Task ##P/C – Spike: Instrument Rental Application

Repository: https://github.com/hakl04/MobileAppDev/tree/main/Assignment2

Goals:

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

Insert summary here:

The project's objective is to create a basic proof-of-concept mobile application for a music studio that lets customers rent musical instruments and equipment on a monthly basis. The app will display main information like name, rating, multi-choice attributes, and monthly "price" in credits and will display three to four rental goods one at a time. After choosing an item, users will be taken to a detailed view for booking. They will use a "next" button to travel among the items. User-friendly design features will be incorporated into the application, such as Chip and Ratingbar widgets for attribute selection and necessary field error checking. It will incorporate visual feedback (such as Toast or Snackbar) for booking actions and use Parcelable objects to make data transfer between activities easier. User stories, use cases, screen layout drawings, and a report outlining technical implementations and design decisions will all be included in the final deliverable.

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

- Model and Data Binding
- Android Studio Style
- UI UX Design
- Using Chip and ChipGroup
- Parcelable Object
- Rental App Logic

Project Plan

ID	Tasks	Expected Complete Date	Status
1	Planning the project	18/02/2025	Completed
2	Draw drafts for the user interfaces	20/02/2025	Completed
3	Implement the core application logic of Main Activity and Finish designing app UI	25/02/2025	Completed
4	Implement the designed application UI into the project xml layouts	26/02/2025	Completed
5	Implement data class Instrument and implement data binding	28/02/2025	Completed
6	Write the core logic for Rent Activity	02/03/2025	Completed
7	Implement sound effect	02/03/2025	Completed
8	Implement Parcelable Instrument class	04/03/2025	Completed
9	Learn how to work with Chip and ChipGroup	05/03/2025	Completed
10	Implement Chip and ChipGroup for instrument accessories	06/03/2025	Completed
11	Write The Goal and Project Plan for the assignment report	08/03/2025	Completed
12	Finish all remaining application feature	09/03/2025	Completed
13	Write the Issues and recommendation and Resource Used	09/03/2025	Completed
14	Write the Learning Gap for the assignment report	10/03/2025	Completed
15	Creating test cases for the project and run them	11/03/2025	Completed

16	Fix any remaining bug and project finalisation	15/03/2025	Completed
17	Finalise the report	16/03/2025	Completed
18	Submit the assignment	16/03/2025	Completed

Tools and Resources Used

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

- Android Studio: The IDE used to develop the application
- https://developer.android.com/topic/libraries/architecture/saving-states?hl=en
- https://stackoverflow.com/questions/4384890/how-to-disable-an-android-button
- https://www.youtube.com/watch?v=klazwO0RbJQ

Knowledge Gaps and Solutions

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

Gap 1: Model and Data Binding

((Insert steps required to address his knowledge gap))

The first step was to create a data class for the instruments. The class will hold all data needed for the instruments in the app. These includes the instrument's name, image, cost per month, rented month count, rating, description and accessories.

```
data class Instrument (
    var name : String,
    var image : Int,
    var cost : Int,
    var months : Int,
    var rating : Float,
    var description : String,
    var accessories: Map<String, Int>
```

We can use data binding to dynamically bind values rather than setting text for TextView with a hard value as is typical. To show the Instrument object's rating property, I'm using data binding. The rating value is bound to the TextView by setting the android:text attribute to @{String.valueOf(instrument.rating)}. This implies that the text displayed in the TextView will immediately update to reflect any changes made to the instrument object's rating property. The rating, which is a Float, is converted into a string for correct display thanks to the String.valueOf() method.

```
<TextView
    android:id="@+id/textView2"
    style="@style/secondarytext"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    android:text="@{String.valueOf(instrument.rating)}" />
```

I'm initializing the data binding and setting the content view of my Activity using the DataBindingUtil class. This method generates an instance of the generated binding class, which is assigned to the binding variable, in addition to inflating the layout specified in activity_main.xml.

```
binding = DataBindingUtil.setContentView( activity: this, R.layout.activity_main)
//initialize the binding that control the instrument being displayed
```

I can set the instrument of the binding. Whenever I do this, the Views with data binding values like the rating above will automatically update its value.

```
binding.instrument = instruments[instrumentIndex]; //set the instrument being displayed
```

Gap 2: Android Studio Style

((Insert steps required to address his knowledge gap))

To customize the look of Chip components in my Android application, I have a style definition in style.xml. Each item in this style specifies a specific property for the Chip, such as setting android:layout_width to match_parent to make the Chip fill the width of its parent view, setting android:layout_height to 60sp to define its height, setting android:textSize to 24sp to make the text readable, and setting android:textAlignment to center to center the text within the Chip.

```
<style name="chip">
    <item name="android:layout_width">match_parent</item>
    <item name="android:layout_height">60sp</item>
    <item name="android:textSize">24sp</item> <!-- Set text size -->
    <item name="android:textAlignment">center</item>
    <item name="android:drawablePadding">0dp</item>
    <item name="chipStrokeColor">@android:color/transparent</item> <!-- Remove border -->
    <item name="android:textColor">@color/textlight</item> <!-- Change text color -->
    <item name="chipBackgroundColor">@color/unchecked</item> <!-- Set initial background color -->
</style>
```

To apply the visual properties I previously set, I'm using the custom style defined in style.xml with style="@style/chip" in this Chip element. The android:id attribute gives the Chip an ID so I can use it in my code.

```
<com.google.android.material.chip.Chip
    android:id="@+id/chip"
    style="@style/chip"
    android:checkable="true"
    android:text="@{instrument.accessories.keySet().toArray()[0]}" />
```

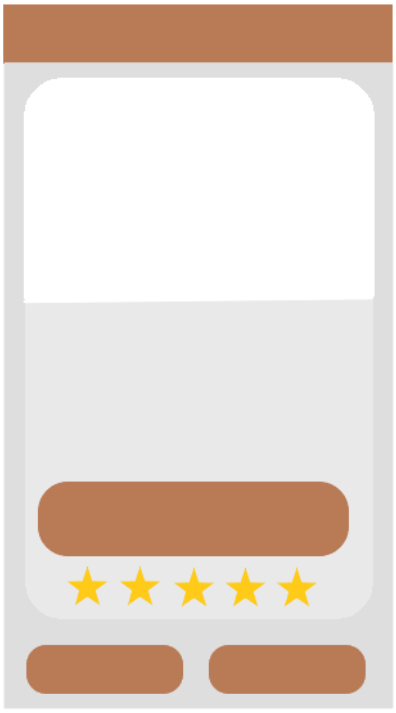
Gap 3: UI/UX Design

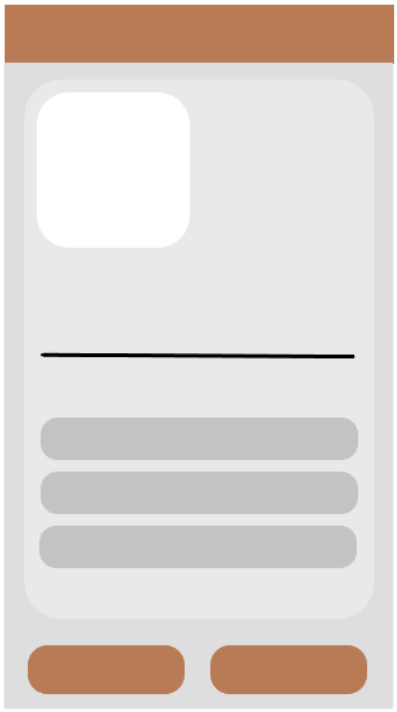
With the portrait/ landscape switching feature solution mentioned in the previous knowledge gap, when the app orientation is changed from portrait to landscape and vice versa, the states of the app like current hold

and current score will be reset. We do not want this to be the case because the score and hold should be consistent even after screen rotation.

((Insert steps required to address his knowledge gap))

The first step is to design the user interface by sketching. I did research on mobile app interface, particularly rental application user interface, and come up with these 2 designs for the 2 layout the app will have.





User Stories:

- As a musician, I want to cycle through available instruments and see their main information like name, rating and cost per month so that I can find the right one for my needs.
- As a musician, I want to be able to choose accessories accompanying my rented instruments and see the cost of those accessories per month as well as the total price of the booking

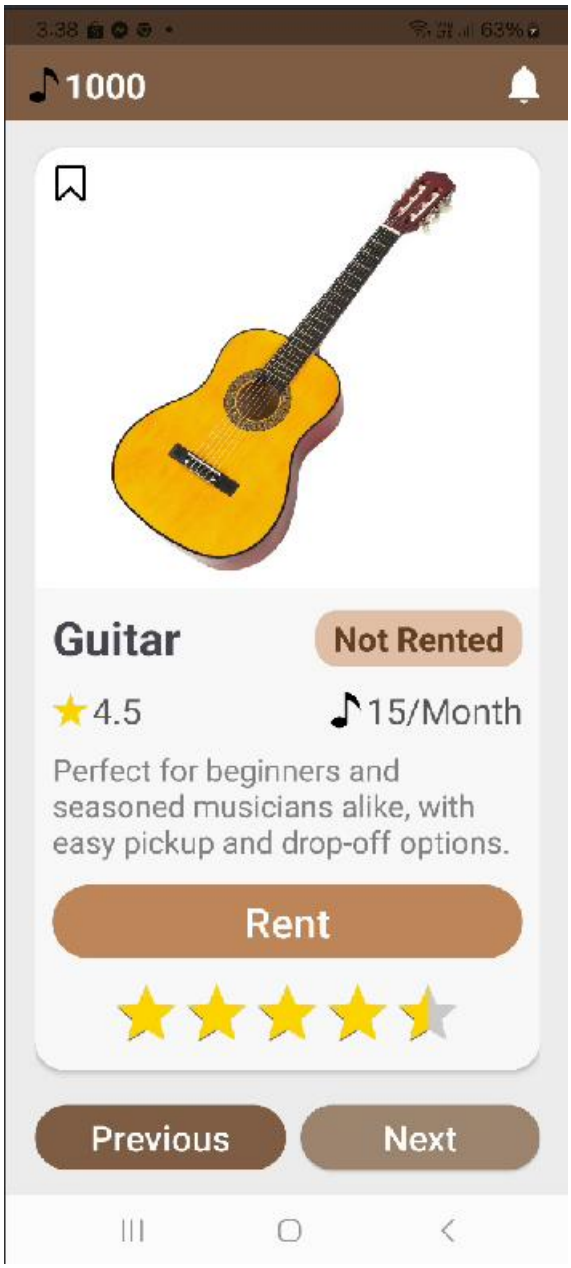
Use Case: Searching Instrument

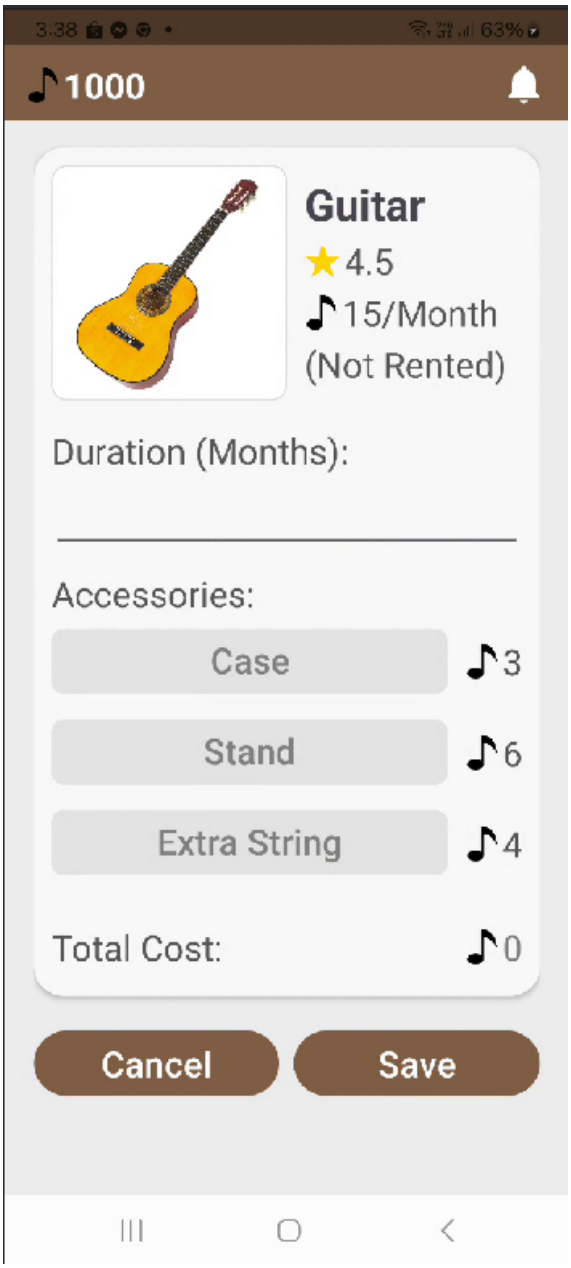
- Actors: User
- Preconditions: The user start the application
- Main Flow:
  - The user press Next to progress through the instruments and Previous to go back to previous instruments
  - The user give rating to the instrument if they want.
  - The user presses the Rent button to start renting

Use Case: Renting an Instrument

- Actors: User
- Preconditions: The user has pressed the Rent button of an instrument from the Main Activity.
- Main Flow:
  - The user Fill in the amount of time they want to rent (in month).
  - The user selects any accessories they want accompany the instrument.
  - The user presses the Save button

Based on the sketches I came up earlier, I designed the XML layout of the 2 activity.





Gap 4: Using Chip and ChipGroup

((Insert steps required to address his knowledge gap))

I'm defining a container for several Chip components that can be arranged and controlled collectively in this ChipGroup element. By adding vertical spacing between the Chips, the attribute app:chipSpacingVertical="2dp" enhances the visual distinction and overall layout attractiveness. Setting app:singleSelection="false" lets users choose more than one Chip in the group, which is helpful in situations where there are various alternatives to choose from, like picking from multiple instrument accessories. By making it simple for users to choose their preferences and keeping the layout neat and orderly, this configuration improves user interaction.

```
<com.google.android.material.chip.ChipGroup
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_weight="1"
    app:chipSpacingVertical="2dp"
    app:singleSelection="false">
```

I define three Chip components within a ChipGroup in this snippet of code, each of which represents one accessory from the instrument.accessories map. A custom style from style.xml with style="@style/chip" is used to style each Chip consistently, guaranteeing a consistent look. Because each chip has a unique ID (@+id/chip, @+id/chip2, @+id/chip3), programmatic reference is made simple. The chips can be selected thanks to the android:checkable="true" feature, which makes it easier for users to interact by giving them a variety of options. Expressions such as @{instrument.accessories.keySet().toArray()[0]} retrieve the first key for the first Chip, the second key for the second Chip, and so on. The android:text property also leverages data binding to dynamically set the text of each Chip to the relevant accessory name. This method produces a user interface that is both engaging and aesthetically pleasing, allowing users to easily select their preferred accessories while maintaining an organized layout.

```
<com.google.android.material.chip.Chip
    android:id="@+id/chip"
    style="@style/chip"
    android:checkable="true"
    android:text="@{instrument.accessories.keySet().toArray()[0]}" />
```

For each chip in the chipGroup array, I'm configuring an OnCheckedChangeListener in this code snippet to manage user interactions when the chips are checked or unchecked. When a chip is checked (isChecked is true), I use setChipBackgroundColorResource to change its background color to R.color.secondary and setTextColor to white. I add the matching accessory cost from the instrument.accessories map to the perMonth variable at the same time. On the other hand, I remove the accessory cost from perMonth and change the text color to R.color.textlight and the background color to R.color.unchecked when the chip is unchecked. To add audio feedback to the interaction, I also use soundPool.play to play a sound effect. Lastly, I use updateUI() to reload the user interface and make sure that all of the data that is shown reflects the current selections and calculated costs.

```
chipGroup[i].setOnCheckedChangeListener { _, isChecked ->
    if (isChecked) {
        chipGroup[i].setChipBackgroundColorResource(R.color.secondary) // Set checked color
        chipGroup[i].setTextColor(ContextCompat.getColor( context: this, R.color.white)); // Set checked text color
        perMonth += instrument.accessories.values.toIntArray()[i]; // Add to cost per month
    } else {
        chipGroup[i].setChipBackgroundColorResource(R.color.unchecked) // Set unchecked color
        chipGroup[i].setTextColor(ContextCompat.getColor( context: this, R.color.textlight));// Set unckecked text color
        perMonth -= instrument.accessories.values.toIntArray()[i]; // Reduce from cost per month
    }
    updateUI();
}
```

Gap 5: Parcelable Object

((Insert steps required to address his knowledge gap))

In order for the Instrument class to be able to be passed between activities, It will inherit from the Parcelable class.

```
data class Instrument (
      var name : String,
      var image : Int,
      var cost : Int,
      var months : Int,
      var rating : Float,
      var description : String,
      var accessories: Map<String, Int>
) : Parcelable {
```

Each property of the Instrument is extracted by the constructor using a Parcel as a parameter. I read "property" as a string that included "parcel."The image, cost, months, and rating characteristics utilizing parcel are displayed after readString().toString().parcel and readInt().readFloat() when necessary. I use parcel for the description once more.toString() and readString().

I read a Bundle from the parcel using parcel in order to handle the accessories property, which is a Map<String, Int>.readBundle(). The keys and values from the bundle are then transformed into a map using bundle?.keySet()?.associateWith { key -> bundle.getInt(key) }. When passing objects between activities or fragments in the Android application, this method makes sure that all of the Instrument object's properties are accurately recreated from the parcel, enabling effective serialization and deserialization.

```
constructor(parcel: Parcel) : this(
    parcel.readString().toString(),  // Read each property of the data class
     parcel.readInt(),
     parcel.readInt(),
     parcel.readInt(),
     parcel.readFloat(),
    parcel.readString().toString(),
     parcel.readBundle().let { bundle ->  //Use Bundle to get the accessories Map
          bundle?.keySet()?.associateWith { key -> bundle.getInt(key) } ?: emptyMap()
     }
) {
}
```

In order to serialize the Instrument data class's properties for storing in a parcel, I'm putting the Parcelable interface into practice. The necessary techniques are used to write each property to the parcel. The name is written by String(name).writeInt(picture), package.writeInt(price), package.parcel and writeInt(months).While parcel handles float and integer values, writeFloat(rating) handles both.The description is written using writeString(description).

To store the key-value pairs for the accessories property, which is a Map<String, Int>, I make a new Bundle. Using a for loop, I add each key-value pair to the bundle with bundle as I iterate through each entry in the accessories map.putInt(value, key). Lastly, I use parcel to write the bundle

```
override fun writeToParcel(parcel: Parcel, flags: Int) {
    parcel.writeString(name)  // Write each property of the data class
    parcel.writeInt(image)
    parcel.writeInt(cost)
    parcel.writeInt(months)
    parcel.writeFloat(rating)
    parcel.writeString(description)
    val bundle = Bundle()
    for ((key, value) in accessories) {  // Use Bundle to write the accessories Map
        bundle.putInt(key, value)
    }
    parcel.writeBundle(bundle)
}
```

Now with the Instrument class inherit from Parcelable and with all necessary constructor and writeToParcel functions written, I can pass it like I with primitive data type.

```
intent.putExtra( name: "updatedinstrument", instrument);
    // Put the Parcelable Instrument Object to be received by Main Activity
```

In the MainActivity, I can retrieve the object put by RentActivity by using getParcelableExtra. I have to put in the class type parameter in order to receive the correct class type.

```
val updatedinstrument = it.data?.getParcelableExtra( name: "updatedinstrument", Instrument::class.java);
    // Get the parcelable Instrument Object put by Rent Activity
```

Gap 6: Rental App Logic

There are 3 buttons in Main Acitivity. If the Next button is clicked, The app will cycle to the next instrument, looping back to the first instrument if the entire list is cycled through. The button also update the UI, rebinding the Instrument class, update the rating bar and instrument image. The previous button do the same thing as the Next button, but instead of cycling forward, it cicles backward and loop back to the last element if cycled to the first element. The button will also play sound and update the UI. The Rent button when clicked, will create a new Intent to move to Rent Activity. In the intent ,it will put int the current instrument as a parcelable object and the current credit. It then go ahead and use getResult to go to Rent Activity and await a return.

```
override fun onClick(v: View?) {
    when(v?.id){
        R.id.btn_next ->{                   // Switch to the next instrumnt in the list
            if(instrumentIndex >= instruments.size - 1) instrumentIndex = 0;
            else instrumentIndex++;
            updateUI()
            soundPool.play(buttonSound, leftVolume: 1f, rightVolume: 1f, priority: 0, loop: 0, rate: 1f)
        }
        R.id.btn_previous ->{               // Switch to the previous instrument in the list
            if(instrumentIndex <= 0) instrumentIndex = instruments.size - 1;
            else instrumentIndex--;
            updateUI()
            soundPool.play(buttonSound, leftVolume: 1f, rightVolume: 1f, priority: 0, loop: 0, rate: 1f)
        }
        R.id.btn_rent ->{  // Go the Rent Activity and pass in the current credit and the selected instrument
            val intent = Intent( packageContext: this, RentActivity::class.java);
            intent.putExtra( name: "instrument", instruments[instrumentIndex]);
            intent.putExtra( name: "credit", credit);
            getResult.launch(intent);
            soundPool.play(buttonSound, leftVolume: 1f, rightVolume: 1f, priority: 0, loop: 0, rate: 1f) // Play sound
        }
    }
}
```

In the rent activity, the save button when clicked, will validate the information being filled by the user, in this case, the edit text for duration in month. If the alidation is satisfied, It put the update the rented month property of the instrument object and put it in the Intent to Main Activity. It also putin the calculated total cost to later be reduced from credit in Main Activity. If calidation failed, it will show a snack bar to notify the user of any error.

```
R.id.btn_save ->{
    if(validate()){
        instrument.months = duration.text.toString().toInt();

        intent.putExtra( name: "updatedinstrument", instrument);
            // Put the Parcelable Instrument Object to be received by Main Activity

        intent.putExtra( name: "message", value: "Successfully Booked"); // Message for snack bar in Main Activity
        intent.putExtra( name: "cost", value: perMonth * duration.text.toString().toInt()); // Put in cost to reduct

        Log.d( tag: "Validation", msg: "Valid")

        setResult(RESULT_OK, intent); // Set RESULT_OK and finish to return to MainActivity, triggering the getResu
        finish();
    }
    else{
        showSnackBar(error, R.drawable.xmark);
        soundPool.play(errorSound, leftVolume: 1f, rightVolume: 1f, priority: 0, loop: 0, rate: 1f)

        Log.d( tag: "Validation", msg: "Invalid: " + error)
    }
}
```

If the cancle button in RentActivity is clicked, it will put in the intent the canclled message and return to Main Activity.

```
R.id.btn_cancel ->{
    intent.putExtra( name: "message",  value: "Cancelled"); // Set message for snack bar i
    setResult(RESULT_OK, intent); // Return to Main Activity
    finish();
}
```

The getResult function will be trigger when return from Rent Activity. It get the updated instrument from Rent Activity and update the current instrument with the new one. It also get the message put by Rent Activity and show a snack bar to notify the user.

```
val getResult = registerForActivityResult(ActivityResultContracts.StartActivityForResult()){ // Function run when return fro
    if(it.resultCode == Activity.RESULT_OK){
        val updatedinstrument = it.data?.getParcelableExtra( name: "updatedinstrument", Instrument::class.java);
            // Get the parcelable Instrument Object put by Rent Activity

        if(updatedinstrument != null){   // Only update instrument if Ren Activity actually return an Instrument object
            instruments[instrumentIndex] = updatedinstrument;
            val cost = it.data?.getIntExtra( name: "cost",  defaultValue: 0) // Get the total cost returned from Rent Activity
            if(cost != null){
                credit -= cost;  // Pay the cost of renting the instrument
            }
            updateUI()

            Log.d( tag: "Instrument",  msg: "Setted Instrument")
        }
        else Log.d( tag: "Instrument",  msg: "Null Instrument")

        val message = it.data?.getStringExtra( name: "message")
        if(message != null){
            showSnackBar(message, R.drawable.checkmark)  // Show snack bar of the returned message from Rent Activity
        }
        soundPool.play(successSound,  leftVolume: 1f,  rightVolume: 1f,  priority: 0,  loop: 0,  rate: 1f)
    }
}
```

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 1: Incorrect Rental App Logic

When coding the main functionality of the instrument rental application, I ran into a few problems. One of which was when I click the Rent button to go to the RentActivity and save the booking to return to MainActivity, the state of the app is not saved

Solutions:

The reason for this is the return to MainActivity is handled by creating a new Intent to go to MainActivity, this would mean go to a completely new MainActivity. To resolve this, instead of go from MainActivity to RentActivity and back to MainActivity using Intent the normal way, I created a getResult function in MainActivity to handle the return from RentActivity. Instead of going back using another Intent, RentActivity will set result o OK and finish, returning to MainActivity right where it left off.

```
val getResult = registerForActivityResult(ActivityResultContracts.StartActivityForResult()){ // Function run when return fro
    if(it.resultCode == Activity.RESULT_OK){
        val updatedinstrument = it.data?.getParcelableExtra( name: "updatedinstrument", Instrument::class.java);
            // Get the parcelable Instrument Object put by Rent Activity

        if(updatedinstrument != null){   // Only update instrument if Ren Activity actually return an Instrument object
            instruments[instrumentIndex] = updatedinstrument;
            val cost = it.data?.getIntExtra( name: "cost",  defaultValue: 0) // Get the total cost returned from Rent Activity
            if(cost != null){
                credit -= cost;  // Pay the cost of renting the instrument
            }
            updateUI()

            Log.d( tag: "Instrument",  msg: "Setted Instrument")
        }
        else Log.d( tag: "Instrument",  msg: "Null Instrument")

        val message = it.data?.getStringExtra( name: "message")
        if(message != null){
            showSnackBar(message, R.drawable.checkmark)  // Show snack bar of the returned message from Rent Activity
        }
        soundPool.play(successSound,  leftVolume: 1f,  rightVolume: 1f,  priority: 0,  loop: 0,  rate: 1f)
    }
}
```

Issue 2: Binder Classes Errors

I ran into this issue several time when implementing databinding in the xml layouts. Some time, the Binder class will have errors even though I haven't modified it. This is troublesome cause its really hard to know what is actually the problem when no specific errors are shown.

Solutions:

Turn out, the reason for the binder classes fail to compile was syntax error in the binding code in the xml layout. This is tricky to find because the error in data binding syntax is not shown to the user. At the end, I have to carefully read through my data binding code to ensure there is no incorrect syntax in the xml files. Once that is done, the issue is resolved.

Issue 3: Cannot Pass Parcelable Object

When my Instrument data class only have string, integer and float properties, the process of making it a parcelable object is straight forward. But when I implement accessories into the Instrument class using a Map, I cannot just write it to parcel like string, int or float. When attempting to do this, the object fail to write to parcel, and I receive a null object when retrieving it.

Solutions:

As discussed in the knowledge gap session, this issue was because Map objects cannot be write to parcel and read from parcel normally like primitive values. This was resolved using bundle when writing the Map to parcel and when read it from parcel. I Loop through each entry in the accessories map using a for loop, adding each key-value pair to the bundle with bundle.putInt(key, value). Finally, I write the bundle using parcel. I also read a Bundle from the parcel using parcel in order to handle the accessories property

```
val bundle = Bundle()
for ((key, value) in accessories) {  // Use Bundle to write the accessories Map
    bundle.putInt(key, value)
}
parcel.writeBundle(bundle)
```

Issue 4: Espresso Test Case Across Multiple Activities

When running an Espresso test case, in a single activity, no problem occurs. However, when I run a test case that span across both MainActivity and RentActivity, the test case will always fail because Espresso does not handle intent and activity changes by default

Solutions:

To resolve this issue, I needed to use ActivityScenarioRule. By automatically managing the lifetime of activities and making sure they are launched and cleaned up appropriately throughout tests, ActivityScenarioRule streamlines Android UI testing. It manages activity transitions smoothly, recovers the state of the activity, and permits secure user interface interactions without causing the main thread to stall. Test code becomes cleaner and easier to comprehend as a result of the reduction of boilerplate code and possible errors. All things considered, it abstracts away a large portion of the complexity associated with manual activity management, freeing up testers to concentrate on creating efficient tests for a variety of tasks.

```
@get:Rule
var activityScenarioRule = ActivityScenarioRule(MainActivity::class.java)
```

Reference:

HOEHLE, H. & VENKATESH, V. 2015. Mobile application usability. *MIS quarterly,* 39**,** 435-472.

CRAIG, C. & GERBER, A. 2015. *Learn android studio: build android apps quickly and effectively*, Apress.

CUELLO, J. & VITTONE, J. 2013. *Designing mobile apps*, José Vittone.

MOSKALA, M. & WOJDA, I. 2017. *Android Development with Kotlin*, Packt Publishing Ltd.