Task P – Spike: Core 2 – Sharing Is Caring

GitHub repo: https://github.com/SoftDevMobDevJan2024/core2-104222196

Goals:

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

Build a multi-activity app that communicates data between activities with Intents and incorporates advanced UI widgets. Practice writing well-factored code. Writing UI tests with Espresso.

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

- Building a multi-activity app
  - Launch one activity from another
  - Pass data back and forth between activities
- Incorporating advanced UI widgets
  - Using RatingBar
  - Using Slider
  - Using SnackBar
- Writing well-factored code
  - Defining and reusing styles
- Writing UI tests with Espresso

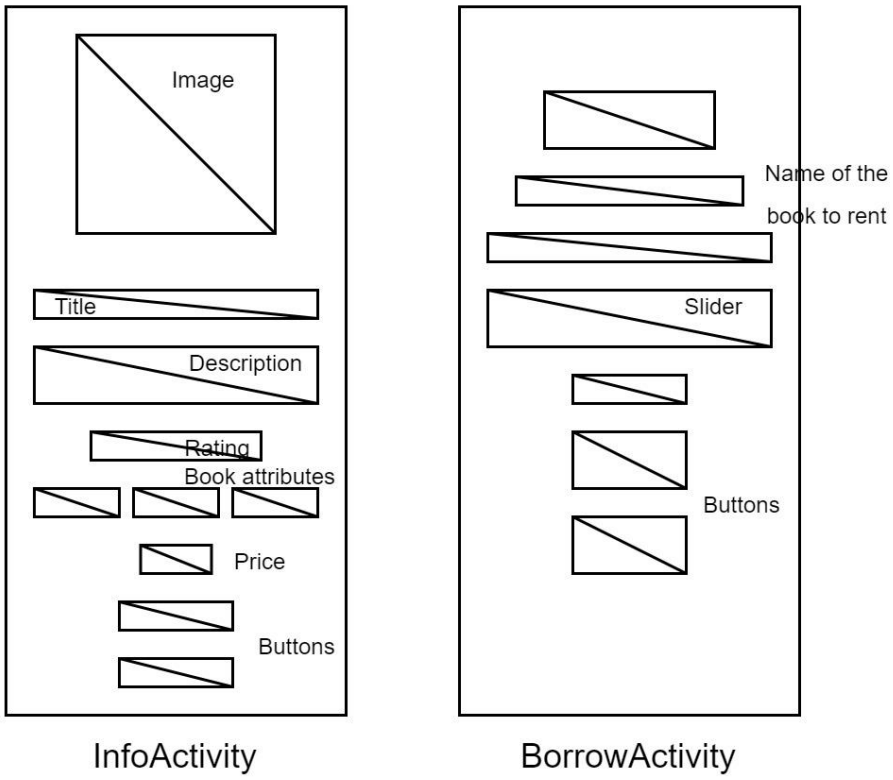
Tools and Resources Used

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

- Android Studio
- registerForActivityResult function: https://developer.android.com/reference/androidx/activity/result/ActivityResultCaller#public-methods_1
- StartActivityForResult contract: https://developer.android.com/reference/androidx/activity/result/contract/ActivityResultContracts.StartActivityForResult
- Intent: https://developer.android.com/reference/android/content/Intent
- RatingBar widget: https://developer.android.com/reference/android/widget/RatingBar
- Slider widget: https://developer.android.com/reference/com/google/android/material/slider/Slider
- SnackBar widget: https://developer.android.com/reference/com/google/android/material/snackbar/Snackbar
- Reusing styles: https://developer.android.com/develop/ui/views/theming/themes#Styles
- Espresso: https://developer.android.com/training/testing/espresso/basics
- Testing Sliders in Espresso: https://stackoverflow.com/questions/65390086/androidx-how-to-test-slider-in-ui-tests-espresso

Knowledge Gaps and Solutions

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



Sketches of the application. The application will feature two screens corresponding to two activities. InfoActivity is the entry point of the application and is launched when the application starts. BorrowActivity can be visited by clicking “Rent” on a book.

Gap 1: Building a multi-activity app

1. To add another activity to the app, that activity must be defined in the AndroidManifest.xml file. In the code below, InfoActivity is the main activity which is launched when the app is started. BorrowActivity is launched from InfoActivity.

```
<application ...>
    <activity
        android:name=".BorrowActivity"
        android:exported="false" />
    <activity
        android:name=".InfoActivity"
        android:exported="true">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

2. Define a class named BorrowActivity which inherits from AppCompatActivity. The code inside this activity is written in the same manner as the main activity. It has its own layout file and must override the onCreate function of the superclass.

```
class BorrowActivity : AppCompatActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_borrow)

        ... other pieces of code go here...
    }
}
```

3. In order to launch BorrowActivity from InfoActivity and have BorrowActivity pass the data back, we first need to create an activity launcher with the registerForActivityResult function. This block of code is written inside the InfoActivity class.

```
private val startForResult =
    registerForActivityResult(ActivityResultContracts.StartActivityForResult())
{ result ->
        if (result.resultCode == RESULT_OK) {
            // If the result code is OK, meaning the book has been rented
successfully, extract
            // the number of rented days from the ActivityResult object.
            val data: Intent? = result.data
            val daysRented = data?.extras?.getInt("daysRented")
            // Updates the current book object (which must be the one that has
been rented.)
            books[currentBookIndex].daysRented = daysRented
            // Updates the information on the screen.
            showBook(books[currentBookIndex])

            // Shows a Snackbar to let the user know that they has rented
successfully.
            val dueDate = LocalDate.now().plusDays(daysRented!!.toLong())
            val dateString = dueDate.format(DateTimeFormatter.ISO_LOCAL_DATE)
            Snackbar.make(findViewById(R.id.content),
                getString(R.string.rent_successful, dateString), 3000).show()

        } else if (result.resultCode == RESULT_CANCELED) {
            // If the result code is CANCELED, shows a Snackbar to let the user
know that they
            // has canceled the rent.
            Snackbar.make(findViewById(R.id.content),
getString(R.string.rent_cancelled), 3000).show()
        }
    }
```

The registerForActivityResult function returns an ActivityResultLauncher object that has a method to launch an activity. This function requires a contract object that specifies the type of the input to start the activity and the type of the result. In this case, the StartActivityForResult contract specifies that the input type is Intent and result type is ActivityResult. This function also receives a callback which is called when the result is available. This callback takes the result returned from another activity and operates on it, in this case to change the data inside InfoActivity.

4. When we wish to launch BorrowActivity from InfoActivity, we call the following block of code:

```
val intent = Intent(this, BorrowActivity::class.java)
intent.putExtra("book", books[currentBookIndex])
startForResult.launch(intent)
```

This block of code creates a new Intent object to start BorrowActivity and passes extra data into that object. In this case, the extra data is a Book data object. It then launches the activity with the launcher we created earlier.

5. The Book data class must implement the Parcelable interface. The @Parcelize annotation generates the implementation of the interface automatically for you.

```
@Parcelize
data class Book(
    @StringRes val nameResource: Int,
    @StringRes val descriptionResource: Int,
    @DrawableRes val imageResource: Int,
    val rating: Float,
    val pricePerDay: Double,
    val hasHardcover: Boolean,
    val isFiction: Boolean,
    val isInEnglish: Boolean,
    var daysRented: Int? = null
) : Parcelable { }
```

6. To read the data passed by InfoActivity, we use the following line inside BorrowActivity:

```
// Gets the Book object passed into the Intent and into this activity.
val selectedBook = intent.extras?.getParcelable<Book>("book")
```

7. When we are finished with BorrowActivity and wish pass back some data to InfoActivity, we run the following code:

```
val intent = Intent(this, InfoActivity::class.java)
intent.putExtra("daysRented", days)
setResult(RESULT_OK, intent)
finish()
```

This code creates an Intent object to pass back to InfoActivity, pass data into it, set the result to be returned, and end the activity. When the activity ends, the result will be propagated to the activity which launched it.

Gap 2: Incorporating advanced UI widgets

1. The InfoActivity makes use of the RatingBar widget to show the rating for each book. Inside the layout.xml file for the InfoActivity, the RatingBar can be specified with the following code (ConstraintLayout is used):

```
<RatingBar
    android:id="@+id/ratingView"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="4dp"
    android:scaleX=".5"
    android:scaleY=".5"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/bookDescriptionView" />
```

2. To programmatically control the number of stars that appear in the rating, set the “rating” property of the RatingBar view to a number:

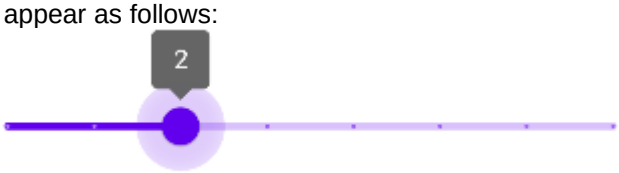
```
ratingBar = findViewById(R.id.ratingView)
ratingBar.rating = rating
```

The result can be seen below:

3. The BorrowActivity makes use of the Slider widget to let the user select an integer from 0 to 7. Inside the layout file for BorrowActivity, the Slider can be specified with the following code:

```
<com.google.android.material.slider.Slider
    android:id="@+id/daySelector"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_marginStart="16dp"
    android:layout_marginTop="32dp"
    android:layout_marginEnd="16dp"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintHorizontal_bias="1.0"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toBottomOf="@+id/howManyDays"
    android:valueFrom="0.0"
    android:valueTo="7.0"
    android:stepSize="1.0"
    android:contentDescription="@string/slider_content_description"/>
```

Important attributes for the Slider include the min and max value (valueFrom and valueTo respectively), and the step size. The result will appear as follows:



4. To programmatically read the value that the user has chosen when the Slider is interacted with, we can attach an event listener to it as follows:

```
var days = 0
val daySelector = findViewById<Slider>(R.id.daySelector)

// Adds an event listener to the Slider to update the number of rented days.
daySelector.addOnChangeListener { _, value, _ ->
    days = value.toInt()
}
```

The code above reads the value of the slider and assigns it to the variable “days” when the slider is interacted with.

5. Both activities also make use of the SnackBar widget. This widget appears as a message at the bottom of the screen to inform the user. This widget is not created in the layout but rather programmatically in the Kotlin code.

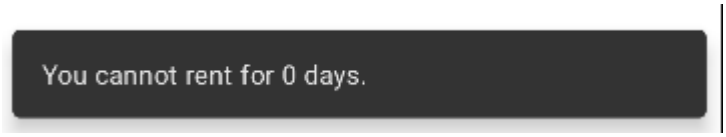
```
val snackbar = Snackbar.make(findViewById(R.id.content),
    getString(R.string.zero_days_disallowed), 3000)
```

The code above creates a SnackBar in the content view with a message telling the user that they cannot borrow a book for 0 days that appears for 3 seconds. At the moment, the SnackBar does not appear just yet.

6. To make the SnackBar show up, we must call the “show” method on it:

```
snackbar.show()
```

The result looks like this:



Gap 3: Reusing styles for better-factored code

1. Several UI components of the application share the same style, for example, all buttons in the app look like this:

2. To reuse styles across different components, we first need to specify them inside the styles.xml file in the values folder.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <style name="Button">
        <item name="android:layout_width">wrap_content</item>
        <item name="android:layout_height">wrap_content</item>
        <item name="android:background">@drawable/button_shape</item>
        <item name="android:textAllCaps">false</item>
    </style>

</resources>
```

A style is wrapped inside a <style> tag and comes with a name. Inside the <style> tag, we can specify one or more items representing the attributes associated with that style.

3. To apply this style to our UI elements, we specify the style attribute to those elements in the layout.xml file.

```
<Button
    android:id="@+id/confirmButton"
    style="@style/Button" />
```

Gap 4: Writing UI tests with Espresso

1. To test the InfoActivity class, create a test class in the androidTest module with the following boilerplate code:

```
@RunWith(AndroidJUnit4::class)
@LargeTest
class Core2EspressoTest {

    @get:Rule
    val activityRule = ActivityScenarioRule(InfoActivity::class.java)

}
```

This code sets up the class where we can write tests for the activity.

2. A test in Espresso is annotated with @Test. Inside this test, we can perform operations on views and check if their values match certain criteria.

```
@Test
fun clickNext1Time() {
    onView(withId(R.id.nextButton)).perform(click())


onView(withId(R.id.bookTitleView)).check(matches(withText(R.string.book_2_name))
)

onView(withId(R.id.bookDescriptionView)).check(matches(withText(R.string.book_2_
description)))
    onView(withId(R.id.costView)).check(matches(withText("$3.1")))
}
```

The code above selects the view with the ID “nextButton”, performs a click action on it, and checks if the other text views update their texts after this interaction.

3. Although the test class’ activity rule specifies only the InfoActivity class, we can test the BorrowActivity class too if one of our interactions launches it. The following code tests if the name of the book we have selected shows up correctly in the BorrowActivity after we click “rent”.

```
@Test
fun clickNext2TimesThenRent() {
    repeat(2) {
        onView(withId(R.id.nextButton)).perform(click())
    }

    onView(withId(R.id.rentButton)).perform(click())
    onView(withId(R.id.currentlyRenting)).check(matches(withText("You are now
renting Atomic Habits")))
}
```

4. Espresso does not support testing advanced UI widgets such as Sliders. However, we can write custom code for testing these elements. The following code specifies two methods that allow us to check and set the value of a Slider view.

```
private fun withValue(expectedValue: Float): BaseMatcher<View?> {
    return object : BoundedMatcher<View?, Slider>(Slider::class.java) {
        override fun describeTo(description: Description) {
            description.appendText("expected: $expectedValue")
        }

        override fun matchesSafely(slider: Slider?): Boolean {
            return slider?.value == expectedValue
        }
    }
}

fun setValue(value: Float): ViewAction {
    return object : ViewAction {
        override fun getDescription(): String {
            return "Set Slider value to $value"
        }

        override fun getConstraints(): Matcher<View> {
            return ViewMatchers.isAssignableFrom(Slider::class.java)
        }

        override fun perform(uiController: UiController?, view: View) {
            val seekBar = view as Slider
            seekBar.value = value
```

```
            }
        }
}
```

5. In the Test method, we can use these methods like so:

```
@Test
fun rentThenSelect3Days() {
    onView(withId(R.id.rentButton)).perform(click())
    onView(withId(R.id.daySelector)).perform(setValue(3.0f))

    onView(withId(R.id.daySelector)).check(matches(withValue(3.0f)))
    onView(withId(R.id.totalCost)).check(matches(withText("Total cost: $7.5")))
}
```

The code above launches the BorrowActivity, sets the slider to 3 and checks if the total cost reflects the number of days selected on the Slider.

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: UI tests are limited

This code for this assignment used Espresso to perform UI tests. However, as Espresso only supports testing basic views such as TextViews and Buttons out of the box, the tests implemented are rather limited. Custom code had to be written to test the Slider widget, and more would have to be written for widgets such as RatingBar or SnackBar. A different UI testing library with better support for advanced widgets should be looked into for a more thorough testing of the UI.