FORMAN CHRISTIAN COLLEGE (A CHARTERED UNIVERSITY)

Department of Computer Science

COMP 468(A) ASSIGNMENT REPORT # 2

CACULATOR APP

JOSHUA SADAQAT [240-545460]

Instructor: Mr. Adeem Akhtar

Session 2020-2024

INTRODUCTION

This report presents an analysis of **Josh's Calculator App**, created as part of **Assignment 2** for the **Mobile Application Development** course. The project builds on previous lab work by incorporating a basic calculator, which was then expanded into a more advanced version with additional features such as error handling, decimal input, and a user-friendly interface. The app allows users to switch between two modes: Basic (Lab Calculator) and Advanced (Assignment Calculator).

The following sections will outline the functionality of both calculators, including a detailed analysis of each function in the advanced calculator.

PROJECT REQUIREMENTS

The key project requirements include:

- **Interface Design**: Create a visually appealing and user-friendly layout with buttons for numbers, arithmetic operations, a decimal point, and a display for results.
- **Functional Requirements**: Implement basic operations (addition, subtraction, multiplication, division), handle decimal input, and manage transitions between activities (Basic and Advanced calculators).
- **Development**: Follow coding best practices, ensure modular design, and implement robust error handling.
- **Testing**: Perform testing for different screen sizes, orientations, and edge cases.

LAB CALCULATOR (BASIC MODE)

The **Lab Calculator**, implemented in MainActivity.kt, performs simple arithmetic operations. It provides the foundation for the more advanced Assignment Calculator.

UI Design

- The layout includes two input fields for numbers
  - (editText1 and editText2)
  - four buttons for arithmetic operations (+, -, *, /)

- There is "**Result"** Output which shows the results for the operations performed on these two numbers.
- There is a button labelled **"Josh's Calculator"** which allows the user to transition to the advanced calculator.

Functionality

- **Arithmetic Operations**: The app performs basic operations using button click events handled by the onClick() method. Each button performs a specific operation (addition, subtraction, multiplication, or division) based on the button's ID:

CODE:

```
override fun onClick(v: View?) {
    val num1 = editText1.text.toString().toDoubleOrNull()
    val num2 = editText2.text.toString().toDoubleOrNull()

    if (num1 == null || num2 == null) {
        resultTextView.text = "Please enter valid numbers"
        return
    }

    val result = when (v?.id) {
        R.id.btn_add -> num1 + num2
        R.id.btn_sub -> num1 - num2
        R.id.btn_mul -> num1 * num2
        R.id.btn_div -> {
            if (num2 != 0.0) num1 / num2 else "Cannot divide by zero"
        }
        else -> "Unknown operation"
    }

    resultTextView.text = "Result is: $result"
}
```

- **Division by Zero**: The app includes a check to prevent division by zero. If the second number (num2) is zero during division, it returns an error message ("Cannot divide by zero").

- **Activity Transition**: A button labeled **"Josh's Calculator"** allows the user to switch to the advanced calculator. This is achieved through an **Intent** that triggers the transition from the basic mode to the advanced mode:

  CODE:

```
// Set onClick listener for the Josh's Calculator button
btnJoshsCalculator.setOnClickListener {
    // Redirect to the JoshsCalculator activity
    val intent = Intent( packageContext: this@MainActivity, JoshsCalculator::class.java)
    startActivity(intent)
}
```

ASSIGNMENT CALCULATOR (ADVANCED MODE)

The **Assignment Calculator** (JoshsCalculator.kt) is a more advanced version of the calculator that supports decimal input, percentage calculations, backspace functionality, and error handling. It builds upon the basic calculator but with significant enhancements in both functionality and UI design.

UI Design

- The advanced calculator includes buttons for numbers, operators (+, -, *, /, %), a decimal point (.), two equal buttons (btnEquals1 and btnEquals2), and a backspace button.
- The layout is designed to be visually appealing and responsive to different screen orientations (portrait and landscape).

Functionality

Each function in the advanced calculator serves a specific role in handling user input, performing calculations, and managing errors. Below is a breakdown of the key functions:

onDigit(digit: String)

This function appends the pressed digit to the calculator's input display (tvInput), replacing the initial "0" if it's the first input. It ensures no more than 14 characters are entered to avoid overflow.

CODE:

```
// Appends the digit pressed to the input or replaces the initial "0"
@SuppressLint("SetTextI18n")
private fun onDigit(digit: String) {
    if (tvInput.text.length < 14) {
        if (stateError) {
            tvInput.text = digit
            stateError = false
        } else if (tvInput.text.toString() == "0") {
            // Replace "0" with the first digit
            tvInput.text = digit
        } else {
            tvInput.append(digit)
        }
        lastNumeric = true
    } else {
        tvInput.text = "Error = Max 14 digits allowed."
    }
}
```

Purpose: Adds digits to the input string, replacing the initial zero if necessary. Limits input length to prevent errors.

---

onOperator(operator: String)

This function appends an operator (+, -, *, /, %) to the input string, provided the last input was numeric. It also ensures that only one operator is added consecutively.

CODE:

```
private fun onOperator(operator: String) {
    if (lastNumeric && !stateError) {
        tvInput.append(operator)
        lastNumeric = false
        lastDot = false
    }
}
```

Purpose: Handles operator input by appending it to the current input only if the last character is a number, ensuring a valid expression is being formed.

---

onDecimalPoint()

This function adds a decimal point to the input string. It checks that a decimal has not already been added and that the previous input was numeric.

CODE:

```
private fun onDecimalPoint() {
    if (lastNumeric && !stateError && !lastDot) {
        tvInput.append(".")
        lastNumeric = false
        lastDot = true
    }
}
```

Purpose: Ensures that only one decimal point can be added to a number, preventing multiple decimal points in a single numeric value.

---

onClear()

This function resets the input display to "0" and clears any error states, preparing the calculator for a new calculation.

CODE:

```
private fun onClear() {
    tvInput.text = "0"
    lastNumeric = false
    stateError = false
    lastDot = false
}
```

Purpose: Clears the current input and resets the calculator to its initial state, ready for new operations.

---

onBackspace()

This function removes the last character from the input string. If all characters are deleted, it resets the input to "0."

CODE:

```
private fun onBackspace() {
    val text = tvInput.text.toString()
    if (text.isNotEmpty() && text != "0") {
        tvInput.text = text.dropLast( n: 1)
    }
    if (tvInput.text.isEmpty()) {
        tvInput.text = "0"
    }
}
```

Purpose: Allows the user to delete the last character entered, similar to a backspace function on a physical calculator.

---

onEqual()

This function evaluates the current mathematical expression entered by the user. It parses the input string, replacing percentage operators (%) with equivalent expressions (*0.01), and calculates the result using the evaluate() function.

CODE:

```
private fun onEqual() {
    if (lastNumeric && !stateError) {
        try {
            val expression = tvInput.text.toString().replace( oldValue: "%",  newValue: "*0.01")
            val result = evaluate(expression)
            tvInput.text = result.toString()
        } catch (e: Exception) {
            tvInput.text = "Error"
            stateError = true
            lastNumeric = false
        }
    }
}
```

Purpose: Evaluates the entire mathematical expression and updates the display with the result. Handles errors gracefully by displaying an error message if the expression is invalid.

---

onSaveInstanceState()

Functionality: This function saves the current state of the calculator when the screen is rotated or the app is temporarily closed. It stores the current input (`tvInput.text`).

CODE:

```
override fun onSaveInstanceState(outState: Bundle) {
    super.onSaveInstanceState(outState)
    // Save the current input and result
    outState.putString("inputText", tvInput.text.toString())
}
```

Explanation: Ensures that the current state of the calculator (including user input) is saved and can be restored when the app is reopened or the device is rotated.

---

onRestoreInstanceState()

Functionality: This function restores the previously saved state when the app is reopened or the device is rotated, allowing users to continue where they left off.

CODE:

```
override fun onRestoreInstanceState(savedInstanceState: Bundle) {
    super.onRestoreInstanceState(savedInstanceState)
    // Restore the saved input and result
    val inputText = savedInstanceState.getString( key: "inputText")
    if (inputText != null) {
        tvInput.text = inputText
    }
```

Explanation: Restores the input field to its previous state, ensuring that the user's progress is not lost during configuration changes like screen rotations.

---

evaluate (expression: String)

This function evaluates the mathematical expression entered by the user. It uses a recursive descent parser to process and calculate the result.

CODE:

```
// Parses and evaluates the expression
private fun evaluate(expression: String): Double {
    return object : Any() {
        var pos = -1
        var ch = 0

        fun nextChar() {...}

        fun eat(charToEat: Int): Boolean {...}

        fun parse(): Double {...}

        fun parseExpression(): Double {...}

        fun parseTerm(): Double {...}

        fun parseFactor(): Double {...}
    }.parse()
}
```

nextChar()

Moves the parser to the next character in the expression string.

CODE:

```
fun nextChar() {
    ch = if (++pos < expression.length) expression[pos].code else -1
}
```

eat(charToEat: Int): Boolean

Skips over characters (like spaces) and checks if the current character matches the expected one (charToEat). If so, it advances the pointer and returns true.

CODE:

```
fun eat(charToEat: Int): Boolean {
    while (ch == ' '.code) nextChar()
    if (ch == charToEat) {
        nextChar()
        return true
    }
    return false
}
```

parse()

Initiates parsing of the expression by first advancing to the next character, then parsing the entire expression using parseExpression(). It throws an error if an unexpected character is encountered.

CODE:

```
fun parse(): Double {
    nextChar()
    val x = parseExpression()
    if (pos < expression.length) throw RuntimeException("Unexpected: " + ch.toChar())
    return x
}
```

parseExpression()

Handles parsing of addition and subtraction. It first parses a term (which can include multiplication and division) and then checks for the presence of addition (+) or subtraction (-).

CODE:

```
fun parseExpression(): Double {
    var x = parseTerm()
    while (true) {
        when {
            eat('+'.code) -> x += parseTerm() // addition
            eat('-'.code) -> x -= parseTerm() // subtraction
            else -> return x
        }
    }
}
```

parseTerm()

Handles multiplication and division. It first parses a factor (a number or expression inside parentheses) and then checks for multiplication (*) or division (/).

CODE:

```
fun parseTerm(): Double {
    var x = parseFactor()
    while (true) {
        when {
            eat('*'.code) -> x *= parseFactor() // multiplication
            eat('/'.code) -> x /= parseFactor() // division
            else -> return x
        }
    }
}
```

parseFactor()

Parses individual numbers, parentheses, and unary operators (like + or -). It checks for parentheses and recursively parses expressions inside them.

CODE:

```
fun parseFactor(): Double {
    if (eat('+'.code)) return parseFactor() // unary plus
    if (eat('-'.code)) return -parseFactor() // unary minus

    val x: Double
    val startPos = pos
    if (eat('('.code)) { // parentheses
        x = parseExpression()
        eat(')'.code)
    } else if (ch >= '0'.code && ch <= '9'.code || ch == '.'.code) {
        while (ch >= '0'.code && ch <= '9'.code || ch == '.'.code) nextChar()
        x = expression.substring(startPos, pos).toDouble()
    } else {
        throw RuntimeException("Unexpected: " + ch.toChar())
    }

    return x
}
```

Explanation: This recursive function handles the parsing and evaluation of the entire expression, ensuring the correct order of operations (i.e., multiplication/division before addition/subtraction).

TESTING AND DEBUGGING

The app has been thoroughly tested across various screen sizes and orientations (both portrait and landscape). It successfully handles edge cases such as:

- Handling large numbers
- Inputting multiple operators or decimal points
- Division by zero
- Clearing input and backspace functionality

The testing phase also ensured that the app's UI remains responsive and functional in both orientations, providing a smooth user experience.

[ATTACHED PLAESAE FIND THE UI & OUTPUT SS IN THE PDF FILE]

COMPARISON BETWEEN LAB CALCULATOR & ASSIGNMENT CALCULATOR

Feature	Lab Calculator	Assignment Calculator
Basic Arithmetic Operations	Supports [+, -, *, /]	Supports [+, -, *, /, %]
Decimal Handling	Not supported	Supported
Character Deletion	Not supported	Supports backspace functionality
Error Handling	Basic error handling for division by zero	Advanced error handling with character limits and invalid input prevention
Complex Expressions	Not supported	Supports evaluation of complex expressions
Digit Limitation	No limit	Limits input to 14 digits
UI Design	Basic and functional	Visually enhanced with advanced features

---

CONCLUSION

Thus this Josh's Calculator App meets the requirements for **Assignment 2**. The app showcases a progression from a basic Lab Calculator to a more advanced Assignment Calculator. The advanced features, including error handling, expression evaluation, and backspace functionality, enhance the user experience and provide robust handling of various mathematical operations.

The assignment demonstrates a solid understanding of UI/UX design, event handling, and mobile app development best practices.