CSCI 448 – Lab 02A
Friday, January 27, 2023
LAB IS DUE BY **Friday, February 03, 2023 11:59 PM**!!

The final addition to MonsterLab will include two components:

1) A UI/UX addition to display a larger version of the monster's image
2) A structural modification to store the UI state

It won't be until the second component is in place through a View Model that the first component will be fully in place.

Step 1 – UI/UX

Part 1.I – Setup the `MonsterDisplay`

This will be the last composable that we create. In the `presentation` package, create a new Kotlin file named `MonsterDisplay`.

This composable function will accept a nullable `Monster` object as a parameter.

If the provided `Monster` object is not null, then we want to display the associated image (just like we did in the `MonsterCard`). Add an additional `modifier` argument to the `Image` composable specifying to fill the max size.

Otherwise if the `Monster` object is null, then display a `Text` message stating "`Please select a monster`." Be sure to utilize the strings resources appropriately.

Hooray, our first piece of UI Logic!

To properly test this composable, create two previews. The first preview pass a Mike Wazowski `Monster` object as the argument (copy it from the repository). The second preview pass a null argument. You should see the Mike Wazowski image and the select a monster text as appropriate.

Part 1.II – Add the `MonsterDisplay`

On our `MonsterLabScreen`, we'll now display both the list AND the corresponding monster. Mimic the following composable tree:

- **`MonsterLabScreen`**
  - **`Column`**
    - **`Box – modifier : weight = 0.5f, fill max size`**
      - **`MonsterList – monsters : MonsterRepo.monsters`**
    - **`Box – modifier : weight = 0.5f, fill max size`**
      - **`MonsterDisplay – monster : null`**

This will stack the list on the top half of the screen and the display on the bottom half of the screen. Since composables are measured at the time they are created, we must nest our composables inside of a `Box` which can then be sized appropriately. The `weight` modifier tells the composable what percentage of the available space to use. Since each box has a weight of `0.5`, they each will fill 50% of the column space.

In either the preview, or by deploying the app, you can now scroll through the list – which occupies only half the screen now.

Step 2 – Store the UI State in a View Model

Part 2.I – Store the UI State and Flow the State Down

The goal is to have the user click on a monster in the list and then see a larger image of the corresponding monster. The corresponding monster that was selected represents the current state of the UI. (Actually, where we are in the scrollable list is also a piece of the UI state. That state is kept internally. Review – what type of composable is `LazyColumn`?) We will track the UI state by hoisting it out of the screen into a View Model. The View Model will be the single source of truth for our UI.

Part 2.I.A – Create the View Model

The View Model component of the MVVM architecture lives in the Presentation Layer. Create a new Kotlin class in the `presentation` package named `MonsterViewModel`. Add a public data member that stores the state of the selected monster and give it an initial value of null (since when the user first comes into the app they have not selected a monster).

```
val selectedMonsterState: MutableState<Monster?> = mutableStateOf(null)
```

This is where the UI state lives. Since the state will change, we need to use `MutableState` so our state object has a setter associated with it.

The View Model also observes and holds the data from the Model Layer. Add to the constructor for our MonsterViewModel class a parameter data member corresponding to the list of monsters.

```
val monsters: List<Monster>
```

The View Model is now in place. When the View Model is constructed, it will receive data from the Model and initializes the View state. We now need to pass the state to the View.

Part 2.I.B – Use the View Model

`MonsterLabScreen` is the composable component that contains both the monster list and the monster display. The screen will need to get the monster from the list and send it to the display – here is where our state comes in.

Since the `MonsterLabScreen` needs state and the `MonsterViewModel` is our single source of truth that holds the state, add a `MonsterViewModel` parameter to `MonsterLabScreen`. Now the list will accept the set of monsters from the View Model and the display will accept the selected state's value.

In both the preview, and the `MainActivity`, we'll create a `MonsterViewModel` object and provide it as an argument. Thinking in the context of our app, the single source of truth becomes apparent. When the app starts, the app creates an instance of `MainActivity`. When `MainActivity` is created, it gets the data from the Model, constructs a single instance of our View Model, and provides the View Model to the View. The activity houses our full MVVM stack.

Run the app at this point and it should function as it had at the end of Step 1. All that remains is to change the state so the View updates.

Part 2.II – Update the UI State by Flowing Events Up

Part 2.II.A – Create the Event

Let's follow the flow of our state down.

- The View Model holds all the data. The View Model is given to `MonsterLabScreen`.
- We then send the list of monsters to `MonsterList`.
- The list creates a `MonsterCard` for each item in the list.
- A single monster is then displayed in `MonsterCard`.

The `MonsterCard` composable is now holding the reference to the single monster that can be selected. It is at this level that we will denote a component can be selected and will fire an event.

We are able to add an `onClick` argument to our `Card` composable. When the card is clicked by the user, our callback will fire. What should we do when clicked? We need to say that a monster was clicked and specify which monster it was. What happens to that monster? Well, whatever it's told to do.

The `MonsterCard` composable will need to accept a function object parameter that corresponds to the click callback. Since we need to specify which monster was clicked, the function signature will accept a Monster as a parameter and return nothing.

```
onMonsterClicked: (Monster) -> Unit
```

Then, the `Card onClick` will pass the received `Monster` object as the argument to the `onMonsterClicked` function.

```
onClick = { onMonsterClicked(monster) }
```

Patch the two preview functions by passing an empty lambda as we are only previewing the View and have no event to handle.

Part 2.II.B – Flow the Event Up

This event will correspond to a state change. The event specifies which monster was clicked – or what our new state will be. Time to flow the event up to the View Model.

`MonsterCard` is called from within `MonsterList`. `MonsterList` will need to expose the event listener that `MonsterCard` requires. Add another function object parameter to `MonsterList` that has the same signature as the `MonsterCard` parameter. Pass this function object as our argument to the `MonsterCard` parameter. Patch the preview as well.

`MonsterList` is now called from the `MonsterLabScreen`. Here is where our state originated and here is where we'll update the View Model. We'll finally provide an implementation for the click event listener. Use a lambda expression to set the selected monster state value on the View Model object.

At this point, our unidirectional flow is complete. The interactive preview mode can demonstrate the state change.

Step 3 – Deploy Your App, Submit, & Respond

Once you have the event updating the state, deploy your app to your device.

When Lab02 is fully complete, you will submit a video of your working app to Canvas. Demonstrate the following actions inside the app:

- Scroll to the bottom of the list
- Select Terri & Terry
- Scroll to the top of the list
- Select James P. Sullivan
- Scroll to the middle of the list
- Select Art

Then stop the recording. Save it as webm format, name the video `<username>_L02.webm`, and upload this file to Canvas Lab02.

Additionally, answer the L02 Exit Interview questions in Canvas. The access code is: **`scaregames`**.

LAB IS DUE BY **Friday, February 03, 2023 11:59 PM**!!