListeToDo uneListe) { this.mesListeToDo.add(uneListe); }
38
39
40     @Override
41     public String toString() {
42         return "ProfilListeToDo{" +
43             "login='" + login.toString() + '\'' +
44             ", mesListeToDo=" + mesListeToDo.toString() +
45             '}';
46     }
47
48 }

```

Pour assurer le bon fonctionnement de la sérialisation, on ajoute une priorité *serialVersionUID* dans chaque de ces trois modèles de donnée.

- **ToastUtil**

On constate que les toasts s'affichent pendant une durée fixe, qui est 3.5s pour *Toast.LENGTH_SHORT* et 5s pour *Toast.LENGTH_LONG*, qui est gênant quand on veut faire afficher des toasts avec une petite écart. On crée donc cette classe pour le résoudre. En effet, cette classe crée une liste des toasts, quand on veut afficher un autre toast alors que celui précédent était en train de s'afficher, elle va appeler la méthode *cancel()* de celui précédent et insérer le dernier dans la liste et afficher.

```

1  package com.example.todolist;
2
3  import android.content.Context;
4  import android.widget.Toast;
5
6  import java.util.ArrayList;
7
8  public class ToastUtil {
9      private static ArrayList<Toast> toastList = new ArrayList<Toast>();
10
11     public static void newToast(Context context, String content) {
12         cancelAll();
13         Toast toast = Toast.makeText(context, content, Toast.LENGTH_SHORT);
14         toastList.add(toast);
15         toast.show();
16     }
17
18     public static void cancelAll() {
19         if (!toastList.isEmpty()){
20             for (Toast t : toastList) {
21                 t.cancel();
22             }
23             toastList.clear();
24         }
25     }
26 }
```

- Sérialisation des objets

On souhaite transmettre des données entre activités. Pour faciliter la transmission, il est mieux de transmettre les structures des données regroupées, c'est-à-dire les objets, mais pas les données indépendantes.

Par conséquent, on a besoin de sérialiser des objets pour pouvoir les transmettre. On fait donc les structures de donnée (les Model) implémenter de l'interface *Serializable*.

```

5  public class ItemToDo implements Serializable
6  public class ListeToDo implements Serializable
6  public class ProfilListeToDo implements Serializable
```

Après la transmission, il faut les déserialiser pour obtenir l'objet original.

```
27  profil = intent.getSerializableExtra( name: "profil") as ProfilListeToDo
```

- Stockage des données d'utilisateur & Passage des données entre activités

Pour pouvoir modifier le compte qui correspond au pseudo entré dans des activités suivantes (*ShowListActivity*, *ChoixListActivity*), on a besoin de transmettre les données entre activités, par exemple transmettre les données du compte obtenues dans *MainActivity* aux *ShowListActivity*, *ChoixListActivity*.

Pour le faire, on utilise le *SharedPreferences* qui est une méthode de stockage interne donc ne demande pas d'autre permission, et qui est facile à utiliser. Et on utilise JSON pour stocker.

On utilise le *SharedPreferences* pour réaliser la transmission des préférences.

Initialisation des objets :

```
20     private lateinit var sp: SharedPreferences
21     private lateinit var editor: SharedPreferences.Editor
36     sp = PreferenceManager.getDefaultSharedPreferences(context: this)
37     editor = sp.edit()
```

Stocker les changements :

```
45     btnOK.setOnClickListener { it: View!
46         login = pseudo.text.toString()
47         if (login==""||login==null){
48             ToastUtil.newToast(context: this, content: "Please enter your login!")
49         } else {
50             ToastUtil.newToast(context: this, content: "click on btnOK")
51             if (cbRemember.isChecked) {
52                 editor.putString("login", login)
53                 editor.commit()
54             }
55         }
56     }
57
58     cbRemember.setOnClickListener { it: View!
59         ToastUtil.newToast(context: this, content: "click on cb")
60         editor.putBoolean("remember", cbRemember.isChecked)
61         editor.commit()
62         if (!cbRemember.isChecked) {
63             editor.putString("login", "")
64             editor.commit()
65         }
66     }
67
68 }
69 }
```

Relire des préférences changées :

```
112     override fun onStart() {
113         super.onStart()
114         // relire les preferences partagées de l'app
115         val cbR = sp.getBoolean(key: "remember", defaultValue: false)
116
117         // actualiser l'état de la case à cocher
118         cbRemember.isChecked = cbR
119
120         // si la case est cochée, on utilise les preferences pour définir le login
121         if (cbRemember.isChecked) {
122             val l = sp.getString(key: "login", defaultValue: "login inconnu")
123             pseudo.setText(l)
124         } else {
125             // sinon, le champ doit être vide
126             pseudo.setText("")
127         }
128     }
129 }
```

On l'utilise aussi pour réaliser la transmission de l'information du compte.

Initialisation des objets :

```

25     private lateinit var editorJson: SharedPreferences.Editor
26     private lateinit var login: String
27     val gson = Gson()

41         // a SP object to pass messenger between activities
42     spJson = getSharedPreferences(name: "SP_Data_List", MODE_PRIVATE)
43     editorJson = spJson.edit()

```

Lire le compte de stockage interne (format JSON) et le transforme à l'objet (s'il existe) ou créer un nouveau compte (s'il n'existe pas), et le passer à l'activité suivante (de *MainActivity* à *ChoixListActivity* par exemple) :

```

57     var jsonStr: String? = spJson.getString(login, defaultValue: "")
58
59     // create the profil of this login if not exists, load if exists
60     if (jsonStr == null || jsonStr == "") {
61         profil = ProfilListeToDo(login)
62     } else {
63         profil = gson.fromJson(jsonStr, ProfilListeToDo::class.java)
64     }

```

D'ailleurs, on a aussi besoin de transmettre les données modifiées à l'activité précédente, pour qu'il soit plus cohérent.

Prenons la transmission entre *MainActivity* et *ChoixListActivity* par exemple :

Transmettre les données du compte à *ChoixListActivity* :

```

74     val intent = Intent(packageContext: this, ChoixListActivity::class.java)
75     intent.putExtra(name: "profil", this.profil)
76     startActivityForResult(intent, requestCode)

```

On utilise *startActivityForResult()* au lieu de *startActivity()* quand on veut recevoir les données à l'inverse.

Transmettre les données modifiées de *ChoixListActivity* à *MainActivity* quand on appuie le bouton retour dans *ChoixListActivity* :

```

154     // pass data to MainActivity when press "back"
155     override fun onBackPressed() {
156         val intent = Intent()
157         intent.putExtra(name: "profil", profil)
158         setResult(RESULT_OK, intent)
159         super.onBackPressed()
160     }

```

Recevoir les données transmises dans *MainActivity* et les stocker en appelant *commit()* :

```
139 // receive data from ChoixListActivity after press "back"
140 override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?) {
141     super.onActivityResult(requestCode, resultCode, data)
142     when (requestCode) {
143         requestCode -> if (resultCode === RESULT_OK) {
144             val returnData: ProfilListeToDo = data!!.getSerializableExtra("profil") as ProfilListeToDo
145             this.profil = returnData
146         }
147     }
148     Log.i(CAT, this.profil.toString())
149
150     // stock changes
151     val response = gson.toJson(this.profil)
152     editorJson.putString(login, java.lang.String.valueOf(response)) // save Json
153     editorJson.commit() // stock
154 }
```

Pour le passage de données entre *ChoixListActivity* et *ShowListActivity*, la procédure est la même.

- Ajoute des nouvelles listes et des nouveaux items

Le fonctionnement est que quand on clique sur le bouton OK, une nouvelle liste ou un nouvel item est ajouté. Donc on fait un *onClickListener* pour le bouton OK.

```
74     btnOKList.setOnClickListener { it: View!  
75         var newListName = etNewList.text.toString()  
76         if (newListName==null || newListName==""){  
77             ToastUtil.newToast(context, content: "Please enter the name of list")  
78         }else {  
79             ToastUtil.newToast(context, content: "Add \"\$newListName\"")  
80             adapter.addData(newListName)  
81  
82             // add new list  
83             var newList : ListeToDo = ListeToDo(newListName)  
84             profil.ajouteListe(newList)  
85  
86             etNewList.setText("") //clear the input area  
87         }  
88  
89     }  
59     btnOKItem.setOnClickListener { it: View!  
60         Log.i(CAT, msg: "map: "+adapter.checkStatus.toString())  
61         var newItemName = etNewItem.text.toString()  
62         if (newItemName==null || newItemName==""){  
63             ToastUtil.newToast(context, content: "Please enter the name of item")  
64         }else {  
65             ToastUtil.newToast(context, content: "Add \"\$newItemName\"")  
66  
67             // add new item  
68             var newItem : ItemToDo = ItemToDo(newItemName)  
69             adapter.addData(newItem) // add new item  
70  
71             listTodo.ajouteItem(newItem)  
72  
73             etNewItem.setText("") // clear the input area  
74     }  
75 }
```

Il faut aussi qu'on réalise une méthode dans l'adapter pour que le nouvel item

s'affiche.

```

41   public fun addData(item: ItemToDo) {
42       // add data in the list to display
43       dataset.add(item)
44       checkStatus.put(dataset.size-1, item.isFait)
45       notifyItemChanged(dataset.size)
46   }
47
48
49   public fun addData(text: String) {
50       // add data in the list to display
51       var uneListe: ListeToDo = ListeToDo()
52       uneListe.titreListeToDo = text
53       dataset.add(uneListe)
54       notifyItemChanged(dataset.size)
55   }

```

On a aussi besoin d'appliquer réellement ces modifications dans la structure des données.

```

44   public void ajouteListe(ListeToDo uneListe) {
45       this.mesListeToDo.add(uneListe);
46   }
47
48   public void ajouteItem(ItemToDo unItem){
49       this.lesItems.add(unItem);
50   }
51

```

4. Conclusion

On a fait au premier pas notre application TodoList, en faisant cela, on a appris comment mettre en place de layouts évolués utilisant des *RecyclerView* et des *Adapters*, comment faire la Sérialisation ou la Désérialisation en JSON avec la librairie GSON, comment stocker et lire de données en préférences.

5. Perspectives

- Ajouter une façon de supprimer les listes ou les items.
- Donner des avis quand l'utilisateur saisi le pseudo en fonction des comptes en stockage.
- Compléter le Settings en ajoutant le mot de passe.
- Trouver une autre façon plus adéquate pour stocker les données plus complexes (API par exemple).

6. Bibliographie

- <https://developer.android.com/training/data-storage/app-specific?hl=zh-cn>
- <https://www.cnblogs.com/Free-Thinker/p/11937671.html>
- <https://blog.csdn.net/w410589502/article/details/59117647>
- <https://www.an.rustfisher.com/android/io/file/read-write-json-file/>
- <https://blog.csdn.net/gjy211/article/details/52414962>
- https://blog.csdn.net/weixin_35594924/article/details/113451785
- <https://www.jianshu.com/p/c04b8899cf85>
- https://blog.csdn.net/QI_Rainbow/article/details/52073278
- <https://www.jianshu.com/p/a34c644e3431>
- <https://www.runoob.com/kotlin/kotlin-loop-control.html>
- <https://blog.csdn.net/leejizhou/article/details/51105060>
- <https://blog.csdn.net/chaoyu168/article/details/88942466>