

|  |  |
| --- | --- |
| School of Engineering, Computing and Built Environment | |
|  | |
| Department of Computing | |
|  | |
| **DIPLOMA IN COMPUTER STUDIES** | |
|  | |
| [DMT22114 MOBILE TECHNOLOGY] | |
|  |
| [LECTURER: LAI KIM KIM] |

Assignment : [1]

|  |
| --- |
| [Basic Bill Calculation Application: Billculator] |
| |  |  |  | | --- | --- | --- | | STUDENT NO | NAME | Marks | | 0204677 | Lim Zhe Yuan |  | |
|  |

|  |  |  |
| --- | --- | --- |
| DUE DATE | : | [25/02/2022] |
|  | | |
| TOTAL MARKS | : | [100 MARKS / 20%] |

**Table of Contents**

Main Report 1

INTRODUCTION 1

FUNCTIONALITIES OF THE APPLICATION 2

STRENGTHS OF THE APPLICATION 8

FLAWS OF THE APPLICATION 10

Appendix 14

Marking Rubric 21

**Main Report**

**Introduction**

Billculator is a basic bill calculation app that serves to simplify bill calculations by providing a sleek and user-friendly interface that improves user efficiency and preferability. Though simple and lightweight, it provides numerous functionalities that will make calculating bills easier and quicker, shifting user’s focus to other things that really matter and boosting productivity. Balancing between design and functionality, Billculator looks to be the next big utility app that is worth investing time and energy for.

**Emulators/Virtual devices used**

The device used to capture the following diagrams will be indicated in their respective labels using the format ***(Device: deviceName****)* when necessary.

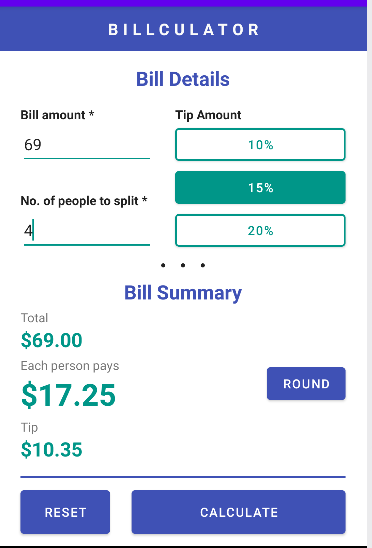
Devices:

* **Nexus** (1440x2560; 560dpi)
* **Pixel** (1080x1920; 420dpi)
* **Galaxy Nexus** (720x1280; xhdpi)

**Functionalities of the application**

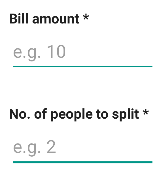
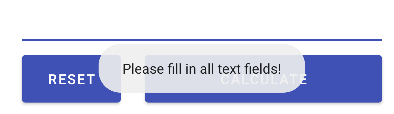
As a bill calculation app, Billculator provides the following functionalities. As a disclaimer, **Nexus** is the device used for all emulator diagrams shown in this section.

* **Bill splitting calculation**



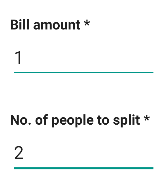
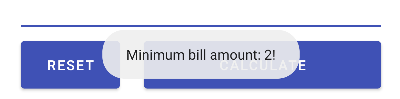
***Diagram 1.1.0:*** *Billculator example view*

Billculator allows users to perform basic bill splitting calculations. Inputs for both bill amount and number of people to split the bill with must be determined and entered by the user before calculating, otherwise a toast error message will appear and prompts the user to fill up any required inputs, which are indicated by a ‘**\***’ symbol in their labels.

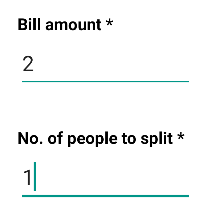
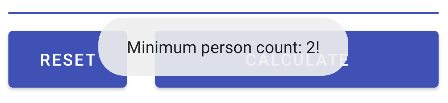


***Diagram 1.1.1:*** *Toast error message that appears when inputs are empty*

Billculator also restricts users from calculating bill amounts that are less than 2 to avoid granularized calculation results. A toast error message will appear and notify the user that the minimum bill amount is 2 (depends on the minimum value set in code) if the user inputs incorrectly. Besides that, Billculator also refrains users from entering values less than 2 in the “Number of people to split” field to avoid logical errors. Another toast error message will appear and notify the user that the minimum number of people that can be entered is 2 if the user inputs incorrectly.

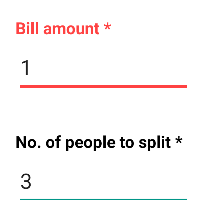
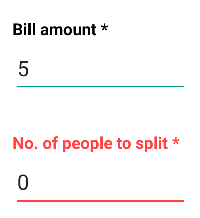
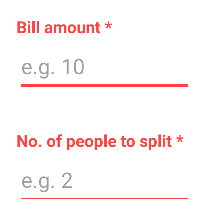


***Diagram 1.1.2:*** *Toast error message that appears when bill amount is less than 2*



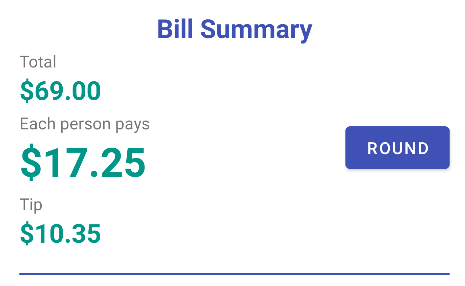
***Diagram 1.1.3:*** *Toast error message that appears when number of people to split is less than 2*

Upon pressing the “Calculate” button, input field values undergoes validation before performing bill calculations. If there are invalid inputs in the input fields provided, the problematic input fields will be highlighted in red and focused to aid users in error detection and recovery. This feature ultimately improves usability and reduces the user’s time to resolve unexpected calculation issues.



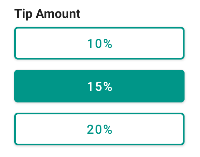
***Diagram 1.1.4:*** *Inputs are highlighted in red if they are invalid*

If all inputs are valid, a bill summary about the bill total, the amount of cash each person needs to pay for, and the amount of tip will be displayed accordingly.



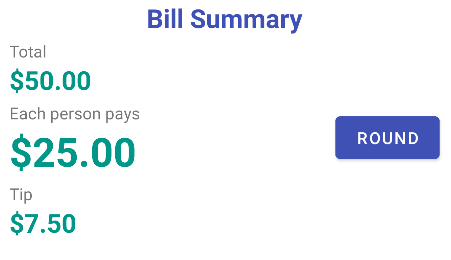
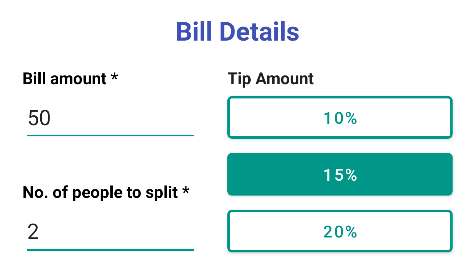
***Diagram 1.1.5:*** *Bill summary view*

* **Tip calculation**



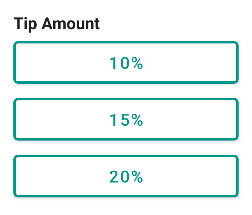
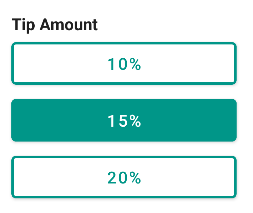
***Diagram 1.2.0:*** *Tip buttons*

Aside from bill splitting calculations, Billculator also allows users to calculate tips based on their bill amount. There are 3 tip percentages to choose from: 10%, 15% and 20%. Once selected, the tip amount is calculated based on the chosen tip percentage and the bill amount that was inputted by the user when the user presses the “Calculate” button.



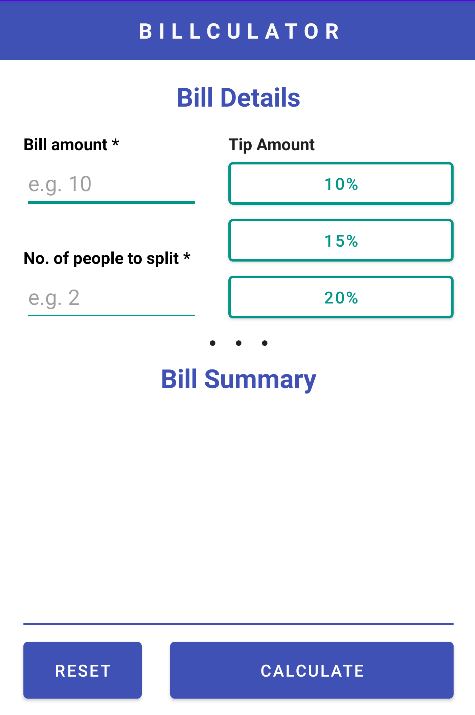
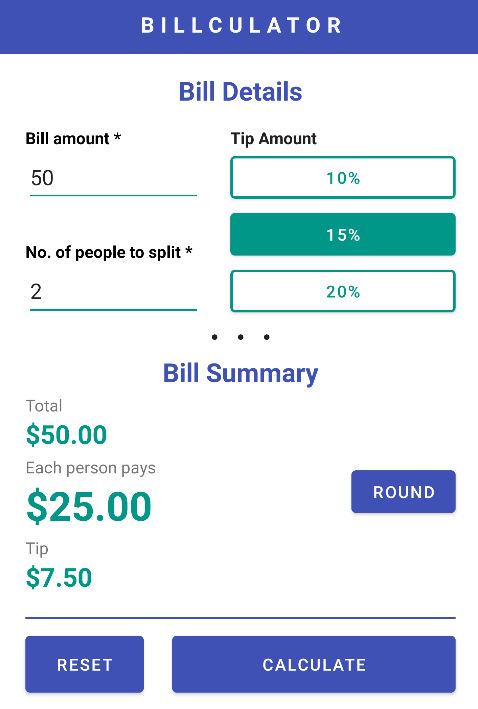
***Diagram 1.2.1:*** *Choosing a tip percentage and the result in the bill summary*

If the user had chosen a tip percentage previously and wants to deselect it, the user simply has to click on the same button that they have chosen again. This deselects the button and excludes the tip calculation process when the user presses the “Calculate” button.



***Diagram 1.2.2:*** *Deselecting a tip button*

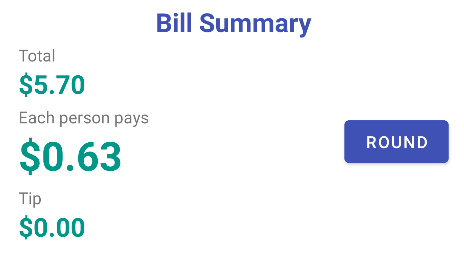
* **Reset function**



***Diagram 1.3.0:*** *Reset function in action*

Regardless of the view, the “Reset” button reverts the app back to its default view with all design components reverting back to holding their default attributes and values. This function is useful for users when they lost their sense of direction and flow of the app and wants to restart from the beginning, improving user satisfaction with the app.

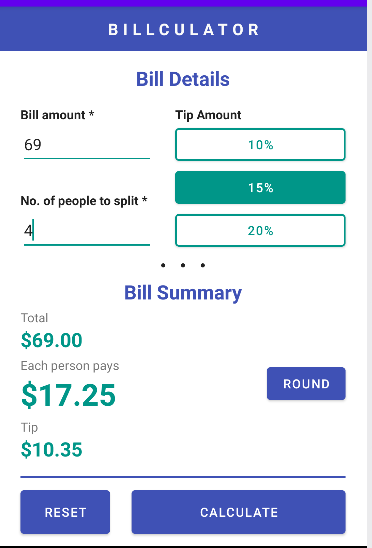
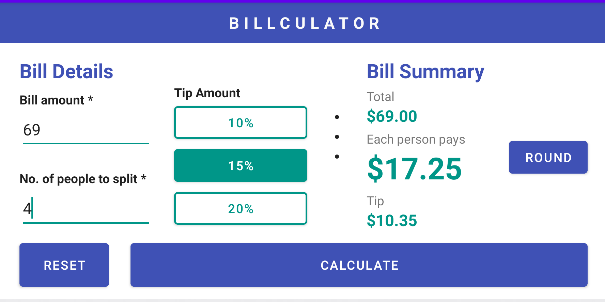
* **Round function**



***Diagram 1.4.0:*** *Round function in action*

The “Round” button is used to round the total bill amount in the bill summary up or down and determine the shared pay amount and tip based on the new bill total. It is displayed in the bill summary only when the user performs a successful bill calculation. Upon clicking the “Round” button, it disappears and the new total, shared pay amount, and tip are determined instantaneously. This feature is useful when users uses the app to calculate floating bill amounts.

* **Double orientation modes**



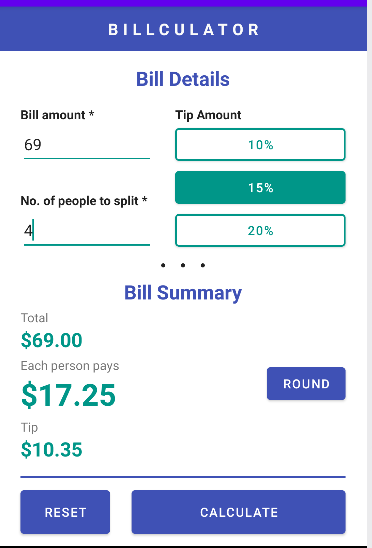
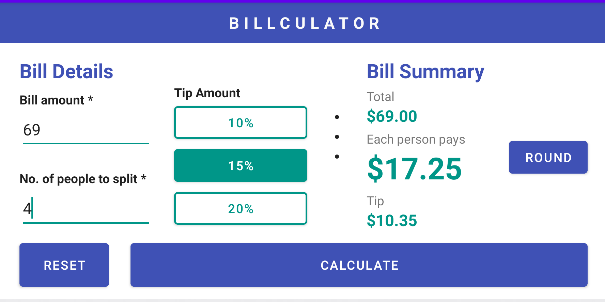
***Diagram 1.5.0:*** *Portrait and landscape orientations of Billculator*

Additionally, Billcalculator enables users to work on the app from both portrait and landscape orientations. So, users do not need to worry about the availability of the app if they need to set their phones in different orientation modes because of different circumstances.

**Strengths of the application**

The following points are some strengths that the application boasts:

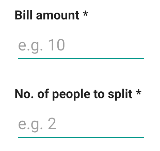
* **One-screen display**



***Diagram 2.1.0:*** *Single-screen view of Billculator in both orientations* ***(Device: Nexus)***

Billculator serves to reduce workloads of calculating bills. Therefore, only a single, simple screen is used for both portrait and landscape orientations to display all the necessary information that is needed for users to organize and plan their shares of the bill by appropriately utilizing the spaces of a single screen. Compared to using a normal calculator, users do not need look at their calculator history and remember the formulas and values that are needed to determine the answer; They only need to provide the required inputs for them to get the same correct result, reducing mathematical mistakes made from manual calculations.

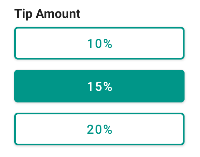
* **Hints to inputs**



***Diagram 2.2.0:*** *Input field hints* ***(Device: Nexus)***

Hints are provided in each input fields to guide users into entering valid data values that are expected from them. It helps users draw logical conclusions as to what they should enter in the input fields when they are learning the user interface initially.

* **Feasible tip controls**



***Diagram 2.3.0:*** *Tip buttons* ***(Device: Nexus)***

Although in the form of buttons, tip percentage controls can be used easily because they behave like radio buttons. Not only do they look aesthetically pleasing, but they also respond to user selections by displaying the responses clearly without any latency. It helps users recognize currently selected tip buttons even when they are using their peripheral vision.

* **Moderately large action button touch areas**



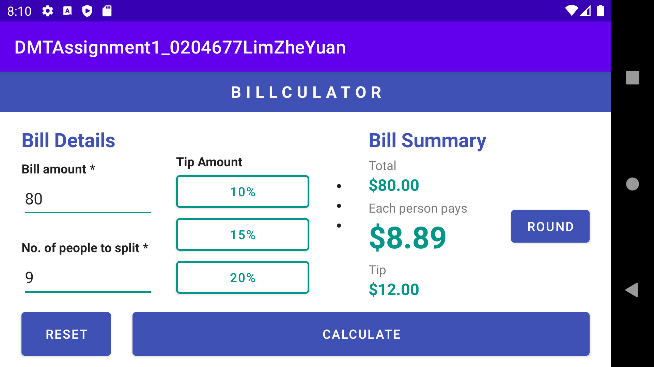
***Diagram 2.4.0:*** *Action buttons in portrait* ***(top)*** *and**landscape* ***(bottom)*** *orientation* ***(Device: Nexus)***

Action buttons like the “Reset” and “Calculate” buttons are intentionally enlarged in both portrait and landscape orientations to enable easier registration of button clicks. It is designed to increase accessibility of the action buttons to the user due to the fact that action buttons are more frequently used. The “Calculate” button is also intended to be larger than the “Reset” button to prevent users from accidentally pressing the “Reset” button which erases working input values.

**Flaws of the application**

Presently, there are still flaws that can be found from the app, the following are some of the flaws that are pending to be fixed:

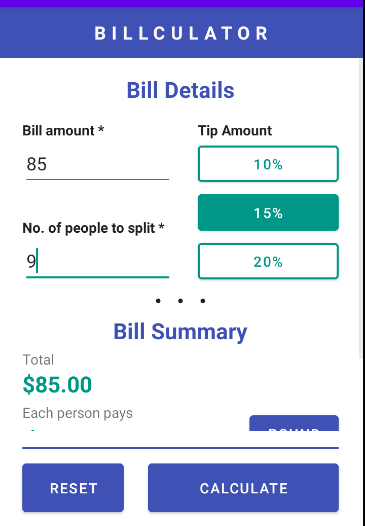
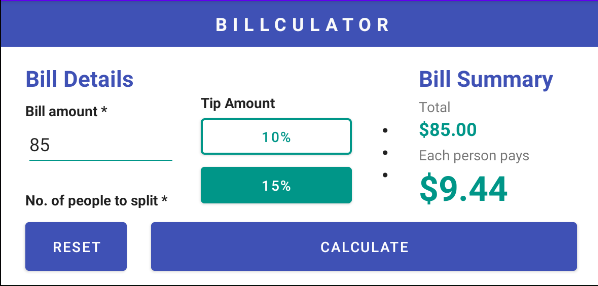
* **Small touch area for controls**



***Diagram 3.1.0:*** *Small touch area for input controls and “Round” button* ***(Device: Nexus)***

Due to displaying every element in one screen, control sizes in the app may be too small for users who have larger fingers to interact with the controls and may pose as a problem when working with the app. They would need to pay more attention to their finger positions to precisely land them onto the intended target, which would possibly cause frustrations because of higher rates of failing to interact with the app components.

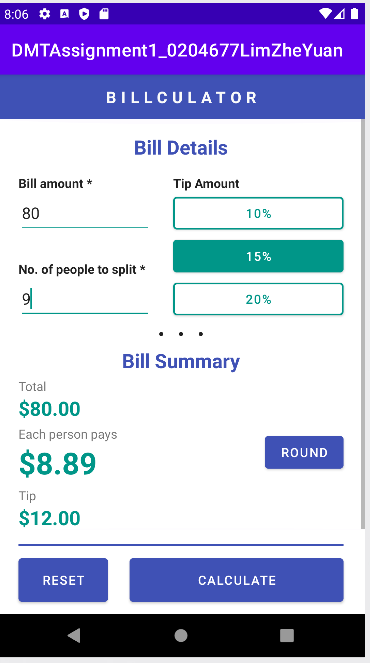
* **Views may abruptly cut**



***Diagram 3.2.0:*** *Bill summary is partially hidden is both orientations* ***(Device: Galaxy Nexus)***

Depending on the device used by the user, the app view may abruptly cut because of the device’s insufficient screen height. Although this does not pose any major problems because of having the ability to scroll down, users can no longer see full details of the bill summary at a glance and will need to scroll down to view hidden contents. This is an issue for users who wants to get instant computed results from the app but interacts with the app to a bare minimum.

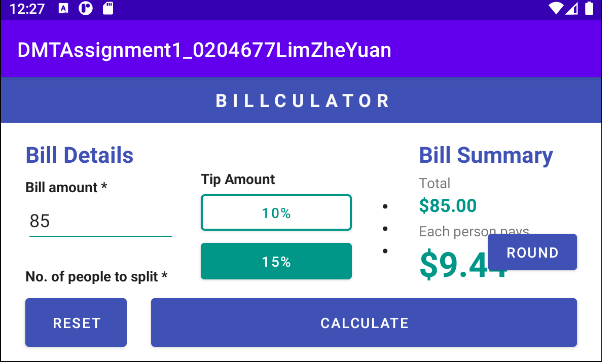
* **Scroll still appears with little overflow**



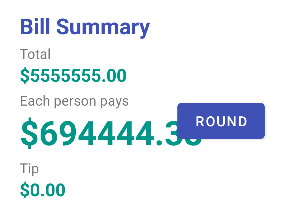
***Diagram 3.3.0:*** *Scrollbar appears even if there is a tiny height overflow* ***(Device: Pixel)***

Depending on the device used by the user, the app also enables users to scroll up and down by a tiny margin if their phone has almost enough, yet insufficient screen height. Although this does not pose any major problems, it may leave users confused and weirded out by the bizzare appearance of the scrollbar. This phenomenon impacts the app’s reputation as a whole and may cause users to lower user ratings and lose the app’s competitive edge.

* **View elements near the bill summary section may overlap**



***Diagram 3.4.0:*** *Button overlaps bill summary details if screen width is too short* ***(Device: Galaxy Nexus)***



***Diagram 3.4.1:*** *Data value is hidden behind the “Round” button if its length is too long* ***(Device: Nexus)***

Depending on the device used by the user, view elements near the bill summary section may overlap each other if the device’s screen width is too short or if the element lengths are too long. This is a major issue because users may not view the full value displayed in the bill summary and will not know the result computed by the system. It also adds visual clutter to the screen and cause user confusion over the user interface design, which impacts user performance and ultimately the popularity of the app.

**Appendix**

* **MainActivity.java**

package com.example.dmtassignment1\_0204677limzheyuan;  
  
import androidx.annotation.NonNull;  
import androidx.appcompat.app.AppCompatActivity;  
  
import android.graphics.Color;  
import android.os.Bundle;  
import android.view.KeyEvent;  
import android.view.View;  
import android.widget.Button;  
import android.widget.EditText;  
import android.widget.TextView;  
import android.widget.Toast;  
import java.util.InputMismatchException;  
  
public class MainActivity extends AppCompatActivity {  
 //Design views  
 private EditText txtBill, txtPeople;  
 private Button btnTip10, btnTip15, btnTip20, btnReset, btnCalculate, btnRound;  
 private TextView lblBill, lblPeople, lblTotal, lblTip,lblResult, resTotal, resTip,resResult;  
 //Global consts and vars  
 private final double minBillAmount = 2;  
 private int tipBtnSelected = -1;  
 private double tipPerc = 0;  
  
 @Override  
 public void onSaveInstanceState(@NonNull Bundle outState) {  
 //save states before orientation change  
 //values  
 outState.putString("resResult", resResult.getText().toString());  
 outState.putString("resTotal", resTotal.getText().toString());  
 outState.putString("resTip", resTip.getText().toString());  
 //selected buttons  
 outState.putInt("tipBtnSelected", tipBtnSelected);  
 //visibility  
 outState.putInt("vis\_lblResult", lblResult.getVisibility());  
 outState.putInt("vis\_lblTotal", lblTotal.getVisibility());  
 outState.putInt("vis\_lblTip", lblTip.getVisibility());  
 outState.putInt("vis\_resResult", resResult.getVisibility());  
 outState.putInt("vis\_resTotal", resTotal.getVisibility());  
 outState.putInt("vis\_resTip", resTip.getVisibility());  
 outState.putInt("vis\_btnRound", btnRound.getVisibility());  
 super.onSaveInstanceState(outState);  
 }  
  
 @Override  
 protected void onRestoreInstanceState(@NonNull Bundle savedInstanceState) {  
 super.onRestoreInstanceState(savedInstanceState);  
 //Restore previous values before orientation change:  
 //values  
 resResult.setText(savedInstanceState.getString("resResult"));  
 resTotal.setText(savedInstanceState.getString("resTotal"));  
 resTip.setText(savedInstanceState.getString("resTip"));  
  
 //selected buttons  
 int savedBtnSelected = savedInstanceState.getInt("tipBtnSelected");  
 //Set value of tip selected before orientation change  
 //A.N. Everything can be restored because onRestoreInstanceState is called AFTER onCreate  
 tipBtnSelected = savedBtnSelected;  
 if (savedBtnSelected == 0) {  
 tipPerc = 0.1;  
 changeButtonColor("selected", btnTip10);  
 } else if (savedBtnSelected == 1) {  
 tipPerc = 0.15;  
 changeButtonColor("selected", btnTip15);  
 } else if (savedBtnSelected == 2) {  
 tipPerc = 0.2;  
 changeButtonColor("selected", btnTip20);  
 } else {  
 tipPerc = 0;  
 deselectAllTipBtns();  
 }  
  
 //visibility  
 lblResult.setVisibility(savedInstanceState.getInt("vis\_lblResult"));  
 lblTip.setVisibility(savedInstanceState.getInt("vis\_lblTip"));  
 lblTotal.setVisibility(savedInstanceState.getInt("vis\_lblTotal"));  
 resResult.setVisibility(savedInstanceState.getInt("vis\_resResult"));  
 resTip.setVisibility(savedInstanceState.getInt("vis\_resTip"));  
 resTotal.setVisibility(savedInstanceState.getInt("vis\_resTotal"));  
 btnRound.setVisibility(savedInstanceState.getInt("vis\_btnRound"));  
 }  
  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.*activity\_main*);  
  
 //Set reference to design elements  
 lblBill = findViewById(R.id.*lblBill*);  
 lblPeople = findViewById(R.id.*lblPeople*);  
 txtBill = findViewById(R.id.*txtBill*);  
 txtPeople = findViewById(R.id.*txtPeople*);  
 btnTip10 = findViewById(R.id.*btnTip10*);  
 btnTip15 = findViewById(R.id.*btnTip15*);  
 btnTip20 = findViewById(R.id.*btnTip20*);  
 btnReset = findViewById(R.id.*btnReset*);  
 btnCalculate = findViewById(R.id.*btnCalculate*);  
 btnRound = findViewById(R.id.*btnRound*);  
 lblTotal = findViewById(R.id.*lblSummTotal*);  
 lblTip = findViewById(R.id.*lblSummTip*);  
 lblResult = findViewById(R.id.*lblSummResult*);  
 resTotal = findViewById(R.id.*resTotal*);  
 resTip = findViewById(R.id.*resTip*);  
 resResult = findViewById(R.id.*resResult*);  
  
 //Set event listeners  
 //Error recovery - restore label text colour if user reenters input in text fields after an error  
 txtBill.setOnKeyListener(new View.OnKeyListener() {  
 @Override  
 public boolean onKey(View view, int i, KeyEvent keyEvent) {  
 changeControlColor("normal", lblBill, txtBill);  
 return false;  
 }  
 });  
 txtPeople.setOnKeyListener(new View.OnKeyListener() {  
 @Override  
 public boolean onKey(View view, int i, KeyEvent keyEvent) {  
 changeControlColor("normal", lblPeople, txtPeople);  
 return false;  
 }  
 });  
  
 //Set button event listeners  
 btnTip10.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View view) {  
 //If it is already selected, deselect it  
 if (tipBtnSelected == 0) {  
 changeButtonColor("deselected", btnTip10);  
 tipBtnSelected = -1;  
 } else {  
 //Deselect any selected buttons  
 deselectAllTipBtns();  
 //Set bg and text color  
 changeButtonColor("selected", btnTip10);  
 //Set selected button value  
 tipBtnSelected = 0;  
 }  
 }  
 });  
  
  
 btnTip15.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View view) {  
 //If it is already selected, deselect it  
 if (tipBtnSelected == 1) {  
 changeButtonColor("deselected", btnTip15);  
 tipBtnSelected = -1;  
 } else {  
 //Deselect any selected buttons  
 deselectAllTipBtns();  
 //Set bg and text color  
 changeButtonColor("selected", btnTip15);  
 //Set selected button value  
 tipBtnSelected = 1;  
 }  
 }  
 });  
  
 btnTip20.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View view) {  
 //If it is already selected, deselect it  
 if (tipBtnSelected == 2) {  
 changeButtonColor("deselected", btnTip20);  
 tipBtnSelected = -1;  
 } else {  
 //Deselect any selected buttons  
 deselectAllTipBtns();  
 //Set bg and text color  
 changeButtonColor("selected", btnTip20);  
 //Set selected button value  
 tipBtnSelected = 2;  
 }  
 }  
 });  
  
 btnReset.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View view) {  
 //Reset controls  
 deselectAllTipBtns();  
 changeControlColor("normal", lblBill, txtBill);  
 changeControlColor("normal", lblPeople, txtPeople);  
 txtBill.setText("");  
 txtPeople.setText("");  
 //Hide details  
 setSummaryVisible(false);  
 }  
 });  
  
 btnCalculate.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View view) {  
 try {  
 //Declare result containers  
 double tipAmount, splitTotal;  
 //Get inputs  
 double billAmount = Double.*parseDouble*(txtBill.getText().toString());  
 int noOfPeople = Integer.*parseInt*(txtPeople.getText().toString());  
 //Input validation  
 // Bill amount  
 if (billAmount < minBillAmount) throw new InputMismatchException();  
 // No of people  
 if (noOfPeople < 2) throw new InputMismatchException();  
  
 //Get selected button  
 if (tipBtnSelected == 0)  
 tipPerc = 0.1;  
 else if (tipBtnSelected == 1)  
 tipPerc = 0.15;  
 else if (tipBtnSelected == 2)  
 tipPerc = 0.2;  
 else {  
 tipPerc = 0;  
 }  
  
 //Process values  
 splitTotal = billAmount / noOfPeople;  
 tipAmount = billAmount \* tipPerc;  
  
 //Assign values to output before turning them visible  
 resResult.setText(getString(R.string.*billResult*, splitTotal));  
 resTip.setText(getString(R.string.*billResult*, tipAmount));  
 resTotal.setText(getString(R.string.*billResult*, billAmount));  
 setSummaryVisible(true);  
 } catch (InputMismatchException ime) {  
 //Two possible error case:  
 // -Bill amount < min. bill amount  
 // -No of people < 2  
 double billAmount = Double.*parseDouble*(txtBill.getText().toString());  
 int noOfPeople = Integer.*parseInt*(txtPeople.getText().toString());  
  
 if (billAmount < minBillAmount) {  
 //throw error when bill amount < 2 because value too tiny for calculation  
 Toast.*makeText*(getApplicationContext(), "Minimum bill amount: 2!", Toast.*LENGTH\_SHORT*).show();  
 changeControlColor("error", lblBill, txtBill);  
 txtBill.requestFocus();  
 } else if (noOfPeople < 2) {  
 //throw error when no. of people < 2 to avoid arithmetic error  
 Toast.*makeText*(getApplicationContext(), "Minimum person count: 2!", Toast.*LENGTH\_SHORT*).show();  
 changeControlColor("error", lblPeople, txtPeople);  
 txtPeople.requestFocus();  
 }  
 } catch (Exception e) {  
 //throw error when input is empty  
 Toast.*makeText*(getApplicationContext(), "Please fill in all text fields!", Toast.*LENGTH\_SHORT*).show();  
 //Highlight empty inputs  
 if (txtPeople.getText().toString().equals("")) {  
 changeControlColor("error", lblPeople, txtPeople);  
 txtPeople.requestFocus();  
 }  
 if (txtBill.getText().toString().equals("")) {  
 changeControlColor("error", lblBill, txtBill);  
 txtBill.requestFocus();  
 }  
 }  
 }  
 });  
  
 btnRound.setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View view) {  
 //Declare vars  
 double newSplitTotal, total, roundedTotal, newTip;  
 int noOfPeople;  
 //Get value after '$' sign  
 total = Double.*parseDouble*(resTotal.getText().toString().substring(1));  
 noOfPeople = Integer.*parseInt*(txtPeople.getText().toString());  
  
 //Round the total and calculate new split total and tip  
 roundedTotal = Math.*round*(total);  
 newSplitTotal = roundedTotal / noOfPeople;  
 newTip = roundedTotal \* tipPerc;  
 //Set round values to labels  
 resTotal.setText(getString(R.string.*billResult*, roundedTotal));  
 resResult.setText(getString(R.string.*billResult*, newSplitTotal));  
 resTip.setText(getString(R.string.*billResult*, newTip));  
 //Hide button  
 btnRound.setVisibility(View.*GONE*);  
 }  
 });  
 }  
  
 private void setSummaryVisible(boolean visible) {  
 if (visible) {  
 //Set everything visible  
 lblResult.setVisibility(View.*VISIBLE*);  
 lblTip.setVisibility(View.*VISIBLE*);  
 lblTotal.setVisibility(View.*VISIBLE*);  
 resResult.setVisibility(View.*VISIBLE*);  
 resTip.setVisibility(View.*VISIBLE*);  
 resTotal.setVisibility(View.*VISIBLE*);  
 btnRound.setVisibility(View.*VISIBLE*);  
 } else {  
 //Set everything invisible  
 lblResult.setVisibility(View.*GONE*);  
 lblTip.setVisibility(View.*GONE*);  
 lblTotal.setVisibility(View.*GONE*);  
 resResult.setVisibility(View.*GONE*);  
 resTip.setVisibility(View.*GONE*);  
 resTotal.setVisibility(View.*GONE*);  
 btnRound.setVisibility(View.*GONE*);  
 }  
 }  
  
 private void changeButtonColor(String state, Button btn) {  
 if (state.equals("selected")) {  
 //Set bg green and text white  
 btn.setBackgroundTintList(getColorStateList(R.color.*greenClr*));  
 btn.setTextColor(getColorStateList(R.color.*white*));  
 } else if (state.equals("deselected")) {  
 //Set text green and bg white  
 btn.setBackgroundTintList(getColorStateList(R.color.*white*));  
 btn.setTextColor(getColorStateList(R.color.*greenClr*));  
 }  
 }  
  
 private void changeControlColor(String state, TextView lbl, EditText input) {  
 if (state.equals("normal")) {  
 //Set default color - lbl black and input green  
 lbl.setTextColor(Color.*BLACK*);  
 input.setBackgroundTintList(getColorStateList(R.color.*greenClr*));  
 } else if (state.equals("error")) {  
 //Set both lbl and input color red  
 lbl.setTextColor(getColorStateList(R.color.*errorClr*));  
 input.setBackgroundTintList(getColorStateList(R.color.*errorClr*));  
 }  
 }  
  
 private void deselectAllTipBtns() {  
 //Set bg colour  
 //Set text colour  
 //btnTip10  
 btnTip10.setBackgroundTintList(getColorStateList(R.color.*white*));  
 btnTip10.setTextColor(getColorStateList(R.color.*greenClr*));  
 //btnTip15  
 btnTip15.setBackgroundTintList(getColorStateList(R.color.*white*));  
 btnTip15.setTextColor(getColorStateList(R.color.*greenClr*));  
 //btnTip20  
 btnTip20.setBackgroundTintList(getColorStateList(R.color.*white*));  
 btnTip20.setTextColor(getColorStateList(R.color.*greenClr*));  
 //Set selected value to -1  
 tipBtnSelected = -1;  
 }  
}

**Marking Rubric for Assignment**

The assignment will be marked based on the following criteria. Include this sheet as the last page of your submission.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Criteria** | **Very Poor** | **Poor** | **Good** | **Expert** | **Allocated Marks** |
| **Fitness of Purposes & Functionality**  **- Event handling**  **- Output in multiline textview**  **- Validation** | (0-15)  Little or no attempt to implement the feature correctly | (16-23)  A partial implementation of the feature, but some aspects are incorrect and not particularly well coded. May give rise to run-time errors. | (24-32)  A mostly complete implementation of the feature which works correctly, although the coding could be clearer. | (33-40)  A complete implementation of the feature, clearly coded. |  |
| **Build Quality & Usability**  **- Layouts**  **- Model**  **- Data structure usage** | (0-15)  Poor build quality provided according to the Android and general coding standard. Weak in system feddbacks and no input validation. | (16-23)  Fair build quality provided according to the Android and general coding standard. Ambigous in system feedbakcs and minor implementation in input validation | (24-32)  Good build quality provided according to the Android and general coding standard. Good system feedback given with appropriate input validation | (33-40)  Excellent build quality provided according to the Android and coding standard. Provide very clear feedbacks and excellent in input validation. |  |
| **Program code structure (using classes, methos, code indentation & commenting)** | (0-3)  Weak or no program structure and commenting code provided | (4-5)  Poor/Average program structure and commenting code provided | (6-8)  Good program structure and commenting code provided | (9-10)  Excellent program structure and commenting code provided |  |
| **Individual Report** | (0-3)  Little or no evaluation is offered of the work undertaken | (4-5)  Some attempt at an evaluation of the work undertaken. However, the extent or quality is inadequate. | (6-8)  A complete evaluation of the work undertaken. Some outstanding features but not wholly outstanding. | (9-10)  A outstanding evaluation of the work undertaken |  |