CSCI 448 – Lab 04C
Monday, February 13, 2023
LAB IS DUE BY **Thursday, February 23, 2023, 11:59 PM**!!

The CTO hasn't been around recently, the QA Team hasn't sent any new tickets, and the rest of the Dev Team seems to be playing some game with fantasy characters, are they even working? You take this opportunity to make your own branch of the code and implement your side project.

Step 0 – Create Your Own Ticket

You don't enter the ticket into tracking system, but you've grown accustomed to their format so set up your current issue in the same manner:

> *When a user is playing the game, upon rotating the device to landscape the buttons to answer and change questions become squished, move potentially off screen, and the labels are hard to read.*

Step 1 – Customize `QuestionDisplay`

The `QuestionDisplay` composable is the main offender and the easiest to reconfigure. You pull out your sketches and set to work.

Part 1.I – Switch On Orientation

Add one more parameter to the `QuestionDisplay` composable function. This one will be called `orientation`. Android tracks the orientation state as an integer and there are a set of constants that store the predetermined values. Default the value to `Configuration.ORIENTATION_PORTRAIT`.

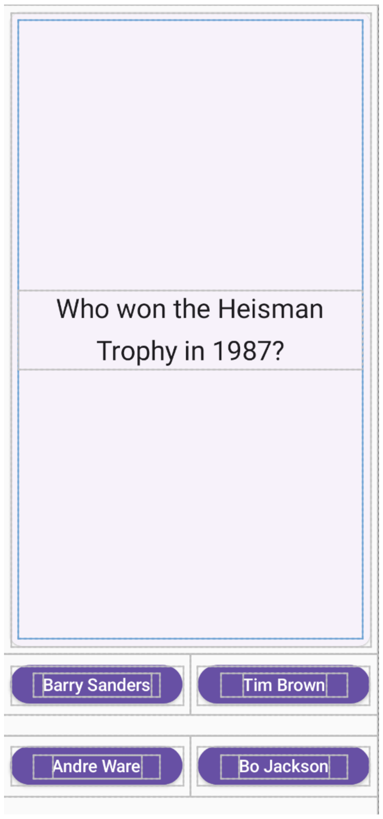
The very first thing to be done inside the `QuestionDisplay` function is to check what the current orientation is. Instead of using an in `if`/`else` for a binary portrait/landscape we will use a `when` statement to switch on the value. We want to specifically target the landscape orientation and any other possible orientation (portrait, square, undefined) will fall back to our previous portrait layout. The when statement allows us to more easily target additional values.

```
when (orientation) {
  Configuration.ORIENTATION_LANDSCAPE -> {
     // create landscape composable tree here
  }
  else -> {
    // create portrait composable tree here
  }
}
```

Part 1.II – Put the Portrait Mode Back in Place

Our entire previously existing composable tree needs to be moved into the `else` block of our when statement.

This composable tree is laid out as follows (see layout bounds to right):



- `Column`
  - `ElevatedCard`
  - `Row`
    - `QuestionButton`
    - `QuestionButton`
  - `Row`
    - `QuestionButton`
    - `QuestionButton`

Encapsulation for the coming part will be key to reduce duplication. Consider moving the specification for the `ElevatedCard` structure into its own composable called `QuestionTextCard`. Likewise for all the customization of the `QuestionButton` UI Logic, encapsulate it in a new composable called `QuestionChoiceButton`. (*This is not required but encouraged. Read on to see why we'll want to avoid duplicating as much logic as possible.*)

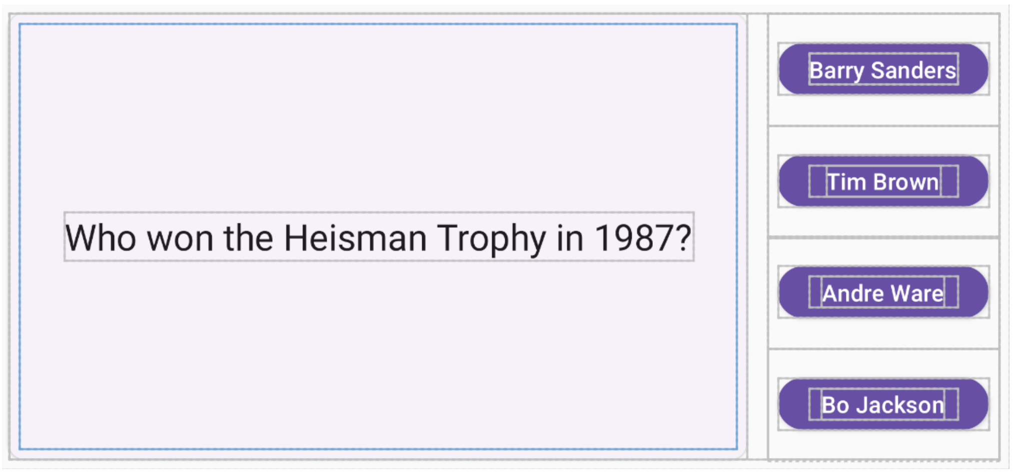
Part 1.III – Handle Landscape Mode

The landscape composable tree will have exactly all the same components, just laid out in a different order. (*Ahh, encapsulation. We'll need to call the same composables in two different spots…potentially with different arguments.*)

The landscape tree is laid out as follows (see layout bounds below):

- `Row`
  - `ElevatedCard`
  - `Column`
    - `QuestionButton`
    - `QuestionButton`
    - `QuestionButton`
    - `QuestionButton`

Nothing missing or added, just changed from the stacked column structure to a side-by-side row structure. We'll maximize our use of screen real estate.



Part 1.IV – Notes from Dr. Paone

A piece of paper falls from your sketchbook. Where'd this come from?

Comment and encouragement towards good design & practice:

The reason we keep splitting each element into its own composable function AND file is to encapsulate the specific constant settings and expose the variable settings as parameters. We want to maximize the reuse of this component. Composable trees tend to be designed top-down but then implemented bottom-up.

We can further improve the two composable trees by removing the duplication of repeated function calls. (Looking at you `Row` and `QuestionButton`.) Some for loops can really clean up our code.

This would give the overall structure for the `QuestionDisplay` function as:

```
when landscape
    Row
        ElevatedCard
        Column
            foreach QuestionButton
else
    Column
        ElevatedCard
        foreach Row
            foreach QuestionButton
```

In each orientation layout, particularly for `QuestionButton`, its called instance only exists once. This ensures that the same settings (modifiers, UI Logic, etc) are applied the same way to every instance.

Don't be fooled by https://xkcd.com/2730/. Proper design is worth the extra time and will save you future headaches. And, IMHO, there's something to be said about clean well-written code. Kotlin really excels in this area – compared to C/C++ and Java.

Part 1.V – Verify the Layout

To ensure the actual layout matches your schema, add a preview composable dedicated to the landscape orientation. We'll need to do two things for this new preview:

1) Create the preview device in landscape orientation
2) Tell the `QuestionDisplay()` to display in landscape mode

For the first, when creating the preview annotation, add a device spec argument:

```
@Preview(showBackground = true, device = "spec:parent=pixel_5,orientation=landscape")
```

You can type it all on or click on the gear next to the preview annotation and change the Orientation drop down.

Then for the second, set the orientation argument to be `Configuration.ORIENTATION_LANDSCAPE`.

Step 2 – Detect the Orientation in `QuestionScreen`

`QuestionScreen` calls `QuestionDisplay` and oversees our full screen layout. We will detect the orientation at this level and pass it along as appropriate. At the start of the `QuestionScreen` composable function, get the orientation from the current configuration:

```
val orientation = LocalConfiguration.current.orientation
```

Pass this value down to QuestionDisplay and voila!

Setup a similar preview for landscape mode. We only need to specify the preview device. The `QuestionScreen` composable will internally check the device orientation and send it on down to `QuestionDisplay`.

Note: check your previous/next buttons. You may need to customize your `QuestionScreen` composable to make sure these two buttons remain visible. YMMV.

Step 3 – Deploy Your App & Submit

You create your pull request and have attached a README outlining the following steps to verify correctness.

> When Lab04 is fully complete, you will submit a video of your working app to Canvas. Demonstrate the following actions inside the app:
>
> - With the device in portrait mode
> - Answer the first question correctly
> - Attempt to answer correctly again
> - Move to the second question
> - Answer the second question incorrectly
> - Move to the first question
> - Attempt to answer
> - Move to your multiple-choice question
> - Answer correctly
> - Rotate the device to landscape mode
> - Move to the next question
> - Move to the previous question
> - Rotate the device to portrait mode

Then stop the recording. Save it as webm format, name the video `<username>_L04.webm`, and upload this file to Canvas Lab04.

Not long after submitting, you see an email come to the top of your inbox with the subject "Pull Request Merged to QA Branch." Excited that the QA Team is already considering your self-driving work, you head over to try and see what game the rest of the developers appear to be playing.

LAB IS DUE BY **Thursday, February 23, 2023, 11:59 PM**!!