CS6057: Mobile Applications
Calculator Application

Baba-Femi Sandro Mark [ID: 20027019]

12/05/2024

1 INTRODUCTION

This project involves the design, development, and implementation of an Android application that serves as a calculator with additional features for saving and managing calculation results. The application aims to provide a functional and user-friendly experience by incorporating multiple screens, dynamic user interactions, persistent storage, and network communication. The development of this application demonstrates practical knowledge and skills in Android development, adhering to best practices and guidelines.

2 REQUIREMENTS

The application must fulfill the following requirements:

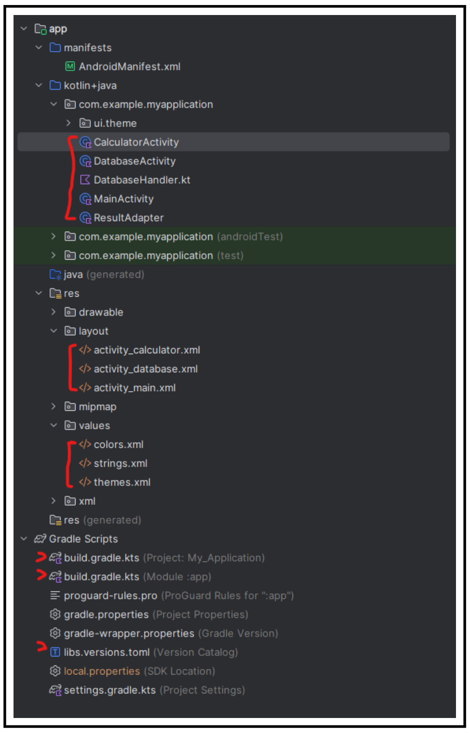
- Incorporate multiple screens (activities/fragments)
- Add methods to switch between screens
- Formulate screen with calculator
- Enable dynamic user interaction
- Implement persistent storage using a SQLite database

3 DESIGN AND DEVELOPMENT

3.1 SOFTWARE ARCHITECTURE

The software architecture of the application is based on the Model-View-Controller (MVC) pattern. This architectural pattern separates the application into three main components:

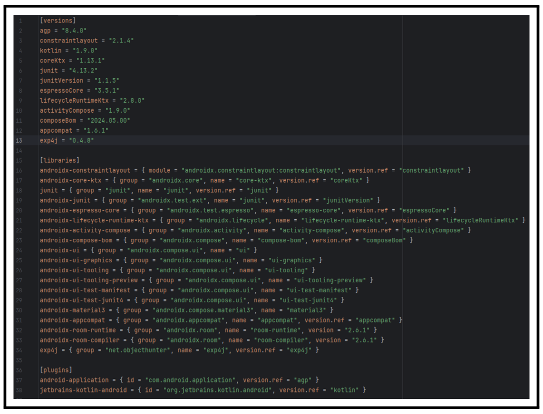
- **Model**: Manages the data and business logic of the application.
- **View**: Handles the presentation layer and user interface.
- **Controller**: Acts as an intermediary, processing user inputs and updating the Model and View accordingly.



```
app
  manifests
    AndroidManifest.xml
  kotlin+java
    com.example.myapplication
      ui.theme
      CalculatorActivity
      DatabaseActivity
      DatabaseHandler.kt
      MainActivity
      ResultAdapter
    com.example.myapplication (androidTest)
    com.example.myapplication (test)
  java (generated)
  res
    drawable
    layout
      activity_calculator.xml
      activity_database.xml
      activity_main.xml
    mipmap
    values
      colors.xml
      strings.xml
      themes.xml
    xml
  res (generated)
Gradle Scripts
  build.gradle.kts (Project: My_Application)
  build.gradle.kts (Module :app)
  proguard-rules.pro (ProGuard Rules for ":app")
  gradle.properties (Project Properties)
  gradle-wrapper.properties (Gradle Version)
  libs.versions.toml (Version Catalog)
  local.properties (SDK Location)
  settings.gradle.kts (Project Settings)
```

Figure 1: Project Directory, highlighted files are user created/modified files

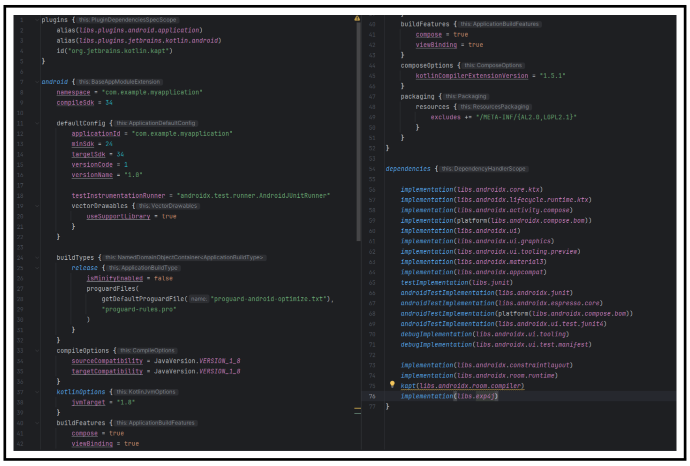
Many functions and command will require libraries to operate, otherwise the program will fail to compile due to errors when these commands are used. The libraries used are all listed in the *libs.versions.toml* file which can be seen in Figure 2. Note that the version numbers can be placed directly into the libray entries but for ease of updating they are placed as variables in one location.



```toml
[versions]
agp = "8.4.0"
constraintlayout = "2.1.4"
kotlin = "1.9.0"
coreKtx = "1.13.1"
junit = "4.13.2"
junitVersion = "1.1.5"
espressoCore = "3.5.1"
lifecycleRuntimeKtx = "2.8.0"
activityCompose = "1.9.0"
composeBom = "2024.05.00"
appcompat = "1.6.1"
exp4j = "0.4.8"

[libraries]
androidx-constraintlayout = { module = "androidx.constraintlayout:constraintlayout", version.ref = "constraintlayout" }
androidx-core-ktx = { group = "androidx.core", name = "core-ktx", version.ref = "coreKtx" }
junit = { group = "junit", name = "junit", version.ref = "junit" }
androidx-junit = { group = "androidx.test.ext", name = "junit", version.ref = "junitVersion" }
androidx-espresso-core = { group = "androidx.test.espresso", name = "espresso-core", version.ref = "espressoCore" }
androidx-lifecycle-runtime-ktx = { group = "androidx.lifecycle", name = "lifecycle-runtime-ktx", version.ref = "lifecycleRuntimeKtx" }
androidx-activity-compose = { group = "androidx.activity", name = "activity-compose", version.ref = "activityCompose" }
androidx-compose-bom = { group = "androidx.compose", name = "compose-bom", version.ref = "composeBom" }
androidx-ui = { group = "androidx.compose.ui", name = "ui" }
androidx-ui-graphics = { group = "androidx.compose.ui", name = "ui-graphics" }
androidx-ui-tooling = { group = "androidx.compose.ui", name = "ui-tooling" }
androidx-ui-tooling-preview = { group = "androidx.compose.ui", name = "ui-tooling-preview" }
androidx-ui-test-manifest = { group = "androidx.compose.ui", name = "ui-test-manifest" }
androidx-ui-test-junit4 = { group = "androidx.compose.ui", name = "ui-test-junit4" }
androidx-material3 = { group = "androidx.compose.material3", name = "material3" }
androidx-appcompat = { group = "androidx.appcompat", name = "appcompat", version.ref = "appcompat" }
androidx-room-runtime = { group = "androidx.room", name = "room-runtime", version = "2.6.1" }
androidx-room-compiler = { group = "androidx.room", name = "room-compiler", version = "2.6.1" }
exp4j = { group = "net.objecthunter", name = "exp4j", version.ref = "exp4j" }

[plugins]
android-application = { id = "com.android.application", version.ref = "agp" }
jetbrains-kotlin-android = { id = "org.jetbrains.kotlin.android", version.ref = "kotlin" }
```

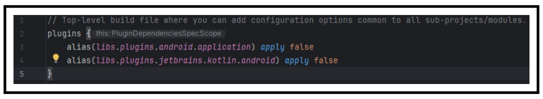
Figure 2: libs.versions.toml

Its also imperative to make sure all dependencies are consolidated in the *build.gradle.kts* file. There are two of these files and each has a different scope; one is project wide the other is module wide. *libs.versions.toml* is also important as it is used by the gradle files to acquiesce the required libraries which the gradle files will implement.



```kotlin
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.jetbrains.kotlin.android)
    id("org.jetbrains.kotlin.kapt")
}

android {
    namespace = "com.example.myapplication"
    compileSdk = 34

    defaultConfig {
        applicationId = "com.example.myapplication"
        minSdk = 24
        targetSdk = 34
        versionCode = 1
        versionName = "1.0"

        testInstrumentationRunner = "androidx.test.runner.AndroidJUnitRunner"
        vectorDrawables {
            useSupportLibrary = true
        }
    }

    buildTypes {
        release {
            isMinifyEnabled = false
            proguardFiles(
                getDefaultProguardFile("proguard-android-optimize.txt"),
                "proguard-rules.pro"
            )
        }
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_1_8
        targetCompatibility = JavaVersion.VERSION_1_8
    }
    kotlinOptions {
        jvmTarget = "1.8"
    }
    buildFeatures {
        compose = true
        viewBinding = true
    }
    composeOptions {
        kotlinCompilerExtensionVersion = "1.5.1"
    }
    packaging {
        resources {
            excludes += "/META-INF/{AL2.0,LGPL2.1}"
        }
    }
}

dependencies {

    implementation(libs.androidx.core.ktx)
    implementation(libs.androidx.lifecycle.runtime.ktx)
    implementation(libs.androidx.activity.compose)
    implementation(platform(libs.androidx.compose.bom))
    implementation(libs.androidx.ui)
    implementation(libs.androidx.ui.graphics)
    implementation(libs.androidx.ui.tooling.preview)
    implementation(libs.androidx.material3)
    implementation(libs.androidx.appcompat)
    testImplementation(libs.junit)
    androidTestImplementation(libs.androidx.junit)
    androidTestImplementation(libs.androidx.espresso.core)
    androidTestImplementation(platform(libs.androidx.compose.bom))
    androidTestImplementation(libs.androidx.ui.test.junit4)
    debugImplementation(libs.androidx.ui.tooling)
    debugImplementation(libs.androidx.ui.test.manifest)

    implementation(libs.androidx.constraintlayout)
    implementation(libs.androidx.room.runtime)
    kapt(libs.androidx.room.compiler)
    implementation(libs.exp4j)
}
```

Figure 3: build.gradle.kts (Module Scope)



```kotlin
// Top-level build file where you can add configuration options common to all sub-projects/modules.
plugins {
    alias(libs.plugins.android.application) apply false
    alias(libs.plugins.jetbrains.kotlin.android) apply false
}
```

Figure 4: build.gradle.kts (Project Scope)

3.2 DATA ARCHITECTURE

The data architecture involves using an SQLite database to store user information and application data. The database schema includes a table named 'Results' with columns for 'id', 'expression', and 'result'. Data Access Objects (DAOs) are used to perform CRUD (Create, Read, Update, Delete) operations on the database, ensuring efficient data management and retrieval.

3.3 INTERFACE DESIGN

The interface design focuses on providing a user-friendly and intuitive experience. Key design principles include the main dashboard, calculator interface, and results list. The calculator will use buttons and a text box as a GUI whilst the database will operate on press and long-press (import and delete respectively). The keys will be colour coded to provide ease of use.

4 IMPLEMENTATION

The implementation phase can be divided into four categories. These are the layout (xml), the activities (kt), the custom classes (kt) and supporting files (xml).

Here is a basic overview of each category and a short description of the files the contain :

Layout

- **activity_main**: Adds two buttons using *ConstraintLayout*
- **activity_calculator**: Adds a texbox with *LinearLayout* and numerous buttons for 0-9 and most basic mathematical operations using *GridLayout*
- **activity_database**: Uses *ConstraintLayout* to display a list

Activities

- **MainActivity**: Allows navigation to the other activities using the two buttons
- **CalculatorActivity**: Uses the various buttons to perform calculations in the textbox
- **DatabaseActivity**: Displays the SQLite database using the list, use the press to load data from database into calculator activity and longpress to erase data from database

Custom Classes

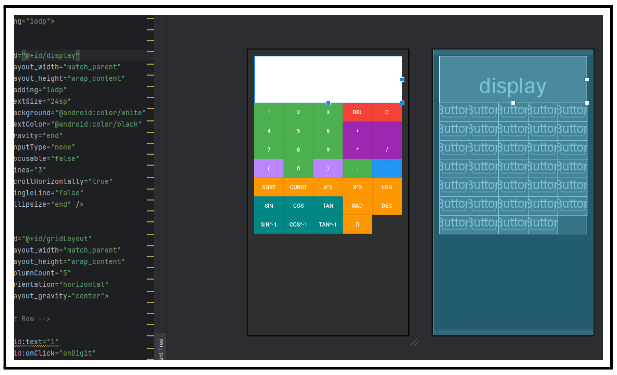
- **DatabaseHandler**: Allows the performance of CRUD operations and conversion of SQlite into a list for *DatabaseActivity*
- **ResultAdapter**: Was used to convert data types earlier in development but is now depreciated

Supporting Files

- **Colors**: Used to define colors for use throughout the project
- **Strings**: Hardcoded string data
- **Themes**: Controls the theme of the application, such as background colour

4.1 LAYOUT

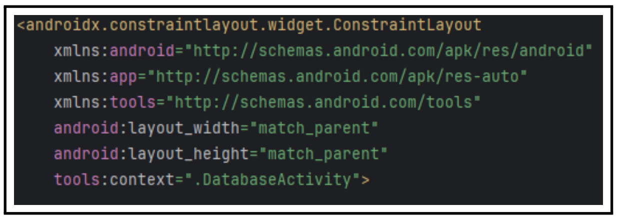
The GUI of the application is largely handled through the layout xml files. This section will not be an exhaustive review of the full xml files for the GUI for brevity. However all the different implementations will be reviewed. That is to say each unique type of entry will be discussed but not every instance of said entry. The layout files can be describes a combination of two concepts: layout style and GUI elements.



```
ng="16dp">

d="@+id/display"
ayout_width="match_parent"
ayout_height="wrap_content"
adding="16dp"
extSize="24sp"
ackground="@android:color/white"
extColor="@android:color/black"
ravity="end"
nputType="none"
ocusable="false"
ines="3"
crollHorizontally="true"
ingleLine="false"
llipsize="end" />

d="@+id/gridLayout"
ayout_width="match_parent"
ayout_height="wrap_content"
olumnCount="5"
rientation="horizontal"
ayout_gravity="center">

t Row -->

id:text="1"
id:onClick="onDigit"
```

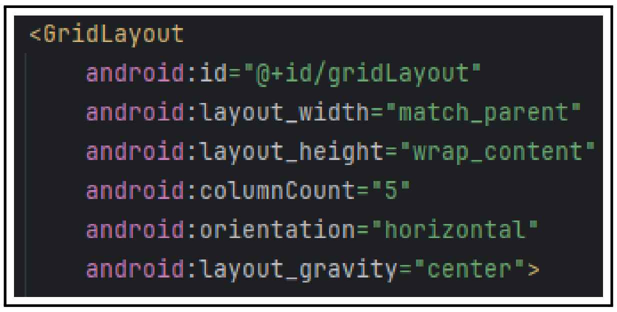
Figure 5: Text/GUI editor side by side

There are a few Layout style such as *GridLayout* or *LinearLayout*, even a combination of various layout styles. These control how the canvas is divided, using a cell like structure. How the 'cells' are arranged, their size, orientation, and borders are all defined by the layout style and its attributes.



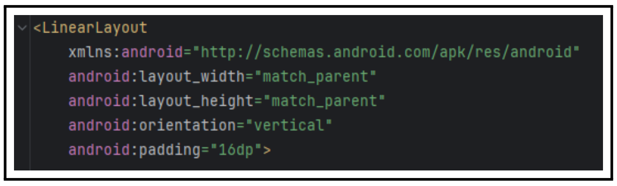
```
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".DatabaseActivity">
```

Figure 6: Example of constraint layout



```
<GridLayout
    android:id="@+id/gridLayout"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:columnCount="5"
    android:orientation="horizontal"
    android:layout_gravity="center">
```

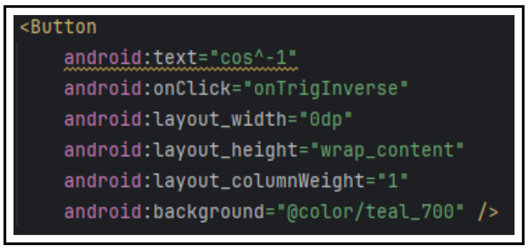
Figure 7: Example of grid layout



```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">
```

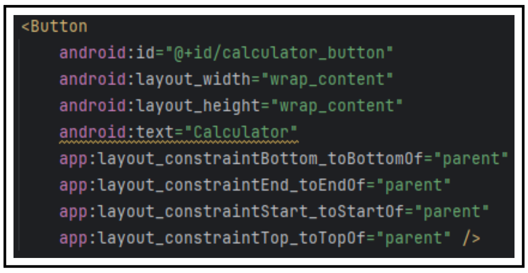
Figure 8: Example of linear layout

GUI Elements can then be placed into these cells, essentially controlling their placement on the canvas. It is important to note that these elements will be placed into the canvas in the order that they appear in the layout file. Depending on the layout style used various attributes will become available or invalid so it is important to make sure that the xml file is purposefully crafted. In android studio it is also possible to use a GUI to generate this xml file and add/remove elements.



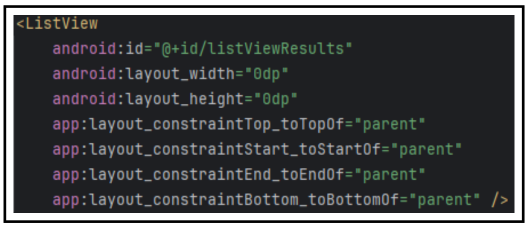
```xml
<Button
    android:text="cos^-1"
    android:onClick="onTrigInverse"
    android:layout_width="0dp"
    android:layout_height="wrap_content"
    android:layout_columnWeight="1"
    android:background="@color/teal_700" />
```

Figure 9: Example of button element in grid layout



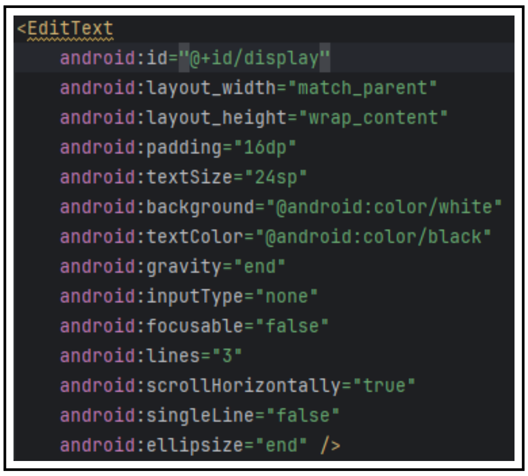
```xml
<Button
    android:id="@+id/calculator_button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Calculator"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />
```

Figure 10: Example of button element in constraint layout



```xml
<ListView
    android:id="@+id/listViewResults"
    android:layout_width="0dp"
    android:layout_height="0dp"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintBottom_toBottomOf="parent" />
```

Figure 11: Example of list element



```xml
<EditText
    android:id="@+id/display"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:padding="16dp"
    android:textSize="24sp"
    android:background="@android:color/white"
    android:textColor="@android:color/black"
    android:gravity="end"
    android:inputType="none"
    android:focusable="false"
    android:lines="3"
    android:scrollHorizontally="true"
    android:singleLine="false"
    android:ellipsize="end" />
```

Figure 12: Example of modifiable text-field element

activity_main

The *activity_main* layout xml file is the first one that shall be discussed. This file utilises the constraint layout style, it only contains two element. That is two buttons, labelled calculator and database, respectively.

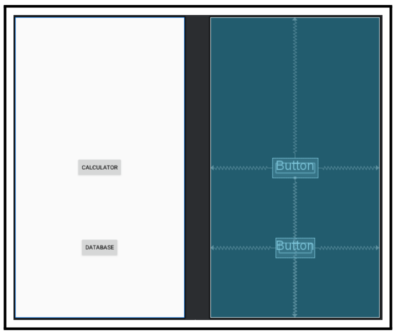


Figure 13: *activity_main.xml* preview

activity_ calculator

The *activity_database* layout xml file also uses the constraint layout style, and only contains the list element.

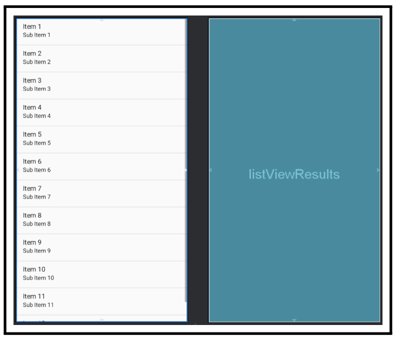


Figure 14: *activity_database.xml* preview

activity_calculator

The *activity_calculator* layout xml file also uses a combination of linear layout and grid layout styles, in that order. The first element is a modifiable text field, within the linear layout style. the rest of the elements are buttons, all within the gridlayout style. They are named after various mathematical operations as well as the number 0-9. They are ordered as shown in Figure 15.

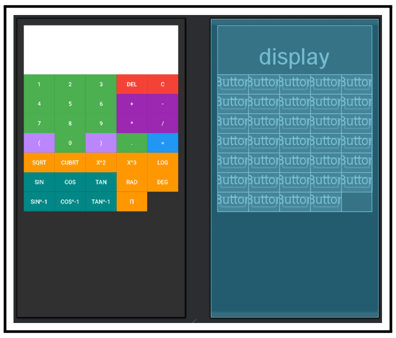


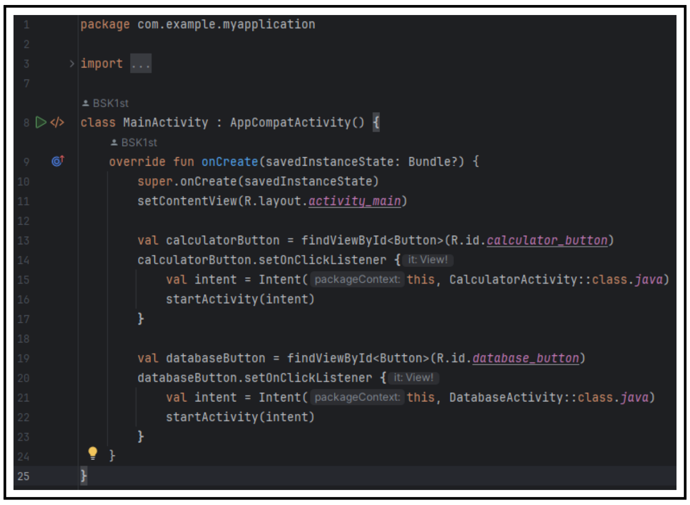
Figure 15: *activity_calculator.xml* preview

4.2 ACTIVITY

The Applications is based on screens, which can be describe programmatically as fragments and activities. Fragments are not used within the program so this section is purely concerned with activities and their implementation. These activities can be programmed using kotlin or java, but in this case java is not used.

MainActivity

The main page is controlled by the *MainActivity* file. This file is fairly short and utilised the *activity_main* file to control its layout. The code controls the result of pressing the buttons defined by the layout file, in this case pressing either button will make the labelled activity active.

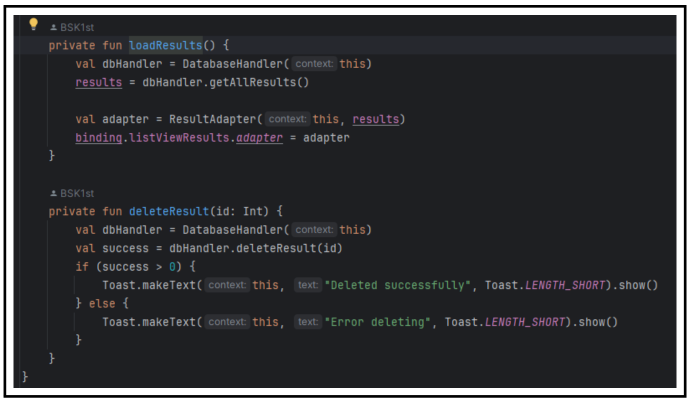


```
package com.example.myapplication

import ...

BSK1st
class MainActivity : AppCompatActivity() {
    BSK1st
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        val calculatorButton = findViewById<Button>(R.id.calculator_button)
        calculatorButton.setOnClickListener { it: View!
            val intent = Intent( packageContext: this, CalculatorActivity::class.java)
            startActivity(intent)
        }

        val databaseButton = findViewById<Button>(R.id.database_button)
        databaseButton.setOnClickListener { it: View!
            val intent = Intent( packageContext: this, DatabaseActivity::class.java)
            startActivity(intent)
        }
    }
}
```

Figure 16: *MainActivity.kt*

DatabaseActivity

The database screen is controlled by the *DatabaseActivity* file. This file utilises the *activity_database* file to control its layout. The list displays the SQLite database via another class (*DatabaseHandler*) that will be discussed later in this report. This activity has two functions that pass the selected list entry to this other class depending on if the list entry was pressed or long-pressed.

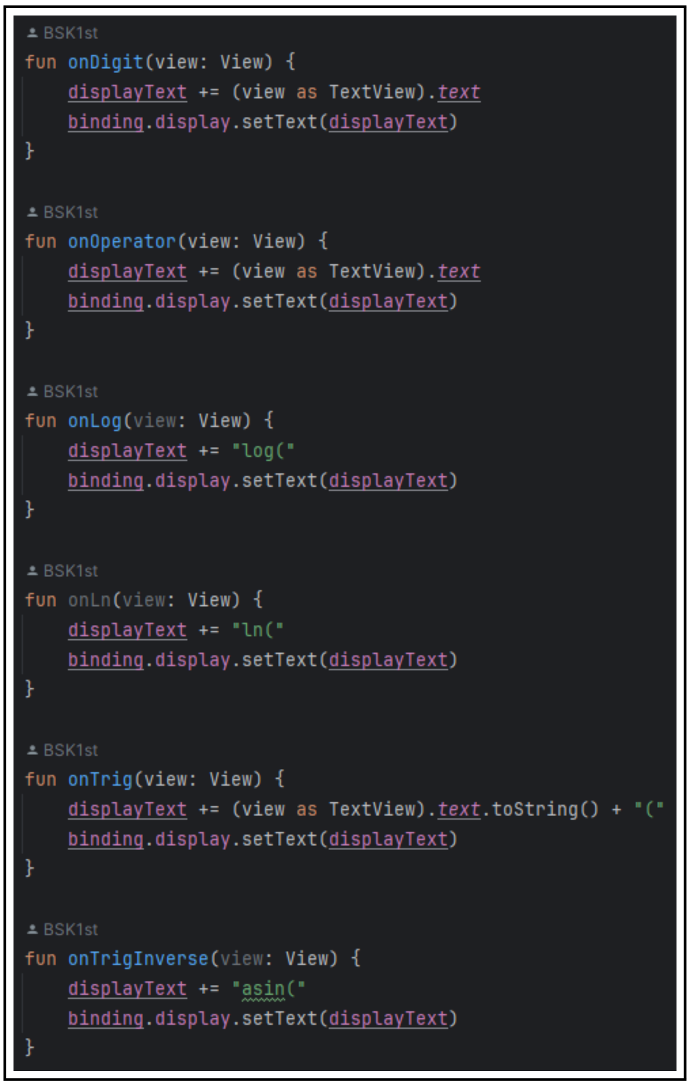


```
    BSK1st
    private fun loadResults() {
        val dbHandler = DatabaseHandler( context: this)
        results = dbHandler.getAllResults()

        val adapter = ResultAdapter( context: this, results)
        binding.listViewResults.adapter = adapter
    }

    BSK1st
    private fun deleteResult(id: Int) {
        val dbHandler = DatabaseHandler( context: this)
        val success = dbHandler.deleteResult(id)
        if (success > 0) {
            Toast.makeText( context: this,  text: "Deleted successfully", Toast.LENGTH_SHORT).show()
        } else {
            Toast.makeText( context: this,  text: "Error deleting", Toast.LENGTH_SHORT).show()
        }
    }
}
```

Figure 17: *DatabaseActivity.kt* snippet

CalculatorActivity

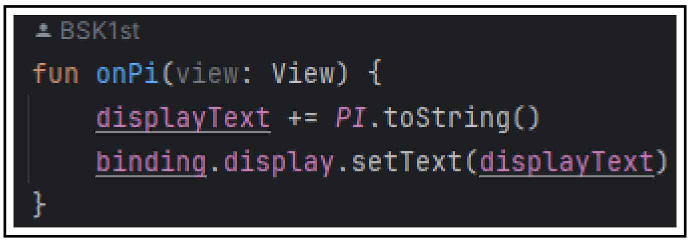
Calculations are performed by the *CalculatorActivity* file. *activity_calculator* is used to set the layout of this activity. Due to the length and the repetitive nature of many of its functions only unique functions will be discussed. What function the button calls when pressed is dictated by android:onClick="Function name.



```
BSK1st
fun onDigit(view: View) {
    displayText += (view as TextView).text
    binding.display.setText(displayText)
}

BSK1st
fun onOperator(view: View) {
    displayText += (view as TextView).text
    binding.display.setText(displayText)
}

BSK1st
fun onLog(view: View) {
    displayText += "log("
    binding.display.setText(displayText)
}

BSK1st
fun onLn(view: View) {
    displayText += "ln("
    binding.display.setText(displayText)
}

BSK1st
fun onTrig(view: View) {
    displayText += (view as TextView).text.toString() + "("
    binding.display.setText(displayText)
}

BSK1st
fun onTrigInverse(view: View) {
    displayText += "asin("
    binding.display.setText(displayText)
}
```

Figure 18: *CalculatorActivity.kt* snippet for functions that control the display

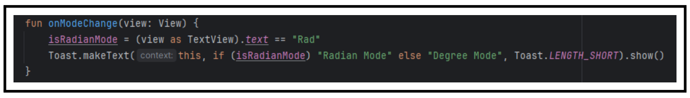
Figure 18 shows various functions that add string data into the text box. This is achieved via two different methods. If the button name is identical to the string, displayText += (view as TextView). text can be used to read the button name and append it into the textbox as a string, otherwise displayText += " string " is used to append the string.



```
BSK1st
fun onPi(view: View) {
    displayText += PI.toString()
    binding.display.setText(displayText)
}
```

Figure 19: *CalculatorActivity.kt* snippet for **onPi()** function

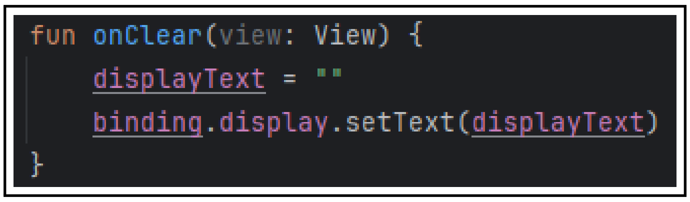
Figure 19 shows a function that adds pi as a string into the text box.



```
fun onModeChange(view: View) {
    isRadianMode = (view as TextView).text == "Rad"
    Toast.makeText( context: this, if (isRadianMode) "Radian Mode" else "Degree Mode", Toast.LENGTH_SHORT).show()
}
```

Figure 20: *CalculatorActivity.kt* snippet for **onModeChange()** function

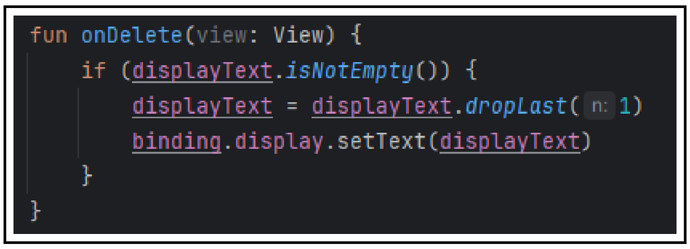
Figure 20 shows a function that toggles between radian and degree calculation methodology depending on its current state.



```
fun onClear(view: View) {
    displayText = ""
    binding.display.setText(displayText)
}
```

Figure 21: *CalculatorActivity.kt* snippet for **onClear()** function

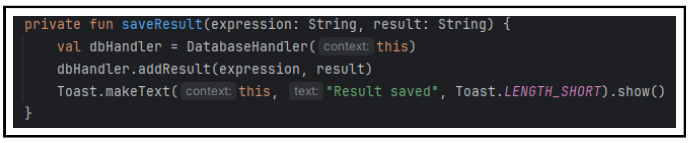
Figure 21 clears the textbox, by setting, instead of appending, it to an empty string.



```
fun onDelete(view: View) {
    if (displayText.isNotEmpty()) {
        displayText = displayText.dropLast( n: 1)
        binding.display.setText(displayText)
    }
}
```

Figure 22: *CalculatorActivity.kt* snippet for **onDelete()** function

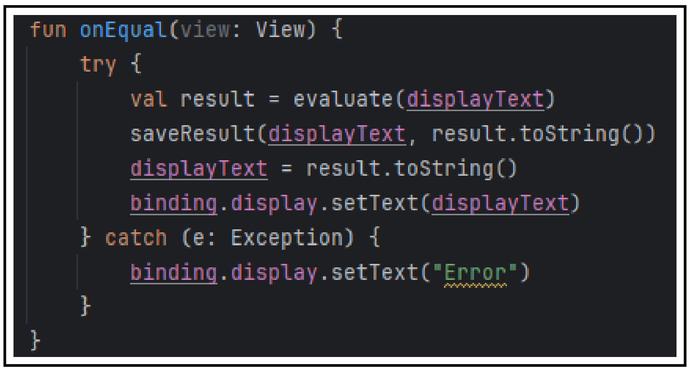
Figure 22 checks if the textbox is not empty and, if its not, erases the last string character in the textbox.



```
private fun saveResult(expression: String, result: String) {
    val dbHandler = DatabaseHandler( context: this)
    dbHandler.addResult(expression, result)
    Toast.makeText( context: this,  text: "Result saved", Toast.LENGTH_SHORT).show()
}
```

Figure 23: *CalculatorActivity.kt* snippet for **saveResult()** function

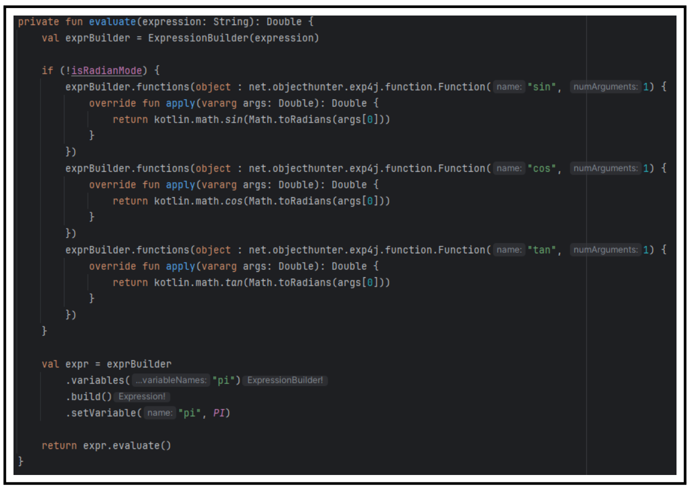
Figure 23 takes two attributes of string data type and passes them to the class *DatabaseHandler*, it also shows a limited lifespan message alerting the user that "Result saved".



```
fun onEqual(view: View) {
    try {
        val result = evaluate(displayText)
        saveResult(displayText, result.toString())
        displayText = result.toString()
        binding.display.setText(displayText)
    } catch (e: Exception) {
        binding.display.setText("Error")
    }
}
```

Figure 24: *CalculatorActivity.kt* snippet for **onEqual** function

Figure 24 takes the string currently in the text box and and passes it to another function, **evaluate()**. The return value is then sent along with the text box contents to the **saveResult()** function, both as string. After that it set the text box to the result string, any invalid expressions return the string "Error" instead.



```
private fun evaluate(expression: String): Double {
    val exprBuilder = ExpressionBuilder(expression)

    if (!isRadianMode) {
        exprBuilder.functions(object : net.objecthunter.exp4j.function.Function( name: "sin",  numArguments: 1) {
            override fun apply(vararg args: Double): Double {
                return kotlin.math.sin(Math.toRadians(args[0]))
            }
        })
        exprBuilder.functions(object : net.objecthunter.exp4j.function.Function( name: "cos",  numArguments: 1) {
            override fun apply(vararg args: Double): Double {
                return kotlin.math.cos(Math.toRadians(args[0]))
            }
        })
        exprBuilder.functions(object : net.objecthunter.exp4j.function.Function( name: "tan",  numArguments: 1) {
            override fun apply(vararg args: Double): Double {
                return kotlin.math.tan(Math.toRadians(args[0]))
            }
        })
    }

    val expr = exprBuilder
        .variables( ...variableNames: "pi") ExpressionBuilder!
        .build() Expression!
        .setVariable( name: "pi", PI)

    return expr.evaluate()
}
```

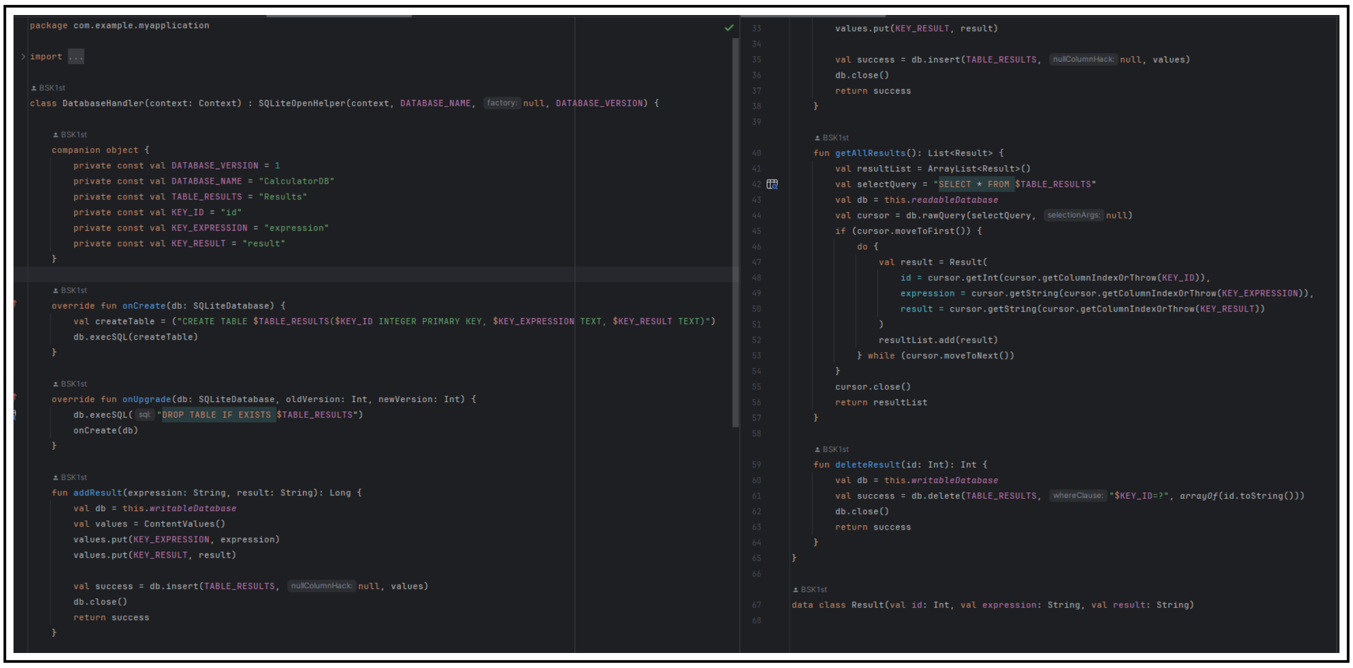
Figure 25: *CalculatorActivity.kt* snippet for **evaluate()** function

Figure 25 uses *exp4j* to evaluate strings as equations and return the value of said equation. This function by default calculates values in radians, which is not an issue unless trig equations calculation functions are used. If the calculator is set to degrees. then the **exprBuilder.functions()** function is used to modify the trig calculation functions by converting the appropriate values into radian equivalents.

4.3 CUSTOM CLASSES

DatabaseHandler

This class is responsible for the CRUD operations in regards to the SQLite database. There are five functions, four of which directly perform the CRUD operations, and one which converts the SQLite database into a list that can be displayed by the *DatabaseActivity* file. It also contains initialisation for SQLite.



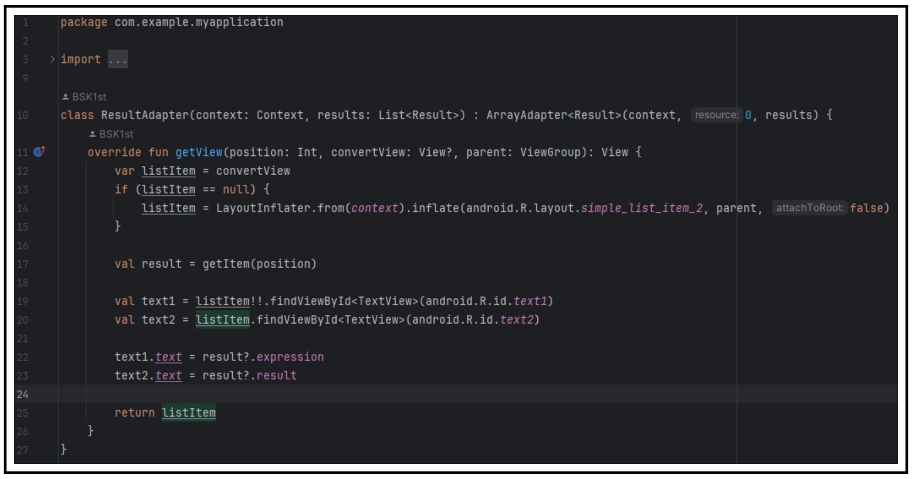
```kotlin
package com.example.myapplication

import ...

BSK1st
class DatabaseHandler(context: Context) : SQLiteOpenHelper(context, DATABASE_NAME, factory: null, DATABASE_VERSION) {

    BSK1st
    companion object {
        private const val DATABASE_VERSION = 1
        private const val DATABASE_NAME = "CalculatorDB"
        private const val TABLE_RESULTS = "Results"
        private const val KEY_ID = "id"
        private const val KEY_EXPRESSION = "expression"
        private const val KEY_RESULT = "result"
    }

    BSK1st
    override fun onCreate(db: SQLiteDatabase) {
        val createTable = ("CREATE TABLE $TABLE_RESULTS($KEY_ID INTEGER PRIMARY KEY, $KEY_EXPRESSION TEXT, $KEY_RESULT TEXT)")
        db.execSQL(createTable)
    }

    BSK1st
    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion: Int) {
        db.execSQL( sql: "DROP TABLE IF EXISTS $TABLE_RESULTS")
        onCreate(db)
    }

    BSK1st
    fun addResult(expression: String, result: String): Long {
        val db = this.writableDatabase
        val values = ContentValues()
        values.put(KEY_EXPRESSION, expression)
        values.put(KEY_RESULT, result)

        val success = db.insert(TABLE_RESULTS, nullColumnHack: null, values)
        db.close()
        return success
    }
```

```kotlin
        values.put(KEY_RESULT, result)

        val success = db.insert(TABLE_RESULTS, nullColumnHack: null, values)
        db.close()
        return success
    }

    BSK1st
    fun getAllResults(): List<Result> {
        val resultList = ArrayList<Result>()
        val selectQuery = "SELECT * FROM $TABLE_RESULTS"
        val db = this.readableDatabase
        val cursor = db.rawQuery(selectQuery, selectionArgs: null)
        if (cursor.moveToFirst()) {
            do {
                val result = Result(
                    id = cursor.getInt(cursor.getColumnIndexOrThrow(KEY_ID)),
                    expression = cursor.getString(cursor.getColumnIndexOrThrow(KEY_EXPRESSION)),
                    result = cursor.getString(cursor.getColumnIndexOrThrow(KEY_RESULT))
                )
                resultList.add(result)
            } while (cursor.moveToNext())
        }
        cursor.close()
        return resultList
    }

    BSK1st
    fun deleteResult(id: Int): Int {
        val db = this.writableDatabase
        val success = db.delete(TABLE_RESULTS, whereClause: "$KEY_ID=?", arrayOf(id.toString()))
        db.close()
        return success
    }
}

BSK1st
data class Result(val id: Int, val expression: String, val result: String)
```

Figure 26: *DatabaseHandler.kt*

ResultAdapter

This calss was initially used to convert the results of calculations into strings to be used in the SQLite database, however the program has change significantly since its introduction. As a result it is depreciated.



```kotlin
package com.example.myapplication

import ...

BSK1st
class ResultAdapter(context: Context, results: List<Result>) : ArrayAdapter<Result>(context, resource: 0, results) {
    BSK1st
    override fun getView(position: Int, convertView: View?, parent: ViewGroup): View {
        var listItem = convertView
        if (listItem == null) {
            listItem = LayoutInflater.from(context).inflate(android.R.layout.simple_list_item_2, parent, attachToRoot: false)
        }

        val result = getItem(position)

        val text1 = listItem!!.findViewById<TextView>(android.R.id.text1)
        val text2 = listItem.findViewById<TextView>(android.R.id.text2)

        text1.text = result?.expression
        text2.text = result?.result

        return listItem
    }
}
```

Figure 27: *ResultAdapter.kt*

5 TESTING

Testing was performed by building the program and installing it into a physical android device. To set up this process the android device needed to be prepared. The process is as follows:

1. Navigate to the device setting
2. Under *About Phone* tap *Build number* 5 times. This will enable developer mode
3. In developer options, search for *wireless debugging* or *usb debugging*
4. go to android studio go to *Running Devices* tab and follow the instructions to connect your phone via the appropriate media
5. The phone should appear under the *Running Devices* tab

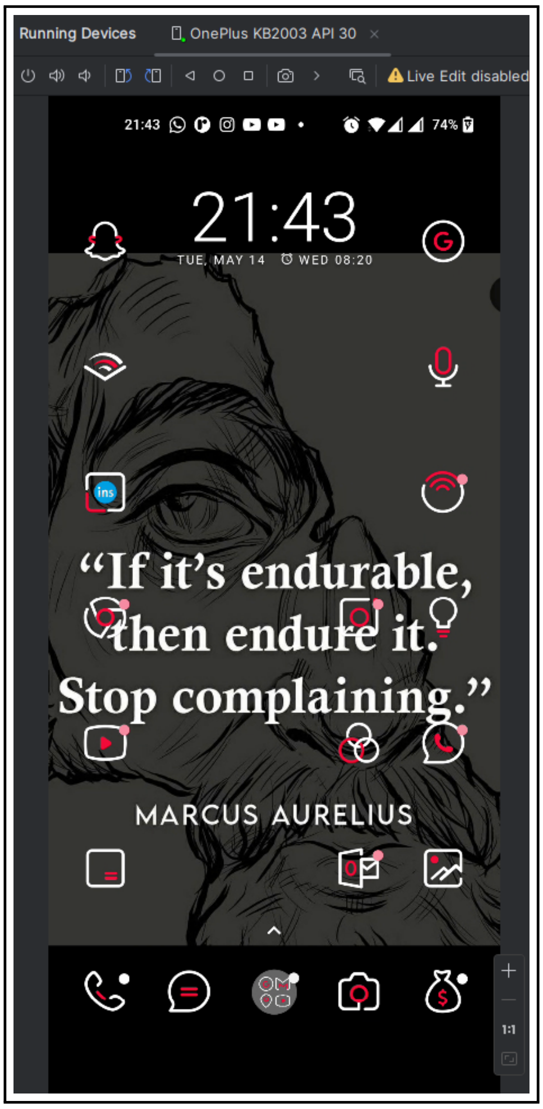
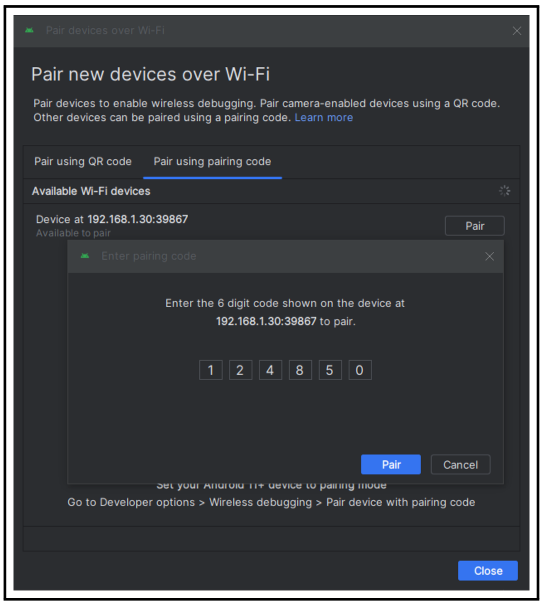


Figure 28: Paired device



Pair devices over Wi-Fi

Pair new devices over Wi-Fi

Pair devices to enable wireless debugging. Pair camera-enabled devices using a QR code. Other devices can be paired using a pairing code. Learn more

Pair using QR code | Pair using pairing code

Available Wi-Fi devices

Device at 192.168.1.30:39867
Available to pair

Enter pairing code

Enter the 6 digit code shown on the device at 192.168.1.30:39867 to pair.

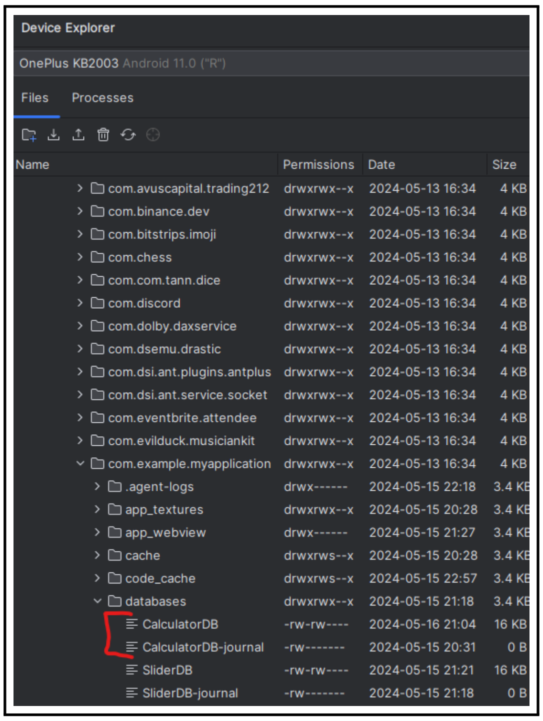
1 2 4 8 5 0

Set your Android 11+ device to pairing mode
Go to Developer options > Wireless debugging > Pair device with pairing code

Figure 29: Wireless pairing method

Once the device is connected, the device is capable of running the application through android studio on the PC. The device can even be controlled from the PC completely. Note that running the program on the device will install the app on the device, this can cause if the database on the device has different characteristic from the newly uploaded code. This is an error that can be remedied by uninstalling the application and reuploading the code.

The database of the program is persistent as it is save in the device, the application can also be used on the device without a connection once it is install, assuming functionality.



Device Explorer

OnePlus KB2003 Android 11.0 ("R")

Name	Permissions	Date	Size
com.avuscapital.trading212	drwxrwx--x	2024-05-13 16:34	4 KB
com.binance.dev	drwxrwx--x	2024-05-13 16:34	4 KB
com.bitstrips.imoji	drwxrwx--x	2024-05-13 16:34	4 KB
com.chess	drwxrwx--x	2024-05-13 16:34	4 KB
com.com.tann.dice	drwxrwx--x	2024-05-13 16:34	4 KB
com.discord	drwxrwx--x	2024-05-13 16:34	4 KB
com.dolby.daxservice	drwxrwx--x	2024-05-13 16:34	4 KB
com.dsemu.drastic	drwxrwx--x	2024-05-13 16:34	4 KB
com.dsi.ant.plugins.antplus	drwxrwx--x	2024-05-13 16:34	4 KB
com.dsi.ant.service.socket	drwxrwx--x	2024-05-13 16:34	4 KB
com.eventbrite.attendee	drwxrwx--x	2024-05-13 16:34	4 KB
com.evilduck.musiciankit	drwxrwx--x	2024-05-13 16:34	4 KB
com.example.myapplication	drwxrwx--x	2024-05-13 16:34	4 KB
.agent-logs	drwx------	2024-05-15 22:18	3.4 KB
app_textures	drwxrwx--x	2024-05-15 20:28	3.4 KB
app_webview	drwx------	2024-05-15 21:27	3.4 KB
cache	drwxrws--x	2024-05-15 20:28	3.4 KB
code_cache	drwxrws--x	2024-05-15 22:57	3.4 KB
databases	drwxrwx--x	2024-05-15 21:18	3.4 KB
CalculatorDB	-rw-rw----	2024-05-16 21:04	16 KB
CalculatorDB-journal	-rw-------	2024-05-15 20:31	0 B
SliderDB	-rw-rw----	2024-05-15 21:21	16 KB
SliderDB-journal	-rw-------	2024-05-15 21:18	0 B

Figure 30: File location on device that contains database

This code was tested by other users as well on their personal smart phones, Callium Dawes being one of the primary testers.

There is video evidence of this programs functionality on
There is video evidence of this programs functionality on
There is video evidence of this programs functionality on
`https://youtu.be/K1ncPa6lHOk`

6 FUTURE DEVELOPMENT

Future development of the application could include the integration of additional features such as enhanced security measures, more advanced user analytics, and expanded functionality such as a latex math based display for advanced equations. Also certain modes like SQRT and Pi can be streamlined (Pi already has the necessary code in *evaluate()* but it hasn't been implemented but it can be done by changing the name of the button from π to "pi"). The code has many little features that could be optimised to improve the user experience.

7 CONCLUSION

In conclusion, the development of this Android application has been a comprehensive exercise in applying theoretical knowledge to practical implementation. The project has successfully met the specified requirements and provided a functional, user-friendly application. Continuous improvements and additional features will further enhance the application in future development cycles.