# Jetpack DataStore Part-1

▼ Why Jetpack data store?

data that we store in the jetpack data store can not be saved in Room ORM because SQLite is designed to store a large amount of structured data so is not a suitable option

it neither should not be saved by shared preferences because it was deprecated since API level 29 and one of its most important problems was that it runs on the main thread so it blocks the UI.

Jetpack data Store works on the background thread and uses Flow.

▼ how it can be used

- make a Kotlin class and name it `PreferencesManager`

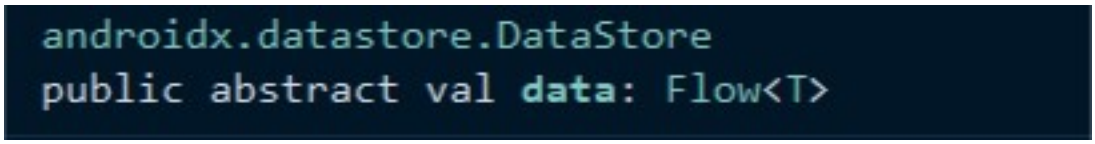
1. we use Hilt to add dependency injection to the project it gives us the constructors which we need for the class and it also injects `PreferenceManager` into the view model which we will discuss later.

```
@Singleton
classPreferencesManager @Injectconstructor(@ApplicationContext context: Conte
xt) {
```

2. we used the Singleton annotation in order to avoid duplication in our code and to optimize it now only for the first time that we need an instance of the class it will be made then it just would be reused. we also passed `context` by this `@ApplicationContext` annotation because we are going to need it later.
3. datastore should be created and named now so we can put our data in.

```
private val dataStore = context.createDataStore("user_preferences")
```

4. datastore can give us a flow of current settings which in our case would be the states of lights and doors. the syntax of this part can ruin the clarity of our view model so we do these operations here.




```
val preferencesFlow = dataStore.data
```

5. the map operator should be added to this peace of code so we can make changes to the values that come through this flow.

```
val preferencesFlow = dataStore.data
        .map { preferences ->
         //configurations
        }
```

6. there are some steps left before adding anything in the map operator one of them is adding an enum class to this file so we access the states of AC.

```
enum class AirConditioner {HEATER, COOLER}
```

7. after that we should prepare preferences keys.

`AirConditioner` is an enum and we just can store primitive types in these variables so we will have to cast them to string.

```
private object PreferencesKeys{
val AC = preferencesKey<String>("AIR_CONDITIONER")
val LIGHTS = preferencesKey<Boolean>("LIGHTS") //"LIGHTS" is just a name
//the object is just a name space which is going to make the code more clear
```

8. let's get back to our `dataStore` it might be the most important part so I will take the code under the microscope.

```
val preferencesFlow = dataStore.data
.map { preferences ->
//configurations
}
```

```
val preferencesFlow = dataStore.data
        .catch { exception ->
            if (exception is IOException) {
                Log.e(TAG, "Error reading preferences", exception)
                emit(emptyPreferences())
            } else {
                throw exception
            }
        }
        .map { preferences ->

            val myAirConditioner = AirConditioner.valueOf(
                preferences[PreferencesKeys.AC] ?: AirConditioner.COOLER.name
            )
            val myLights = preferences[PreferencesKeys.LIGHTS] ?: true
            FilterPreferences(myAirConditioner, myLights)
        }
```

but what does that even mean?

## it is going to look like this

```
val preferencesFlow = dataStore.data
        .catch { exception ->
            if (exception is IOException) {
                Log.e(TAG, "Error reading preferences", exception)
                emit(emptyPreferences())
            } else {
                throw exception
            }
        }
        .map { preferences ->

            val myAirConditioner = AirConditioner.valueOf(
                preferences[PreferencesKeys.AC] ?: AirConditioner.COOLER.name
            )
            val myLights = preferences[PreferencesKeys.LIGHTS] ?: true
            FilterPreferences(myAirConditioner, myLights)
        }
```

## lets just concentrate on this part

```
val preferencesFlow = dataStore.data

        .map { preferences ->

        }
```

map operator returns preferences and lets you make changes to them which in our case the change would be casting enum to string

as I said before we can only store primitive types so enum should be converted to string

this part takes our preference key and returns its value so it can be stored in the variable

in case the preference was null the default value will be AirConditioner.COOLER

does the same thing but because the value is not enum it doesn't need to change

it turns enums to string

```
val preferencesFlow = dataStore.data
        .map { preferences ->
            val myAirConditioner = AirConditioner.valueOf(
                preferences[PreferencesKeys.AC] ?: AirConditioner.COOLER.name
            )
            val myLights = preferences[PreferencesKeys.LIGHTS] ?: true

        }
```

there is still a problem
map operator can
not return two values

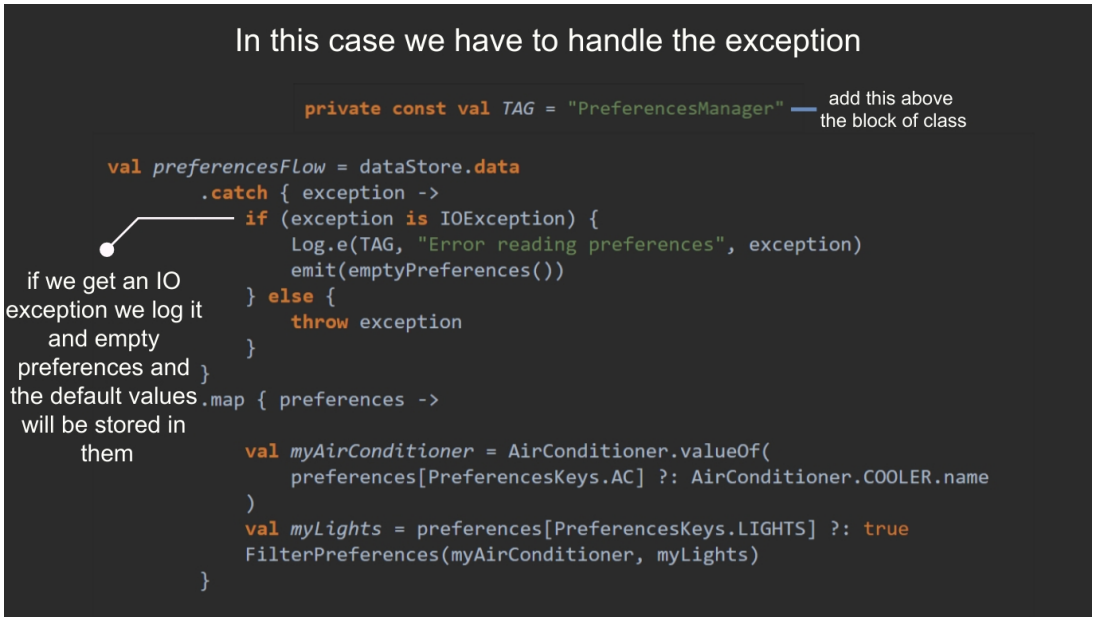
```
data class FilterPreferences(val ac: AirConditioner, val lamps: Boolean)
```

so we have to add this data class outside of our preferences manager class in order to store our variables in

```
val preferencesFlow = dataStore.data
        .map { preferences ->
            val myAirConditioner = AirConditioner.valueOf(
                preferences[PreferencesKeys.AC] ?: AirConditioner.COOLER.name
            )
            val myLights = preferences[PreferencesKeys.LIGHTS] ?: true
            FilterPreferences(myAirConditioner, myLights)
        }
```

and finally .map looks as good as it should

BUT WHAT IF
SOMETHING
GOES
WRONG?!!!




 we are almost done with `PreferencesManager` class there is just one step left

we need two functions to call from our view model and change the states of our preferences

```
suspend fun updateAirConditionerState(acState: AirConditoner) {
        dataStore.edit { preferences ->
            preferences[PreferencesKeys.AC] = acState.name
        }
    }
```

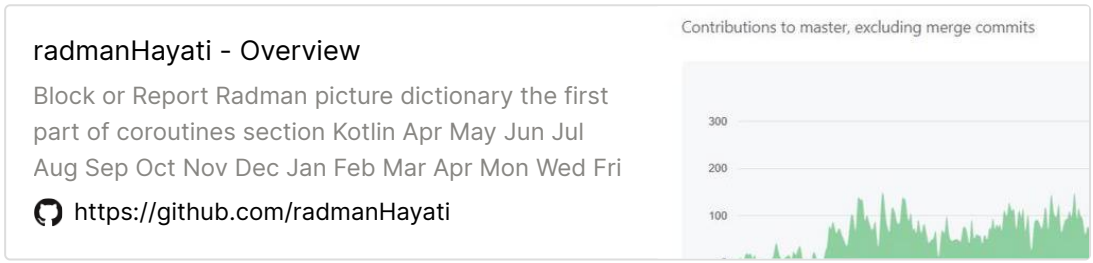
```
suspend fun updateLightsStare(lightsON: Boolean) {
        dataStore.edit { preferences ->
            preferences[PreferencesKeys.LIGHTS] = lightsON
        }
    }
```

`datastore.edit` lets you update the determined preference which will be located `preferences[PreferencesKeys.here]`

 `PreferencesManager` class is finally ready.

This class will be used in the next part where we change the states from our view model.

You can access the source of this project from my Github.




Thanks for your time - **Radman Hayati**