Task P – Spike: Core 3 – The Clubhouse

GitHub repo: https://github.com/SoftDevMobDevJan2024/core3-104222196

Goals:

This section is an overview highlighting what the task is aiming to teach or upskill.

Building an app that reads data from a read-only file and displays it as a list with RecyclerView. Working with lists and menus.

The following list outlines the goal broken down into more specific knowledge gaps involved in the goal.

- Reading data from read-only CSV file.
- Using list methods for sorting and filtering.
- Working with RecyclerView
    - Writing a custom Adapter to populate the list
- Adding a top menu bar for the app

Tools and Resources Used

This section lists related software, tools, libraries, API's, and other resources used for this knowledge gap.

- Android Studio
- Resources.openRawResource method: https://developer.android.com/reference/android/content/res/Resources#openRawResource(int)
- BufferedReader: https://developer.android.com/reference/java/io/BufferedReader
- List sortedBy method: https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/sorted-by.html
- List filter method: https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/filter.html
- How to work with RecyclerView: https://developer.android.com/develop/ui/views/layout/recyclerview
- How to set up the app bar: https://developer.android.com/develop/ui/views/components/appbar/setting-up
- How to add and handle actions of the app bar: https://developer.android.com/develop/ui/views/components/appbar/actions
- notifyDataSetChanged method: https://developer.android.com/reference/androidx/recyclerview/widget/RecyclerView.Adapter#notifyDataSetChanged()

Knowledge Gaps and Solutions

This section presents the listed knowledge gaps and their solutions with supporting images, screenshots and captions where appropriate/required.

Core 3 Questions

- **What are the advantages of RecyclerView over ListView?**
  RecyclerView is more efficient than ListView because it reuses rows as the users scroll the list. This saves Views from being repeatedly destroyed and recreated, which is resource-intensive. Moreover, RecyclerView uses ViewHolders to hold View objects and reduces the number of expensive findViewById calls.

- **Why are you unable to write to the (CSV) file, and what would you need to do to be able to add any new data to the file?**
  Because the CSV is located in the “res” folder, which is read-only. To add new data to the file, it would have to be located in either internal or external storage.

- **When filtering the data, what changes did you make from a RecyclerView with unchanging data?**
  My code used RecyclerView.Adapter instead of ListAdapter so to filter the data and update the list, I had to update the original list object passed into the adapter and notify the adapter of this change through adapter.notifyDataSetChanged(). The drawback to this approach is that the code to filter the data set runs on the main thread, which can slow down the app if the data is large. In addition, using

notifyDataSetChanged is inefficient as the adapter will think that the entire data set has changed and unnecessarily update all the visible views. It would be better to inform the adapter of changes at specific indexes, but unfortunately that would be a hassle to implement as there are so many items being added/removed.

Gap 1: Reading data from CSV file

1. To add the CSV file to the app, put it in the “res/raw” folder. This folder holds files in their raw form. For this data set, each row contains 3 pieces of information: the club’s name, the club meeting location, and the meeting time (represented as a UNIX timestamp). The rows of the CSV file look like this:

```
Cheerleading,Community Room,1711737756000
Cheerleading,Online,1711607213000
Bookworm,Online,1711467367000
ESports,Meeting Room,1711410372000
... more rows below
```

2. To represent each row as an object in the application, we define the following data class:

```
data class Meeting(
    val club: String,
    val location: String,
    val time: Date
)
```

3. Define a singleton object DataModel with a single method readCsvFile which takes as input the application’s resources and the resource ID of the CSV file and returns a list of Meetings. This method will be called in MainActivity.

```
object DataModel {
    fun readCsvFile(resources: Resources, @RawRes resourceId: Int):
List<Meeting> {
        ...
    }
}
```

4. Inside the method, open the resource file as an InputStream, create a BufferedReader for it, read each line until the end of the file and convert each line into a Meeting object.

```
// Opens the CSV file as an input stream.
val inputStream = resources.openRawResource(resourceId)
// Creates a reader object for the input stream.
val reader = BufferedReader(InputStreamReader(inputStream))

val meetings = mutableListOf<Meeting>()
var line: String?

// Reads the file line by line until the end of the file where line is null.
do {
    line = reader.readLine()
    // If line is not null, convert the line into a Meeting object.
    if (line != null) {
        val columns = line.split(",")
        meetings.add(
            Meeting(
                club = columns[0],
                location = columns[1],
                time = Date(columns[2].toLong())
            )
        )
```

```
    }
} while (line != null)

return meetings
```

5. Inside MainActivity, we can call this method to obtain the list of meetings. We can also sort the meetings by time in ascending order with this code:

```
// Reads the meeting data from the CSV file and sorts it by time in ascending
order.
allMeetings = DataModel.readCsvFile(resources, R.raw.groups).sortedBy { it.time
}
```

Gap 2: Using list methods for sorting and filtering

1. This application requires the use of list methods to sort and filter lists. As demonstrated in the previous section, one of these methods is sortedBy:

```
allMeetings = DataModel.readCsvFile(resources, R.raw.groups).sortedBy { it.time
}
```

This method takes a function that is called for each element in the list. In this case, the function returns the "time" field of each meeting, which will be used to order the list. The output of sortedBy is a new list.

2. Another relevant method is "filter". It takes a predicate function that returns true or false and is called for every element in the list. Elements that have the function return true are included in the resulting list.

```
allMeetings.filter { it.club == club }
```

The filter method returns a new list.

Gap 3: Working with RecyclerView

1. Just like TextView or ImageView, RecyclerView is a View and needs to be added into the layout file of the activity.

```
<androidx.recyclerview.widget.RecyclerView
    android:id="@+id/meetingList"
    android:layout_width="match_parent"
    android:layout_height="0dp"
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
    app:layout_constraintTop_toBottomOf="@+id/toolbar"
    tools:layout_editor_absoluteX="-1dp" />
```

2. Inside the activity, the RecyclerView can be referenced with:

```
val recyclerView = findViewById<RecyclerView>(R.id.meetingList)
```

3. The RecyclerView needs to be associated with an adapter. We will use a custom adapter extending the RecyclerView.Adapter class:

```
class MeetingListAdapter(private val dataset: List<Meeting>) :
RecyclerView.Adapter<MeetingListAdapter.ViewHolder>() {
    ... implementation goes here
}
```

The custom adapter holds a reference to a list of Meetings.

4. Inside the custom adapter, define a ViewHolder class that is a wrapper around a View:

```
class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {
    val clubNameView = view.findViewById<TextView>(R.id.clubName)
    val locationView = view.findViewById<TextView>(R.id.location)
    val timeView = view.findViewById<TextView>(R.id.time)
    val iconView = view.findViewById<ImageView>(R.id.typeIcon)
}
```

5. The layout of the View that ViewHolder wraps around is defined in an XML file just like the app layout. In this case, that View has the following components:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
xmlns:android="http://schemas.android.com/apk/res/android" ...>

    <TextView
        android:id="@+id/clubName" .../>

    <TextView
        android:id="@+id/location" .../>

    <TextView
        android:id="@+id/time" .../>

    <ImageView
        android:id="@+id/typeIcon" .../>
</androidx.constraintlayout.widget.ConstraintLayout>
```

6. The custom adapter has to override 3 key methods of the base Adapter class: onCreateViewHolder, which is called when the ViewHolder is created, onBindViewHolder, which is called when data is attached to the ViewHolder, and getItemCount, which returns the size of the data set.

```
// Called when a view holder is created.
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {
    val view = LayoutInflater.from(parent.context).inflate(R.layout.meeting_row,
parent, false)
    return ViewHolder(view)
}

// Binds data to a view holder.
override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    holder.clubNameView.text = dataset[position].club
    holder.locationView.text = dataset[position].location
    holder.timeView.text = dataset[position].time.toLocaleString()

    // If the meeting location is "Online", draws a computer icon in the meeting
row item.
    if (dataset[position].location == "Online") {
        holder.iconView.setImageResource(R.drawable.computer_24px)
    } else {
        holder.iconView.setImageDrawable(null)
    }
}

// Returns the size of the dataset.
override fun getItemCount(): Int {
    return dataset.size
}
```

The onCreateViewHolder creates a ViewHolder object with a View object inflated from the XML file above. The onBindViewHolder method updates the text and icon of the Views inside ViewHolder.

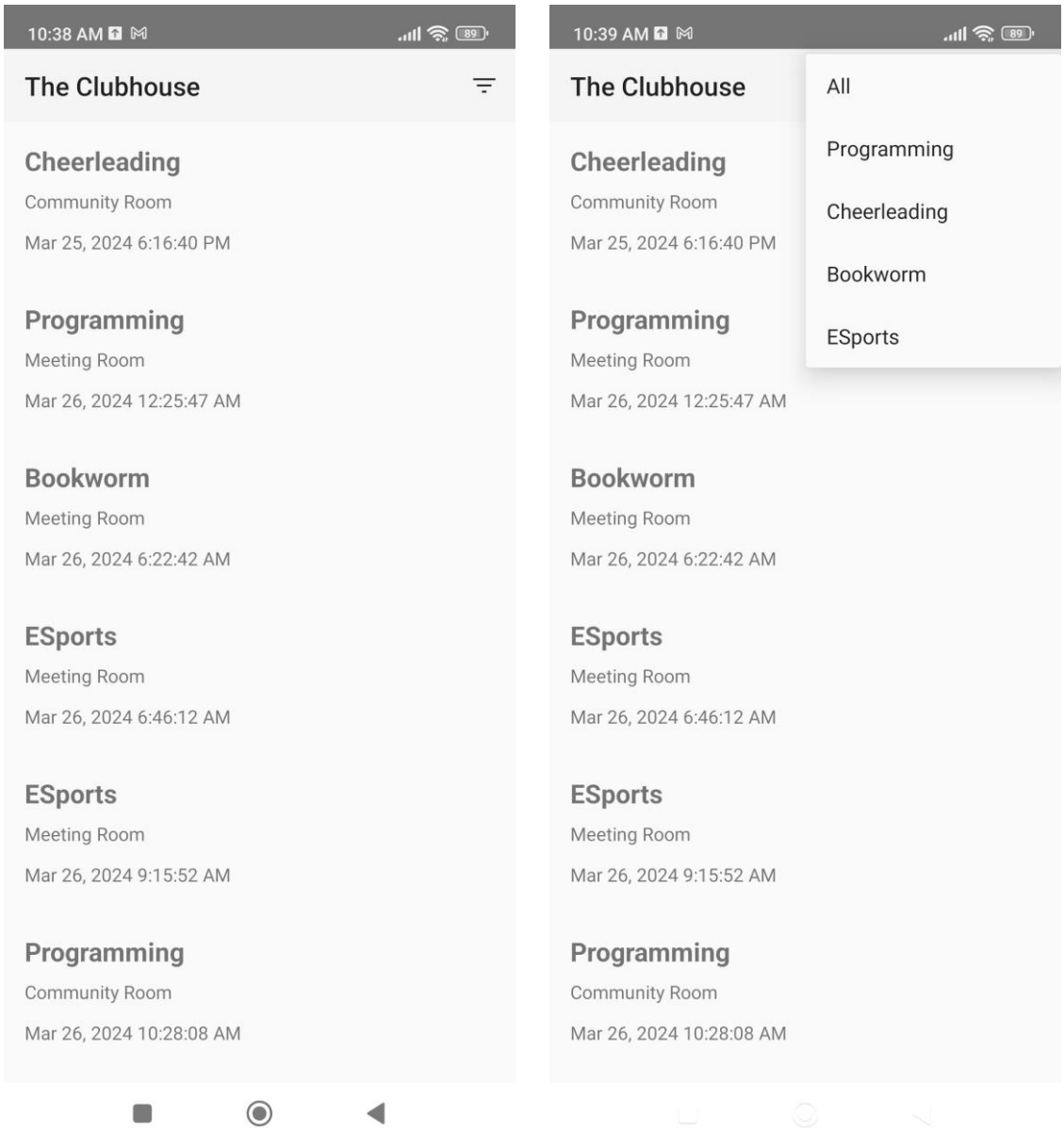
7. Inside the activity, the custom adapter can be instantiated and associated with a RecyclerView with the following code:

```
// Creates a mutable copy of the allMeetings list to pass into the adapter.
meetingsToShow = allMeetings.toMutableList()
// Creates a new ListAdapter object, passing in the mutable array of meetings to
show.
meetingListAdapter = MeetingListAdapter(meetingsToShow)
// Associates the adapter with the recycler view.
recyclerView.adapter = meetingListAdapter
```

meetingsToShow is a mutable list of meetings. The reason it is mutable is that we will have to update it later to filter the list.

Gap 4: Adding a top menu bar and the filter button

1. The app will have a top bar with a menu option to filter the list of meetings. There will be a filter button and when pressed on, the user will be presented with a list of club names to filter by.



2. To add this top bar, add a Toolbar widget to the activity's layout.

```
<androidx.appcompat.widget.Toolbar
    android:id="@+id/toolbar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:background="?attr/colorPrimary"
    android:minHeight="?attr/actionBarSize"
    android:theme="?attr/actionBarTheme"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

3. Inside the activity's onCreate method, set this Toolbar as the app's Action Bar.

```
// Sets the Toolbar as the top app bar.
setSupportActionBar(findViewById(R.id.toolbar))
```

4. To add menu items to the top bar, we need to define them in an XML file first. We will add a menu.xml file inside the res/menu folder with the following contents:

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_filter"
        android:icon="@drawable/filter_list_24px"
        android:title="@string/filter_meetings"
        app:showAsAction="always">
        <menu>
            <item android:id="@+id/action_filter_none"
                android:title="All"
                app:showAsAction="never" />
            <item android:id="@+id/action_filter_programming"
                android:title="Programming"
                app:showAsAction="never" />
            <item android:id="@+id/action_filter_cheerleading"
                android:title="Cheerleading"
                app:showAsAction="never" />
            <item android:id="@+id/action_filter_bookworm"
                android:title="Bookworm"
                app:showAsAction="never" />
            <item android:id="@+id/action_filter_esports"
                android:title="ESports"
                app:showAsAction="never" />
        </menu>
    </item>
</menu>
```

The menu will show a Filter button with a Filter icon. When pressed on, the button will open a list of options to filter by.

5. To add these menu items to the top bar, override the onCreateOptionsMenu method of the activity and write the following code:

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
    menuInflater.inflate(R.menu.menu, menu)
    return true
}
```

6. To make the menu items do something when the user presses on them, override the onOptionsItemSelected method:

```
override fun onOptionsItemSelected(item: MenuItem): Boolean {
    when (item.itemId) {
        R.id.action_filter_none -> filterMeetings("All")
        R.id.action_filter_programming -> filterMeetings("Programming")
        R.id.action_filter_cheerleading -> filterMeetings("Cheerleading")
        R.id.action_filter_bookworm -> filterMeetings("Bookworm")
        R.id.action_filter_esports -> filterMeetings("ESports")
    }
    return true
}
```

This method is called every time the user presses on an item in the menu. It checks for the ID of the selected item and runs the correct filter option.

7. The filterMeetings method is defined as follows:

```
private fun filterMeetings(club: String) {
    meetingsToShow.clear()
```

```
    if (club == "All") {
        meetingsToShow.addAll(allMeetings)
    } else {
        meetingsToShow.addAll(allMeetings.filter { it.club == club })
    }

    meetingListAdapter.notifyDataSetChanged()
}
```

This method mutates the mutable meetingsToShow list that was assigned to the Adapter. It then notifies the adapter that its data set has changed with notifyDataSetChanged().

Open Issues and Recommendations

This section outlines any open issues, risks, and/or bugs, and highlights potential approaches for trying to address them in the future.

Issue: Inefficient list filtering

The current solution uses RecyclerView.Adapter as the base class of the custom adapter, and the data set is updated through calculations on the main thread. This can affect performance if the data set is sufficiently large. In addition, the use of notifyDataSetChanged will also impact performance because the adapter will think that the entire data set has changed and update all the visible views, even those that have not changed. A potential solution to this is to use the ListAdapter class instead, which is more suitable for dynamic lists.