ATLS 4320: Advanced Mobile Application Development
Week 13: Shared Preferences

Data Persistence

https://developer.android.com/training/data-storage

There are multiple data persistence approaches on Android. The approach you pick should be based on

- What kind of data you need to store
- How much space your data requires
- How reliable does data access needs to be
- Whether the data should be private to your app

App-specific storage

https://developer.android.com/training/data-storage/app-specific

All Android devices have two file storage areas: "internal" and "external" storage. These names come from the early days of Android, when most devices offered built-in non-volatile memory (internal storage), plus a removable storage medium such as a micro SD card (external storage). Many devices now divide the permanent storage space into separate "internal" and "external" partitions.

Internal file storage

Internal storage is best when you want to be sure that neither the user nor other apps can access the data. No permission is needed to read and write files to internal storage.

Pros

- Always available
- By default files saved are private to your app
- Other apps and the user can't access the files
- Starting in Android 10(API 29) internal storage directories will be encrypted

Cons

- Hard to share data
- Internal storage might have limited capacity

External file storage

External storage often provides more space than internal storage for app-specific files. Other apps can access these files if they have the proper permissions.

Pros

- Often provides more room

Cons

- Might not always be available as it might resides on a physical volume that the user might be able to remove
- Must verify that the volume is accessible before using

Both internal and external storage provide directories for an app's persistent files and another for an app's cached files.
In both internal and external storage files are removed when the user uninstalls your app

For files that you want to persist past the lifetime of an app you should use shared storage instead.

Shared storage
https://developer.android.com/training/data-storage/shared
Use shared storage for user data that should be accessible to other apps and saved even if the user uninstalls your app.

- Use the MediaStore API to store media content in a common location on the device
- The Storage Access Framework uses a document provider to store documents and other files in a specific directory

Databases
https://developer.android.com/training/data-storage/room
Saving data to Android's SQLite database is ideal for persisting large amounts of structured data locally. Similar to internal storage, Android stores your database in your app's private folder and therefore is not accessible to other apps or the user.
The Room library provides an abstraction layer over SQLite that makes it much easier to work with SQLite. It is highly recommended over using SQLite APIs directly. It is one of the architecture components that is included in Jetpack.

Shared Preferences (different than preferences through Settings)
https://developer.android.com/training/data-storage/shared-preferences.html
Using shared Preferences for data persistence a good choice for small amounts of data that doesn't require structure. Shared Preferences stores data as key/value pairs in an unencrypted XML file in internal storage.

- Keys are always of type String
- Values must be primitive data types: boolean, float, int, long, String, and stringset
- You can use a single file or multiple files
- You can use the default or a named preference
- Preference data can be deleted from the device by the user
- Data is deleted when the app is uninstalled

Using a shared preferences file

- Use getPreferences() if you're only using one shared preference file in an activity as this uses the the default shared preference file for the activity
- Use getSharedPreferences() if you need multiple shared preferences files each with a unique name
- Use MODE_PRIVATE as the security visibility parameter so the preference are private and no other application can access it
    - The other modes, MODE_WORLD_READABLE and MODE_WORLD_WRITEABLE, have been deprecated since API level 17 and since API 24 throw a security exception

Writing to a shared preferences file

1. Create a **SharedPreferences.Editor** by calling **edit()** on your SharedPreferences object
2. Add the key-value pairs using methods like **putInt(), putString(),** and **putStringSet()**
3. Save the changes
    - apply() changes the in-memory SharedPreferences object immediately but writes the data to disk asynchronously
        - does not return a boolean indicating success or failure
        - the recommended way to save changes
    - commit() writes the data to disk synchronously

- you should not call commit() from the UI thread as it could freeze the UI
- returns a boolean indicating success or failure
- not recommended to use

Reading from a shared preferences file

1. Get access to your SharedPreferences object
2. Read in the key-value pairs using methods like **getInt(), getString(),** and **getStringSet()**
    1. optionally provide a default value to return if the key isn't present

List

I'm going to update my List app to use Shared Preferences for data persistence.
Loading, persisting, and managing data are outside of the scope of what a ViewModel class traditionally handles so I'm going to create a separate class to work with shared preferences.

Create a new package named util.
In that package create a new class in that package called SharedPrefsData. This class will be in charge of saving and reading the data using a shared preferences file.

In SharedPrefsData.kt add a constant for the name of your shared preferences file and methods to write to and read from a shared preferences file.

```
class SharedPrefsData {
    private val prefs_file = "ITEMS"
}
```

Because shared preferences can't save an ArrayList I'll be saving each item individually. But knowing how many items there are will be helpful when loading in the data so I save the size of the array as well.

```
    fun saveDataSharedPrefs(itemList: ArrayList<Item>, context: Context) {
        //get access to a shared preferences file
        val sharedPrefs = context.getSharedPreferences(prefs_file,
Context.MODE_PRIVATE)
        //create an editor to write to the shared preferences file
        val editor = sharedPrefs.edit()
        //add size to the editor
        if (itemList != null) {
            editor.putInt("size", itemList.size)
        }
        for ((index, item) in itemList.withIndex()){
            editor.putString("item$index", item.name)
        }
        //save the data
        editor.apply()
    }
```

For loading the data you just need to get access to the shared preferences file using the same file name. Use the keys that you used when saving the data to read the data. Use getInt() for the size as the value is an Integer and getString() for the items as those values are type String.
I also pass in a default value of 0 for size. This is the value that will be returned if that key isn't found in the preferences file such as the first time the loadData() method is called.
If nothing was saved, or if the saved size is 0, we won't try to access any items as they won't exist.

```
    fun loadDataSharedPrefs(context: Context):ArrayList<Item>{
        var loadedItemList = ArrayList<Item>()
        //get access to a shared preferences file
        val sharedPrefs = context.getSharedPreferences(prefs_file,
Context.MODE_PRIVATE)
        val size = sharedPrefs.getInt("size", 0)
        for (i in 0 until size){
            loadedItemList.add(Item(sharedPrefs.getString("item$i", "")!!))
        }
        return loadedItemList
    }
```

Because I want the view model to be the single point of contact for data activities, I'll add methods to our ItemViewModel class to call these methods.
Because the shared preferences methods need the application context I'm going to change my ItemViewModel to have a superclass of AndroidViewModel instead of ViewModel. The only difference between these two superclasses is that AndroidViewModel takes in an application context and ViewModel does not.
Although our view model should not have any references to an activity or fragment context because it might outlive them in the lifecycle, application context is ok because view models have the same lifecycle scope as the application.
Note that because we're calling setValue() in loadData(), the observer we have set up in MainActivity.kt will also be fired.

```
class ItemViewModel(application: Application) :
AndroidViewModel(application){
    val itemList = MutableLiveData<ArrayList<Item>>()
    private var newList = ArrayList<Item>()
    val context = application.applicationContext
    val sharedPrefsData = SharedPrefsData()

    fun add(item: Item){
        newList.add(item)
        itemList.value = newList
    }

    fun delete(item: Item){
        newList.remove(item)
        itemList.value = newList
    }

    fun loadData(){
        newList = sharedPrefsData.loadDataSharedPrefs(context)
        itemList.value = newList
        Log.i("vm load", itemList.value!!.size.toString())
    }

    fun saveData(){

itemList.value?.let{sharedPrefsData.saveDataSharedPrefs(itemList.value!!,
context)}
```

```
    }
}
```

In MainActivity.kt we still want our view model to be the single source for our data so we only want to load data from shared preferences when the application starts and the view model doesn't have any data. We don't need to load the data every time the activity starts because the view model has a lifetime of the application. Add to onCreate()

```
if (viewModel.itemList.value == null){
    viewModel.loadData()
    Log.i("main", "in if to load data")
}
```

There are two possible approaches for when to write the data to shared preferences.

1. Every time the data changes, so when an item is added or deleted. This will ensure our shared preferences is always up to date even if the app is killed (and therefore no lifecycle methods are called) or the phone is rebooted and there's no chance to save the data. Depending on your all this might cause a lot of write calls.
2. Save the data when the app is going into the background by using the onStop() lifecycle method. This will result in fewer write operations but there is a chance the app will lose data if the app is killed in a manner where no lifecycle methods are called (app is killed due to resource constraints, device battery dies).

Since we're using LiveData and already have an observer set up to be called every time the view model data is changed, let's use that to save our data to shared preferences as well.
In the Observer definition you only need to add one line to call saveData() when the data has changed.

```
viewModel.itemList.observe(this, Observer {
    Log.i("main", "in observer")
    adapter.update()
    viewModel.saveData()
})
```

Now when you test your app you should see your item list even after the application has been terminated and restarted. (either stop the app from Android Studio or kill the app on the device by swiping up on its card).

When testing if you ever want to start over with an empty shared preferences file just change the value of your constant and it will create a new shared preferences file.