CSCI 448 – Lab 03A
Monday, January 30, 2023
LAB IS DUE BY **Tuesday, February 14, 2023 11:59 PM**!!

After completing your first app via the online tutorial, you've just been offered an internship at Little Green Games! The development team is very excited to get you up to speed so you can start completing some tickets on their latest app – Quizler. They know this one will be a hit, it just needs a little polishing. The team needs you to get the app installed on your device so you can add in a few new features.

Step 0 – Create the Quizler Project

As you start reading through the documentation, you can tell Little Green Games needs to improve their development process. The Quizler install guide doesn't tell you how to install the app – it tells you how to recreate the app! Egads. Well, you think, this will be a good chance to review all the components of MVVM and getting the state & events in place.

Create a project named Quizler. Be sure to target Android 10 and start with an Empty Compose Activity (Material3).

Check that all the dependencies are up to date. Open the `build.gradle (Project: Quizler)` file and set the compose version to be 1.3.3 (which is the latest as of this writing). Sync Gradle.

Then open the `build.gradle (Module :app)` file and update any dependency it is warning you about. Likely you'll need to set:

- `core-ktx 1.9.0`
- `lifecycle-runtime-ktx 2.5.1`
- `activity-compose 1.6.1`
- `material3 1.1.0-alpha04`

then sync Gradle again.

Step 1 – Model-V-VM

The first step the LGGDG (Little Green Games Development Guide) outlines is the data that needs to be store and represent our quiz questions.

Part 1.I – Make the Strings

There are several places we'll need to place text in our UI. These will be stored centrally in the `strings.xml` file.

String Name	**String Value**
`label_true`	`True`
`label_false`	`False`
`label_next`	`Next`
`label_previous`	`Previous`
`question1`	`Colorado School of Mines is located in South Bend, IN.`
`question2`	`The Battle of Hastings occurred in 1066.`

Additionally, create three more true/false style questions of your choosing so there are five questions all together.

Part 1.II – Create the Model Data

Next is to create the Data structure. Create a new package named `data`. In this package, create a new Kotlin `data class` called `Question`.

We'll specify two immutable data members in the constructor – the question ID and the Boolean answer. Specify the members as such:

```
@StringRes questionTextId: Int
isTrue: Boolean
```

The `@StringRes` annotation will restrict argument values to only the set that comprise string resource IDs. This is a good safety check to ensure that we are assigning valid values to our questions.

Part 1.III – Create the Question Repository

Make a Kotlin `object` (to follow the singleton model) for the `QuestionRepo`. The repository will have a a public list of questions comprised of the five question strings and their respective answers. The first two are provided below:

```
listOf(
  Question(R.string.question1, false),
  Question(R.string.question2, true)
)
```

Step 2 – M – VIEW - VM

The next chapter in the LGGDG walks through creating the accompanying View for the just created Model.

Create another new package called `presentation.question`. There's a sticky note at this point of the guide with a message "`TODO: create ticket for cheat screen`." Perhaps a hint of what's to come? It seems the development team is doing one thing right and grouping related screen components in a common package.

Part 2.I – Create the Screen Components

Part 2.I – `QuestionButton`

The first component is a stateless button called `QuestionButton`. This function takes two parameters: (1) the `buttonText String` (2) the `onButtonClick` event callback. We can now setup the internal button:

```
Button( onClick = onButtonClick ) {
  Text( text = buttonText )
}
```

Add a couple previews to ensure text appears on your button.

Part 2.II – `QuestionDisplay`

The next component is the stateless `QuestionDisplay` composable. This function takes a single `Question` object as a parameter and will display the question text plus a true and a false button.

The composable tree looks like

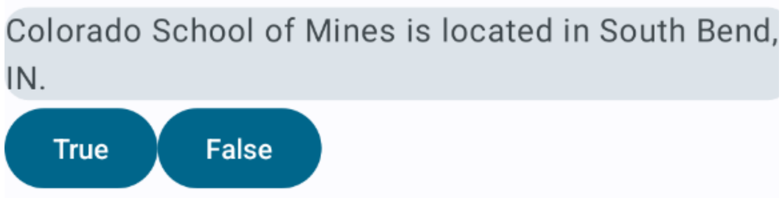
- `Column`
  - `Card`
    - `Text`
  - `Row`
    - `QuestionButton – True`
    - `QuestionButton – False`

For the Text, use the `Question.questionTextId` string. The buttons will use the corresponding true and false labels. But what should the event listener implementations be?

We'll throw in our first bit of UI Logic. Add two more parameters to `QuestionDisplay`, both will correspond to function objects with no parameters and no return value. Name the first `onCorrectAnswer` and the second `onWrongAnswer`.

For the true button, the `onButtonClick` event will call the appropriate function callback based on the actual question answer. For instance:

```
onButtonClick = {
  if( question.isTrue ) {
    onCorrectAnswer()
  } else {
    onWrongAnswer()
  }
}
```


Colorado School of Mines is located in South Bend, IN.

The false button will look similar.

Create the preview to ensure all information is displayed.

Part 2.III – `QuestionScreen`

Finally, it's time to put it all together. Create another Kotlin file called `QuestionScreen`. This composable function will create the following composable tree:

- `Column`
  - `QuestionDisplay`
  - `Row`
    - `QuestionButton – Previous`
    - `QuestionButton – Next`

All the event listeners can be set to an empty lambda with a `TODO` comment that will be implemented later. But first, the question needs to be set.

Step 3 – M – V – View Model

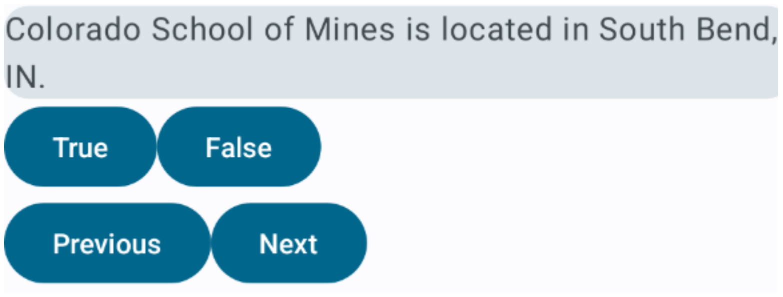
You're getting a better understanding of how the LGGDG is laid out. As expected, the next chapter walks through the setup of the View Model component. Staying in the Presentation Layer, you know this component will necessitate changes to the View as well.

Part 3.I – Create the View Model Question

Begin by creating the `QuestionViewModel` class. The constructor will store a private data member that is a `List` of `Question`s. It will also have a private mutable data member called `mCurrentQuestionIndex` (to denote it is private, prefix the identifier with an `m`) initialized to zero. Additionally, expose a public immutable data member called `currentQuestion` (of type `Question`) whose getter retrieves the current index from the question list.

```
val currentQuestion: Question
    get() = mQuestions[mCurrentQuestionIndex]
```

Returning to the `QuestionScreen`, this composable now depends on the `QuestionViewModel` and should require it as a parameter. The `currentQuestion` member is the value to send to `QuestionDisplay`.



A preview composable will now display the initial question text.

The event listeners now need values.

Part 3.II – Flow Events Up

The `QuestionViewModel` will initially handle two state modifications. These will both be encapsulated within function calls.

The first function will be called `moveToNextQuestion()` and will increment the current question index by one. Be sure to allow the index to wrap back down to zero (i.e. the first question) if the index advances beyond the beyond the size of the list (i.e. past the last question).

The other function will be called `moveToPreviousQuestion()`. This function will decrement the current index by one and similarly wrap the index to the last question. (*Kotlin note & hint: The modulus operator in Kotlin does not work in the expression A % B if A is negative.*)

Back on `QuestionScreen`, our Previous and Next buttons will call through to these two View Model functions.

At this point build, deploy, and run the app. Begin cycling through questions. Wait, the question text doesn't change. The View Model didn't set the current question as a state value!

Part 3.III – Flow State Down

In QuestionViewModel, we need to add another private immutable data member for the current question state. It will be initialized to the current question in our list:

```
private val mCurrentQuestionState =
                mutableStateOf( mQuestions[mCurrentQuestionIndex] )
```

The existing `currentQuestion` member needs to be updated to reflect it corresponds to a state value. The new type will be a State of type Question. We are now exposing an immutable version of the state value.

```
~~val currentQuestion: Question~~
~~    get() = mQuestions[mCurrentQuestionIndex]~~
val currentQuestionState: State<Question>
    get() = mCurrentQuestionState
```

We need to be sure to update the state appropriately. Whenever the question changes (next or previous), be sure to set the current question state's value to the new current question.

The final step is to have the `QuestionScreen` pass the value of the state to `QuestionDisplay`.

Build, deploy, run, and watch the questions fly by.

Satisfied with the reimplementation of Quizler, you realize you've reached the end of the Little Green Games Developer Guide. On the final page of the LGGDG, there are two reflection queries:

(1) **Why is the <u>public immutable getter</u> good practice for the View Model?** Why don't we just expose the entire question list publicly as well as make the current question index public? Similarly, why don't we just publicly expose the mutable state for the current question?

(2) Beginning to think ahead, we are tracking two values (the current question index and the actual current question) with only the latter stored as state. We need to ensure whenever the index changes that the question also changes. **How can this information storage be improved so we only need to track one value? What benefits are there to storing both values separately? Is there a way to keep the same public interface but a less error prone back end?**

Respond to both these questions in Canvas under the **`Lab03A Little Green Games Dev Query`** survey. The access code is **`flyguy`**.

Having successfully completed your onboarding, the dev team gives you your first ticket – informing the user of correct answers and tracking the user's score. It seems LGGDG has a way to go before this app can go live, it may be functional but it's incomplete.

Step 4 – Feature Request: Keep Score

The ticket reads as follows:

Feature Request: Three related UI/UX items need to be added:
(1) Display the user's current score on screen.
(2) If the user answers the question correctly, display to the user a message stating "Your wisdom is unmatched by any in the kingdom!" to denote a correct answer AND increment their score by one.
(3) If the user answers the question incorrectly, display to the user a message stating "You must consult the elders for additional knowledge." to denote an incorrect answer.

This type of change will necessitate modifications across our MVVM framework.

Part 4.I – Expand the Model

Since information will need to be displayed on screen, more strings need to be added to the `strings.xml` file.

String Name	**String Value**
`label_score_formatter`	`Score: %1$d`
`message_correct`	`Your wisdom is unmatched by any in the kingdom!`
`message_wrong`	`You must consult the elders for additional knowledge.`

The value for the `label_score_formatter` string is a string template. When resolving the string resource, we are able to provide an argument to substitute in for the placeholder. The `%1` denotes which argument to substitute (in the event there are multiple) and `$d` denotes the type of the argument (an integer in our case).

Part 4.II – Expand the View Model

`QuestionViewModel` will now track the score as state information. The score will be setup in the same manner that the current question state is setup. There will be:

- A private immutable data member that tracks the mutable state for the integer score. It is initialized to zero. (Call this data member `mCurrentScoreState`)
- A public immutable data member that gets the immutable state for the integer score. (Call this data member `currentScoreState`)

There also needs to be a public function that modifies the score state. Name this new function `answeredCorrect()` and have it increment the score state by one.

Part 4.III – Expand the View

A new composable needs to be created to display the score. Create a new Kotlin file in the `presentation.question` package named `QuestionScoreText`. The composable function depends on an integer score parameter.

A `Text` composable will be emitted and the `text` argument will be set to the format string that was specified with the function parameter passed as the string argument.

```
stringResource(id = R.string.label_score_formatter, score)
```

This new composable needs to be inserted in the `QuestionScreen Column` above the existing `QuestionDisplay` composable passing the current score from the View Model.

The app can be run and the score seen, but the True/False events were never wired up.

Part 4.IV – Connect the View and View Model

The two buttons will display a `Toast` to the user to inform them if they are correct or not. This will require a reference to the current context in order to resolve the associated string. Within the `QuestionScreen` composable, before setting up the `Column`, store a reference to the current context.

```
@Composable
fun QuestionScreen(questionViewModel: QuestionViewModel) {
  val currentContext = LocalContext.current

  Column {
    ...
```

The `onCorrectAnswer` callback will do two things:
(1) Call the View Model `answeredCorrect` function
(2) Create a `Toast` with the correct message

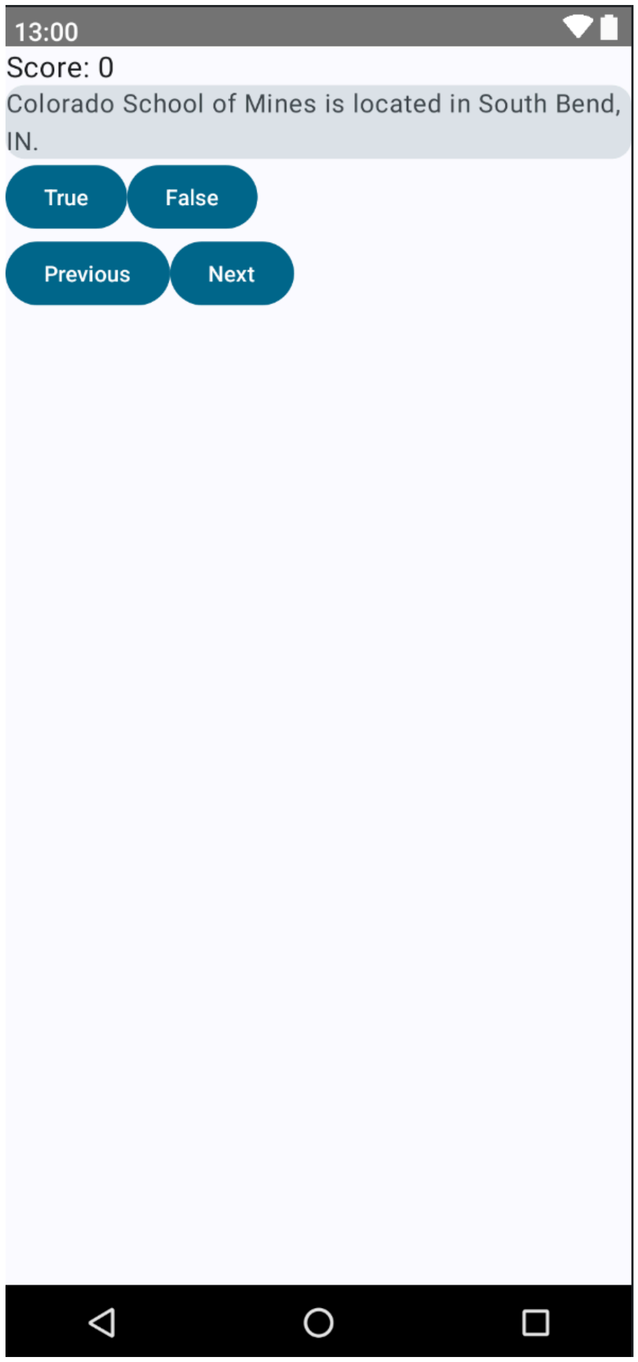
```
Toast
  .makeText(currentContext,
            R.string.message_correct,
            Toast.LENGTH_SHORT)
  .show()
```

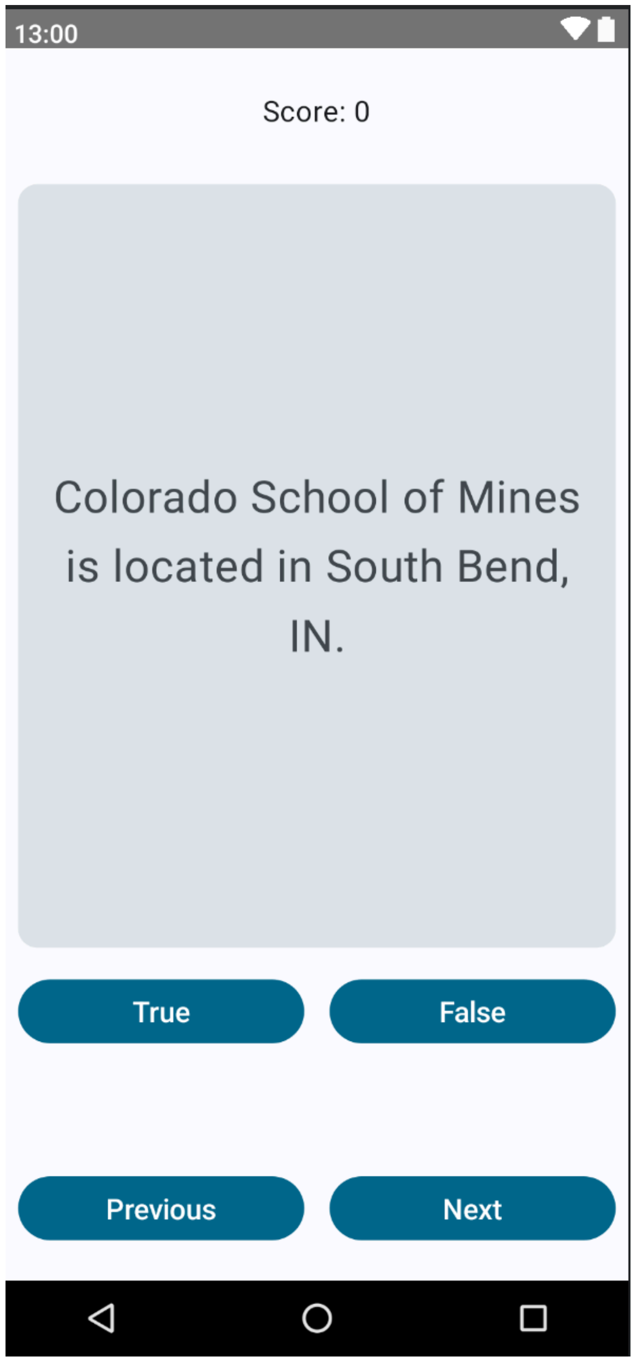
The `onWrongAnswer` callback will only display the wrong message `Toast`.

Everything is now all wired up. Build, deploy, run, and go for the high score! Excited, you're already imagining about future tickets. There are plenty of opportunities to add new features. There also seem to be some loopholes that need closing...

Step XC – Pretty It Up

Looking at the app, it's wholly functional – it does what it's supposed to. However, the look is not very clean or user friendly. Add the necessary modifiers (and potentially nested composables) to better align information. A before and after example is shown.





Step 5 – Deploy Your App

With Quizler deployed and verified working, you keep the LGGDG for reference and await the next ticket to come through as Lab03B.

LAB IS DUE BY **Tuesday, February 14, 2023 11:59 PM**!!