Advanced Mobile Application Development
Week 14: DataStore

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 for files in your app that other apps don't, or shouldn't, access: "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

- Should check the amount of free space before saving files
- 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 increased capacity

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 instead of using SQLite APIs directly. It is one of the architecture components that is included in Jetpack. We'll be looking at this on Thursday.

Key-Value Data
https://developer.android.com/training/data-storage/shared-preferences.html
Shared Preferences (different than preferences through Settings)
For small amounts of data that doesn't require structure saving the data as key-value pairs is a good choice.
The Shared Preferences API has been around since API 1and 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

Some downsides of SharedPreferences include:

- Lack of safety from runtime exceptions
- Lack of a fully asynchronous API
- Lack of main thread safety
- No type safety

DataStore
https://developer.android.com/topic/libraries/architecture/datastore
The new DataStore library has been added to Android Jetpack and provides an improved local data solution. DataStore provides two different implementations.

1. Preferences DataStore to store data using key-value pairs
    a. Create an instance of Datastore<Preferences> using the property delegate created by preferencesDataStore at the top of your Kotlin file so you can access your datastore throughout your application as a singleton.
        i. Context is needed to create a DataStore

    b. DataStore doesn't use plain strings as keys, you need to define a preference key for each value you're storing in the DataStore
    c. The edit() function updates the data in a DataStore
        i. The transform parameter takes a lambda expression that transactionally updates the state in DataStore
        ii. edit() is a suspend function so it must be called from a CoroutinesContext
    d. You can read data using the .data property which returns a Flow<Preference> instance
        i. Use .map to map the Flow<Preference> instance using the preference keys to get the data and then create an instance of your model class.
        ii. Data is retrieved on Dispatchers.IO so your UI thread isn't blocked.
            1. Dispatchers.IO is a Kotlin CoroutineDispatcher made available through Kotlin coroutines
        iii. Can be converted to LiveData using the asLiveData() extension and then observed in your Activity class
2. Proto DataStore to store data as instances of a custom data type using protocol buffers for structured data
    a. Define a class that implements Serializer<T> (T is the type defined in the proto file) to tell DataStore how to read and write your data type
    b. Create an instance of Datastore<T> using the property delegate created by dataStore at the top of your Kotlin file so you can access your datastore throughout your application as a singleton.
    c. The updateData() function updates a stored object
        i. The transform parameter takes a block of code where you update the data
    d. You can read data using the .data property which returns a Flow<Preference> instance
        i. Use .map to map the Flow<Preference> instance using the preference keys to get the data and then create an instance of your model class.

Both DataStore implementations store data asynchronously, consistently, and transactionally, overcoming most of the drawbacks of SharedPreferences. DataStore uses coroutines and Flow to store data asynchronously.

<u>Coroutines</u>
Kotlin coroutines handle concurrency in Kotlin and can be used in Android to simplify asynchronous code and solve to main issues:

1. Long running tasks are tasks that take too long such as network calls or reading/writing from a database and would block the main thread.
2. Main-safety ensures that functions don't block UI updates on the main thread
3. allows you to ensure that any suspend function can be called from the main thread.

Coroutines provide similar functionality to async/await used in other languages like Swift. A coroutine is conceptually similar to a thread in that it takes a block of code to run that works concurrently with the rest of the program. Coroutines are described as lightweight threads because creating a coroutine doesn't allocate a new thread and is not bound to any particular thread. It may suspend its execution in one thread and resume in another one.

- Coroutines can be suspended and resumed without blocking a thread.
    - Suspending vs blocking
        - If you make a blocking call on the main thread's execution, you effectively freeze the UI

        - If you call a suspending function in the main UI thread it can be run on a different thread so it won't block the main UI thread.
        - (images) https://www.raywenderlich.com/1423941-kotlin-coroutines-tutorial-for-android-getting-started#toc-anchor-004
    - A suspending function is simply a function that can be paused and resumed at a later time. They can execute a long running operation and wait for it to complete without blocking.
    - Suspending functions can only be called by another suspending function or within a coroutine.
        - CoroutineScopes define the scope when a coroutine runs and handles their lifecycle
        - Android Kotlin extensions includes built-in coroutine scopes for their lifecycle-aware components
            - ViewModelScope
                - defined for each ViewModel in your app. Any coroutine launched in this scope is automatically canceled when the ViewModel is cleared.
            - LifecycleScope
                - defined for each LifeCycle object. Any coroutine launched in this scope is automatically canceled if the LifeCycle is destroyed.
    - A coroutine can be run using different CoroutineDispatchers. These dispatchers are available through the API:
        - Dispatchers.Default
            - CPU-intensive work, such as sorting large lists or doing complex calculations
        - Dispatchers.IO
            - any input and output such as networking or reading and writing from files
        - Dispatchers.Main
            - recommended dispatcher for performing UI-related events such as showing lists in a RecyclerView or updating Views

Flows are built on top of coroutines and can provide multiple values. A flow is conceptually a *stream of data* that are being asynchronously computed. `Flow<T>` is a flow that emits values of type T.

The following dependencies are needed for DataStore.
Datastore Preferences:
```
implementation "androidx.datastore:datastore-preferences:1.0.0"
```

Datastore Typed:
```
implementation "androidx.datastore:datastore:1.0.0"
```

Although Shared Preference is still available, DataStore is the suggested library moving forward.

Music
Create a new project
Empty Activity template
Name: Music
Package name: the fully qualified name for the project
Language: Kotlin

Minimum SDK: API 21 (21 is the minimum API for Material Design)
Leave legacy libraries unchecked (using these restricts usage of some newer capabilities)
Finish

Add the dependencies for viewmodel, livedata, and DataStore Preferences in the app's grade file.

```
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1"
implementation "androidx.lifecycle:lifecycle-livedata-ktx:2.4.1"
implementation "androidx.datastore:datastore-preferences:1.0.0"
```

Add colors, create a theme with a Material Components parent theme, and apply it to your app.

Layout
In the layout file set up a TextView (can reuse the one already there) and EditText for favorite song with the id editTextSong. Add another TextView and EditText for favorite artist with the id editTextArtist.
I set the TextView's textAppearance to
`@style/TextAppearance.MaterialComponents.Headline6`
I created string resources to use for the EditText hint property.

```
<string name="songText">Favorite song</string>
<string name="artistText">Favorite artist</string>
```

I created a horizontal chain for each TextView EditText pair with style spread and a margin between them.
I added start and top constraints and aligned vertical centers to make sure they were lined up.

Also add a Floating Action Button to the bottom right and set its drawable to the android image ic_menu_save. I changed the id for fab. You should also add a string resource for the contentDescription.

```
<string name="fabSave">save</string>
```

Kotlin class
We're going to create a custom Kotlin class to represent the data we'll be using.
First create a new package for our data related classes. In the java folder select the music folder (not androidTest or test)
File | New | Package
Name: data

Then select the new data package and add a new class called Favorite.
File | New | Kotlin class
Name: Favorite
Kind: Class

The Favorite class will be a data class that includes song and artist, both Strings.
We'll define these properties in the primary constructor in the class header.

```
data class Favorite (val song: String, val artist: String) {
}
```

In our data package we'll create a Kotlin class to store all the logic for writing and reading DataStore preferences.

Select the data package and add a new class called DataStoreRepo.
File | New | Kotlin class
Name: DataStoreRepo
Kind: Class

The class has a property dataStore of type DataStore<Preferences> that is passed into its constructor. In DataStore we can't use a String as a key, we need to create preferences keys. We do that here by creating an object called PreferencesKeys with two properties, one for each preferences keys. We use the stringPreferencesKey() method and pass the name of the key as the name parameter.

```
class DataStoreRepo(private val dataStore: DataStore<Preferences>) {
    private object PreferencesKeys {
        val SONG = stringPreferencesKey("song")
        val ARTIST = stringPreferencesKey("artist")
    }
}
```

To read our data we retrieve it using dataDtore.data as a Flow<Favorite> object.
IOExceptions may get thrown when an error occurs while reading data from preferenceDataStore. It's best practice to handle these exceptions using catch before applying map operator
If an IOException is encountered we throw emptyPreferences else we re-throw the exception raised.
Map() returns a Flow which contains the results of applying the given function to each value of the original Flow. We use the preference keys to get the data. If the key isn't set when you read the data it returns null. We use the Elvis operator to return an empty String instead. We use the data to create an instance of the Favorite model class.
Later we'll convert the Flow object to LiveData in our ViewModel.

```
    val readFromDataStore: Flow<Favorite> = dataStore.data
        .catch { exception ->
            if (exception is IOException) {
                Log.d("DataStoreRepository",
exception.message.toString())
                emit(emptyPreferences())
            } else {
                throw exception
            }
        }
        .map { preference ->
            val song = preference[PreferencesKeys.SONG] ?: ""
            val artist = preference[PreferencesKeys.ARTIST] ?: ""
            Favorite(song, artist)
        }
```

To save to DataStore we call edit() which is a suspend and extension function on DataStore. Edit() takes in a lambda expression for its transform parameter that uses the preferences keys we just defined to write to the datastore. Since it's a suspend function, when called it suspends the current coroutine until the data persists to disk. When complete DataStore.data reflects the change.

```
    suspend fun saveToDataStore(song: String, artist: String){
        dataStore.edit { preference ->
            preference[PreferencesKeys.SONG] = song
            preference[PreferencesKeys.ARTIST] = artist
        }
    }
```

ViewModel

Now we'll create our view model class.
Select the data package and add a new class called MovieViewModel.
File | New | Kotlin class
Name: FavoriteViewModel
Kind: Class

Our FavoriteViewModel class has a property of type DataStoreRepo that is passed into its primary constructor. FavoriteViewModel subclasses the ViewModel class.

Because I want the view model to be the single point of contact for data activities, I'll add logic to our FavoriteViewModel class to call the methods in our DataStoreRepo class.

We define a property called favorites which will read from our datastore. asLiveData() converts the Flow to LiveData so we can observe it later in our Activity.

```
class FavoriteViewModel(private val dataStoreRepo:
DataStoreRepo):ViewModel() {

    val favorites = dataStoreRepo.readFromDataStore.asLiveData()
}
```

We then create a method to save our data that takes in a favorite song and favorite artist. Because the saveToDataSource() method in the DataStoreRepo class is a suspended method, we need to call them from inside a coroutine scope. We use the viewModel scope and run it in the Dispatchers.IO coroutine dispatcher.

```
    fun saveFavorites(songFavorite: String, artistFavorite: String){
        viewModelScope.launch(Dispatchers.IO) {
            dataStoreRepo.saveToDataStore(songFavorite,
artistFavorite)
        }
    }
}
```

The last thing we need in our ViewModel is a factory class that will create our instance of FavoriteViewModel. We have been using "by viewModels()" but that only works for view model classes that have no parameters. Since our view model takes in a DataStoreRepo as a parameter we need this helper class to instantiate it.

```
class FavoriteViewModelFactory(private val dataStoreRepo:
DataStoreRepo): ViewModelProvider.Factory{
    override fun <T: ViewModel> create(modelClass: Class<T>):T{
        return FavoriteViewModel(dataStoreRepo) as T
    }
}
```

MainActivity

In MainActivity.kt we create our datastore at the top level of our Kotlin file, before the MainActivity class definition.

```
private val Context.dataStore by preferencesDataStore(name =
"favorites")
```

preferencesDataStore creates a property delegate for a single process DataStore. This should only be called once in a file, and all usages of the DataStore should use a reference to the same instance.

In onCreate() in MainActivity we need to do the following:

- define our viewModel instance
- set up a click listener on the FAB so when the click event fires we get the values from the EditTexts and save them using our view model saveFavorites() method. I also added a Snackbar for some visual confirmation that the entered data has been saved. You need this string resource.

```
<string name="favsSaved">Favorites saved </string>
```

- we need to observe our view model's favorites property, which is LiveData, so that when the data changes we update the UI. The onChange event will fire when the data is read from our datastore when the activity is launched so then our UI will be updated accordingly.

```
private val Context.dataStore by preferencesDataStore(name =
"favorites")

class MainActivity : AppCompatActivity() {
    private lateinit var viewModel: FavoriteViewModel
    private lateinit var song: TextView
    private lateinit var artist: TextView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        viewModel = ViewModelProvider(this,

FavoriteViewModelFactory(DataStoreRepo(dataStore)))[FavoriteViewModel
::class.java]

        song = findViewById(R.id.editTextSong)
        artist = findViewById(R.id.editTextArtist)
```

```
findViewById<FloatingActionButton>(R.id.fab).setOnClickListener{ view
->
            val mySong = song.text.toString()
            val myArtist = artist.text.toString()
            viewModel.saveFavorites(mySong, myArtist)
            Snackbar.make(view, R.string.favsSaved,
Snackbar.LENGTH_LONG).show()
        }

        viewModel.favorites.observe(this, Observer { favorites ->
            song.text = favorites.song
            artist.text = favorites.artist
        })
    }
}
```

Now when you test your app you should see your favorites 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).