Task ##P/C – Spike: Climbing Application

Repository: https://github.com/hakl04/MobileAppDev/tree/main/Assignment1

Goals:

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

Insert summary here:

The assignment focuses on essential mobile app development skills and entails creating a basic scoring app for a nearby climbing club. In addition to making interactive buttons for climbing, falling, and resetting scores, we will need to learn how to create adaptable layouts for both portrait and landscape orientations. Using conditional logic to enforce rules like avoiding falls before reaching the first hold, the app will develop a dynamic scoring system based on the climber's position on the wall. In order to preserve scores throughout device rotations, we will also learn how to maintain the application state using saveInstanceState. Multi-language support will also be incorporated for wider accessibility. The use of logging will improve debugging abilities by offering insights into monitoring user behavior and resolving problems.

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

- Multiple Language support
- Changing between portrait and landscape
- State persistence when changing orientation
- User-Friendly Interface Design
- Climbing Game logic

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: Multiple language support

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

The first step was to define all the necessary strings in string.xml that will need to be used in the app. These include the app’s name, and strings for Climb, Fall, Reset, Hold, and Score.

```xml
<resources>
    <string name="app_name">Climbing App</string>
    <string name="climb">Climb</string>
    <string name="fall">Fall</string>
    <string name="reset">Reset</string>
    <string name="hold">Hold</string>
    <string name="score">Score</string>
</resources>
```

Instead of setting text for button and Text views with concrete values, we need to set these text values using strings from string.xml. The reason we do this is because this allows us to get strings dynamically from our resources that we can set up to support multiple languages.

```
btnClimb.setText(R.string.climb);
btnFall.setText(R.string.fall);
btnReset.setText(R.string.reset);
```

For multiple language support, we will create values folders in res for different languages (values-vi for Vietnamese for example). Depending on the current language in the device setting, the strings will be taken from the strings.xml of that language.

This will allow us to create a strings.xml for each language, in which we will define our strings for different languages. For example, this is the strings.xml for Vietnamese:

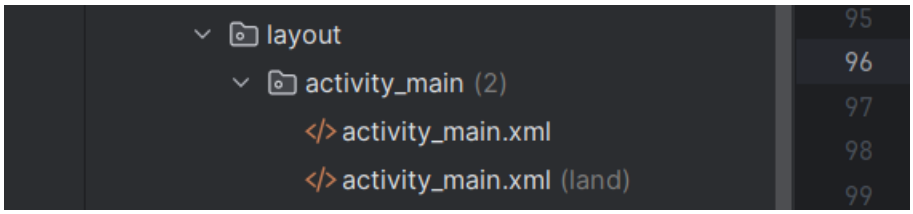
```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">Trò chơi leo trèo</string>
    <string name="climb">Trèo</string>
    <string name="fall">Ngã</string>
    <string name="reset">Chơi lại</string>
    <string name="hold">Điểm Bám</string>
    <string name="score">Điểm</string>
</resources>
```

Gap 2: Changing between portrait and landscape

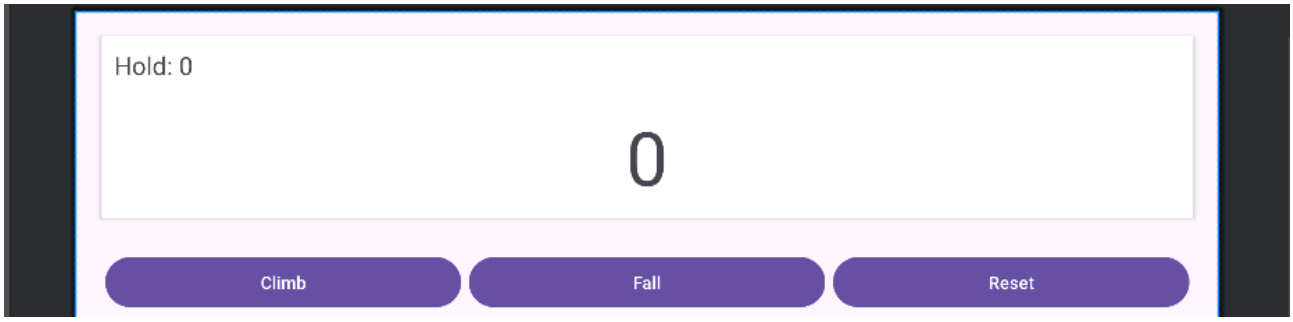
When the app orientation is changed from portrait to landscape and vice versa, the states of the app like current hold and current score will be reset.

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

First, we need to create a new folder specifically for landscape layouts. To do this, we chose layout as the resource type and named the new directory layout-land. This new folder will hold the layout files we want to use when the device is in landscape mode. Android Studio will automatically switch to the layouts in this directory when the app is in landscape mode.



After creating the layout-land folder, we will go ahead and add a layout file that corresponds to our existing portrait layout. I right-click on the layout-land folder and choose New > Layout Resource File, naming it the same as my portrait layout file, activity_main.xml. Once I create this file, I can design the layout for how I want it to appear in landscape orientation.



Gap 3: State persistence when changing orientation

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))

To achieve this, we can use saveInstanceState. saveInstanceState is an essential Android technique that aids in maintaining the state of an Activity or Fragment when it is momentarily destroyed and regenerated, as occurs during screen rotation or other configuration changes, or when the system needs to recover resources.

The first step is to override the onSaveInstanceState method in our Activity. This method is called before the Activity may be destroyed, allowing us to save any necessary state information. In this method, we will use the bundle object (outState) to store the data we want to preserve. We can utilize various methods such as putString, putInt, and others to save different types of data. For this specific app, we will use putInt to save the current score and current hold, and putBoolean to store the Boolean of whether the climber has fallen.

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    outState.putInt("hold", climbingGame.getCurrentHold())
    outState.putInt("score", climbingGame.getCurrentScore())
    outState.putBoolean("fell", climbingGame.getCurrentFell())
}
```

The next step is to handle the restoration of this data. We will override the onRestoreInstanceState method of our Activity. This method is called when the Activity is being recreated, including orientation change from portrait to landscape and vice versa. In this method, we can retrieve previously saved values using the corresponding get methods (like getString, getInt, etc.). We will use getInt to get back the current hold and current score saved in onSaveInstanceState and getBoolean to retrieve the Boolean indicating whether the climber has fallen. These restored variables will then be used to recreate the ClimbingGame object.

```
override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
    var hold : Int = savedInstanceState.getInt( key: "hold",  defaultValue: 0)
    var score : Int = savedInstanceState.getInt( key: "score",  defaultValue: 0)
    var fell : Boolean = savedInstanceState.getBoolean( key: "fell",  defaultValue: false)
    climbingGame = ClimbingGame(hold, score, fell);
}
```

Gap 4: User Friendly Interface Design

This knowledge gap contains many UI/UX-related design choices that I have to come up with solutions.

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

The interface consists of a card that holds the current hold and current score. Below the card are the 3 buttons for Climb, Fall, and Reset. For the portrait layout, the buttons are arranged vertically from top to bottom, and in the landscape layout, they are arranged horizontally from left to right to make better use of the horizontal space.

The score display will need to be recolored depending on the zone (blue from hold 1 to 3, green from hold 4 to 6, and red from hold 7 to 9). This choice will make it easier for users to know which zone they are on. This can be achieved using the setTextColor of TextView. We will use the when clause to set the text color accordingly.

```
fun changeColor(score : Int) {
    when (score) {
        in 1 ≤ .. ≤ 3 -> scoreText.setTextColor(Color.BLUE)
        in 4 ≤ .. ≤ 9 -> scoreText.setTextColor(Color.GREEN)
        in 10 ≤ .. ≤ 18 -> scoreText.setTextColor(Color.RED)
        else -> scoreText.setTextColor(Color.BLACK)
    }
}
```

For a more user-friendly experience, the button should be disabled when that functionality cannot be used (the fall button will be disabled if the climber has already fallen for example).

setEnabled(boolean enabled) method is a function for managing the interactivity of Buttons. This method allows developers to enable or disable a Button based on the application's logic. When a Button is disabled, it cannot be clicked or interacted with by the user, and its visual appearance changes to indicate this state, appearing grayed out. This functionality helps guide user actions and prevent unintended interactions. The climb button will now only be enabled when the climber has not fallen and has not reached the 9th hold. The fall button will only be enabled when the climber has not fallen, the climber has reached at least hold 1, and has not reached hold 9.

```
btnClimb.setEnabled(!climbingGame.getCurrentFell() && climbingGame.getCurrentHold() < 9);
btnFall.setEnabled(!climbingGame.getCurrentFell() && climbingGame.getCurrentHold() > 0 && climbingGame.getCurrentHold() < 9);
```

Gap 5: Climbing Game logic

In order to implement the climbing game logic, the ClimbingGame class was created. The hold variable, which keeps track of the player's current location on the ascending route; the score variable, which shows the total number of points earned; and fell, a boolean that indicates if the player has fallen, are the three

main properties maintained by the class. The getCurrentHold, getCurrentScore, and getCurrentFell methods provide easy access to the player's current hold position, score, and fall status, respectively. These functions will be used by the MainActivity to access the properties.

```
class ClimbingGame(hold : Int, score : Int, fell : Boolean){
    var hold : Int = 0;
    var score : Int = 0;
    var fell : Boolean = false;

    public fun getCurrentHold() : Int = hold;
    public fun getCurrentScore() : Int = score;
    public fun getCurrentFell() : Boolean = fell;
```

The climbing method lets the player go through the holds, increasing the score according to the range of the current grip using the when clause while making sure the player can only climb if they haven't fallen. Logs are used to debug whether the climb was successful.

```
public fun climb() : Int{
    if(!fell){
        score += when (hold) {
            in 0 ≤ .. ≤ 2 -> 1
            in 3 ≤ .. ≤ 5 -> 2
            in 6 ≤ .. ≤ 8 -> 3
            else -> 0
        }
        if(hold < 9) hold ++;
        Log.d( tag: "ClimbingGame",  msg: "Climb successful");
    }
    else{
        Log.d( tag: "ClimbingGame",  msg: "Cannot climb after fell");
    }
    return getCurrentScore();
}
```

The fall function, on the other hand, deals with the fall's effects by preventing the score from falling below zero and deducting 3 points if the player is on a legitimate hold and hasn't previously fallen. Similar to the climb method, logs are used for debugging purposes with 2 log messages indicating whether the fall method call was successful or failed.

```
public fun fall() : Int{
    if(!fell && hold >= 1 && hold < 9){
        score -= 3;
        if(score < 0){
            score = 0;
        }
        fell = true;
        Log.d( tag: "ClimbingGame",  msg: "Fall");
    }
    else{
        Log.d( tag: "ClimbingGame",  msg: "Already fell");
    }
    return getCurrentScore();
}
```

Lastly, the reset method marks fall as false and resets the score, and the hold variable back to zero, returning the game to its starting point. Players can now start over whenever they'd like. A log is also utilized for debugging purposes.

```
public fun reset() : Int{
    score = 0;
    hold = 0;
    fell = false;
    Log.d( tag: "ClimbingGame",  msg: "Reset");
    return getCurrentScore();
}
```

These methods will be called depending on the button clicked (button climb call the climb method, button fall call the fall method, button reset call the reset method). Through this, the climbing app functionality is completed.

```
override fun onClick(v: View?) {
    when(v?.id){
        R.id.btnClimb ->{
            climbingGame.climb();
        }
        R.id.btnFall ->{
            climbingGame.fall();
        }
        R.id.btnReset ->{
            climbingGame.reset();
        }
    }
    updateUI()
}
```

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 Climbing game logic

At first, The logic of my climbing app was designed poorly. The climb function will only add to the score without any consideration of the min and max possible score. The same thing happened with the fall function, which reduce 3 from the score without checking for a negative score and without taking into account when the climber can actually fall (at least hold 1 and before hold 9). This caused the application to behave incorrectly.

Solutions:

The Solution to this problem can be to create a clamping function that set the score to the max value if it exceed it, and to the min value if the score is less than it. Any function that changes the score like adding when climbing and reducing when the climber falls will have to call this clamping function to ensure the score is always within the specified range.

Issue 2: Lost Variables When Switching Between Portrait and Landscape

When switching the layout from changing between portrait and landscape, the variables of the activity will be lost. This is a big issue because accidentally switching from portrait to landscape and vice versa, when a climbing session is ongoing, will result in the current progress being lost.

Solutions:

As discussed in the knowledge gap session, this issue was resolved using saveInstanceState. We can use onSaveInstanceState to save variables we needed to save then use onRestoreInstanceState to retrieve these saved variables. This is an important method to learn and will help me solve similar issues in future projects.

Issue 3: Switching between Portrait/Landscape not working

When I finished making the layout for portrait and landscape by creating a layout-land folder and another activity_main.xml in that folder for the landscape layout, in theory, the app should switch layout automatically on screen rotation. However, when I finished setting that up, the layout did not switch when the screen rotated.

Solutions:

In the end, I fixed the problem by running the setContentView function again in the onConfigurationChanged function. This is the function that runs after a configuration change like a screen rotation. By setting the content view again, we can ensure the layout will be forced to update on screen rotation.

```
override fun onConfigurationChanged(newConfig: Configuration) {
    super.onConfigurationChanged(newConfig)
    setContentView(R.layout.activity_main);
    getUI();
}
```

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.