EditText in Android

EditText is an editable version of TextView. It is ideal for user input fields and supports both single-line and multi-line text input.

```
<EditText
  android:id="@+id/editText"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:hint="Enter your name"/>

```

RadioButton and CheckBox in Android

- **RadioButton**: Used for single selection from multiple options.
- **CheckBox**: Allows multiple selections among a set of options.

```
<CheckBox
  android:id="@+id/checkBox"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:text="Agree to terms"/>

<RadioGroup
  android:layout_width="wrap_content"
  android:layout_height="wrap_content">
  <RadioButton
    android:id="@+id/radioButton1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Option 1"/>
  <RadioButton
    android:id="@+id/radioButton2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Option 2"/>
</RadioGroup>
```

Button in Android

Button triggers an action when clicked. It shares properties with TextView but includes unique attributes for user interaction.

```
<Button
   android:id="@+id/button"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Click Me"/>
```

ImageView in Android

ImageView displays images in the user interface. To use an image, add it to the **drawable** folder.

```
<ImageView
   android:id="@+id/img"
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:scaleType="fitCenter"
   android:src="@drawable/img_nature"/>
```

Common Attributes

- android:src: Specifies the image resource.
- android:scaleType: Determines how the image fits into the view.
- android:background: Adds a background color or drawable.
- android:padding: Adds space around the image.

ScaleType Options

- CENTER: Centers the image without scaling.
- CENTER_CROP: Scales the image uniformly to fill the view.
- CENTER_INSIDE: Scales the image to fit inside the container.
- FIT_XY: Stretches the image to fill the entire view (may distort).
- MATRIX: Applies custom transformations.

ImageButton in Android

ImageButton is similar to ImageView but acts as a clickable button.

```
<ImageButton
  android:id="@+id/imgButton"
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:scaleType="fitCenter"
  android:src="@drawable/img_nature"/>
```

Connecting the Views with the Business Logic

```
MainActivity.kt

class MainActivity : AppCompatActivity() {

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

    val textView = findViewById<TextView>(R.id.textView)
    val editText = findViewById<EditText>(R.id.editText)
    val imageView = findViewById<ImageView>(R.id.imageView)
    val imageButton = findViewById<ImageButton>(R.id.imageButton)
    val checkBox = findViewById<CheckBox>(R.id.checkBox)
    val radioButtonM = findViewById<RadioButton>(R.id.radioButtonMale)
    val radioButtonF = findViewById<RadioButton>(R.id.radioButtonFemale)
    val button = findViewById<Button>(R.id.button)
    button.setOnClickListener {
      val name = editText.text.toString()
      val gender = when {
        radioButtonMale.isChecked -> "Male"
        radioButtonFemale.isChecked -> "Female"
        else -> "Unknown"
      }
      val subscribed = if (checkBox.isChecked) "Subscribed" else "Not
Subscribed"
      textView.text = "Name: $name\nGender: $gender\nSubscription:
$subscribed"
    }
    imageButton.setOnClickListener {
      imageView.setImageResource(R.drawable.another_image)
    }
  }
}
```

ListView in Android

A **ListView** is a versatile UI component in Android that allows you to display items in a vertically scrollable list. It is commonly used for dynamic datasets like contact lists, emails, or search results, where users can easily browse and interact with the data by scrolling up or down.

Key Features of ListView:

1. **Dynamic Data Display**: Suitable for presenting large datasets.
2. **Customizable Items**: Supports the use of **TextView**, **ImageView**, or a combination of views in each list item.

3. **Adapts Data Automatically**: Uses adapter classes like ArrayAdapter or CustomAdapter to bind data dynamically to the view.
4. **Dividers for Aesthetic Design**: Separates list items using dividers, with customizable height and colour.

Common Attributes of ListView

Attribute	**Description**
android:divider	Specifies a divider between items. A drawable or colour can be used.
android:dividerHeight	Sets the height of the divider between items.

```
<ListView
  android:id="@+id/listView"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:divider="@android:color/black"
  android:dividerHeight="1dp"/>
```

What is an Adapter?

An **Adapter** acts as a bridge between a data source and the UI components. It converts the data into individual **View** objects and passes them to the corresponding Adapter View to display.

Key Features of an Adapter:

- Reads data from data sources such as arrays, lists, or databases.
- Converts raw data into UI elements like TextView, ImageView, etc.
- Supplies the views to the Adapter View dynamically.

Common Adapter Types in Android

1. **ArrayAdapter**
   - Used to display a list of strings or simple objects.
   - Example: Displaying a list of names or items.
2. **SimpleAdapter**
   - Ideal for mapping data from a list of maps to views.
   - Example: Populating a list with complex data like images and text.
3. **BaseAdapter**
   - A flexible and customizable adapter that can be extended to create custom adapters for unique use cases.

```
val items = listOf("Apple", "Banana", "Cherry")
val adapter = ArrayAdapter(this, android.R.layout.simple_list_item_1, items)
listView.adapter = adapter
```

What is an Adapter View?

An **Adapter View** is a UI component in Android that relies on an Adapter to provide the data it displays. It does not directly manage data but instead uses the Adapter to supply it. Examples of Adapter Views include **ListView**, **GridView**, and **Spinner**.

Key Features of an Adapter View:

- Displays large datasets efficiently by reusing views for visible items (view recycling).

- Dynamically loads and unloads data as the user scrolls.
- Works seamlessly with Adapters to present data in different layouts

How Adapter and Adapter View Work Together

1. The **Adapter** fetches data from the source and converts it into views.
2. The **Adapter View** requests these views from the Adapter to display them.
3. For large datasets, only the views for visible items are created, and the rest are loaded as needed.

Example of Adapter View with ListView:

```
import android.os.Bundle
import android.widget.ArrayAdapter
import android.widget.ListView
import androidx.appcompat.app.AppCompatActivity

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

      val listView: ListView = findViewById(R.id.listView)
      val items = listOf("Item 1", "Item 2", "Item 3", "Item 4")

      val adapter = ArrayAdapter(this, android.R.layout.simple_list_item_1,
items)
      listView.adapter = adapter
   }
}
```

By using Adapters and Adapter Views, Android ensures that applications can handle and display large datasets efficiently without compromising performance.

RecyclerView in Android

RecyclerView is a flexible and efficient component for creating dynamic lists or grids in Android applications. It is particularly well-suited for handling large datasets and supports heterogeneous layouts, making it a popular choice for applications such as news feeds or navigation drawers.

Features of RecyclerView

- **Supports Heterogeneous Layouts:** Different types of views (e.g., text, images, videos) can coexist within a single RecyclerView.
- **Efficient View Recycling:** Views that scroll off-screen are reused, reducing memory usage and improving performance.
- **Customizable Animations:** Allows for smooth animations during item addition, removal, or updates.

Steps to Implement RecyclerView

1. Plan Your Layout

➢ Decide on the structure (e.g., list or grid).

➢ Use a suitable LayoutManager:

- LinearLayoutManager for vertical or horizontal lists.
- GridLayoutManager for grids.
- StaggeredGridLayoutManager for staggered grids.

➢ Design the layout for each list item in an XML file.

2. Create Adapter and ViewHolder

- **Adapter:** Manages data and binds it to the ViewHolder. Extend RecyclerView.Adapter.
- **ViewHolder:** Caches references to views within a layout for efficient reuse. Extend RecyclerView.ViewHolder.
- Override key methods:
    - onCreateViewHolder(): Inflates the layout for each item.
    - onBindViewHolder(): Binds data to views.
    - getItemCount(): Returns the size of the dataset.

3. Set Up RecyclerView in the Activity

- Initialize the RecyclerView, set its LayoutManager, and attach the custom Adapter.

```
class CustomAdapter(private val dataset: List<String>) :
RecyclerView.Adapter<CustomAdapter.ViewHolder>() {

  class ViewHolder(itemView: View) :
RecyclerView.ViewHolder(itemView) {
    val textView: TextView = itemView.findViewById(R.id.item_text)
  }

  override fun onCreateViewHolder(parent: ViewGroup, viewType: Int):
ViewHolder {
    val view = LayoutInflater.from(parent.context)
      .inflate(R.layout.item_view, parent, false)
    return ViewHolder(view)
  }

  override fun onBindViewHolder(holder: ViewHolder, position: Int) {
    holder.textView.text = dataset[position]
  }

override fun getItemCount(): Int = dataset.size
}
```

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

    val dataset = listOf("January", "February", "March")
    val customAdapter = CustomAdapter(dataset)
    val recyclerView: RecyclerView = findViewById(R.id.recycler_view)

    recyclerView.layoutManager = LinearLayoutManager(this)
    recyclerView.adapter = customAdapter
  }
}
```

Customizing RecyclerView

1. Animations: Add animations for item addition, removal, or updates.
2. Dividers: Add dividers between items using DividerItemDecoration.
3. Heterogeneous Layouts: Implement getItemViewType() in your Adapter to handle different view types.

Picker Views in Android

Picker views are pre-built dialogs in Android that allow users to select a date or time. These pickers ensure users input valid, locale-specific, and correctly formatted date or time values.

Types of Picker Views

1. **DatePicker**

   - Allows users to select a date (year, month, day).
   - Can be used as a standalone widget in a layout or as part of a DatePickerDialog.

```
<DatePicker
  android:id="@+id/datePicker"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:datePickerMode="spinner" />
```

2. **TimePicker**

   - Allows users to select time (hours and minutes).
   - Like DatePicker, it can also be used as a widget or within a TimePickerDialog.

```
<TimePicker
  android:id="@+id/timePicker"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:timePickerMode="spinner"
  android:is24HourView="true" />
```

Combining DatePickerDialog and TimePickerDialog

```
class MainActivity : AppCompatActivity() {

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

    val selectDateTimeButton: Button =
findViewById(R.id.select_datetime_btn)
    val dateTimeText: TextView = findViewById(R.id.datetime_text)

    selectDateTimeButton.setOnClickListener {
      val calendar = Calendar.getInstance()
      val year = calendar.get(Calendar.YEAR)
      val month = calendar.get(Calendar.MONTH)
      val day = calendar.get(Calendar.DAY_OF_MONTH)
      val hour = calendar.get(Calendar.HOUR_OF_DAY)
      val minute = calendar.get(Calendar.MINUTE)

      // Show DatePickerDialog first
      val datePickerDialog = DatePickerDialog(this, { _, selectedYear,
selectedMonth, selectedDay ->

        // Once date is selected, show TimePickerDialog
        val timePickerDialog = TimePickerDialog(this, { _, selectedHour,
selectedMinute ->
          val selectedDateTime = "$selectedDay/${selectedMonth +
1}/$selectedYear $selectedHour:$selectedMinute"
          dateTimeText.text = selectedDateTime
        }, hour, minute, true)

        timePickerDialog.show()
      }, year, month, day)

      datePickerDialog.show()
    }
  }
}
```

Android WebView

Android WebView is a component that allows you to display web pages inside an Android application. It provides developers with greater control over the UI and configuration when integrating web content.

Step 1: Add Internet Permission

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
   package="com.example.webview">

<!-- Internet permission -->
<uses-permission android:name="android.permission.INTERNET"/>

```

Step 2: Add WebView to the Layout File

Add a WebView element to the layout file activity_main.xml to load and display web pages.

```
<WebView
   android:id="@+id/myWebView"
   android:layout_width="match_parent"
   android:layout_height="match_parent" />

```

Step 4: Configure the WebView in MainActivity

```
class MainActivity : AppCompatActivity() {
   private lateinit var myWebView: WebView

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

      // Initialize WebView
      myWebView = findViewById(R.id.myWebView)
      // Load the desired URL
      myWebView.loadUrl("https://www.mbu.com/")
   }
}
```

ANDROID DIFFERENT TYPES OF MENUS

In android, we have a three fundamental type of Menus available to define a set of options and actions in our android applications.

The following are the commonly used Menus in android applications.

- Options Menu
- Context Menu
- Popup Menu

For all menu types, Android provides a standard XML format to define menu items. Instead of building a menu in our activity's code, we should define a menu and all its items in an XML menu resource and load menu resource as a Menu object in our activity or fragment.

In android, to define menu, we need to create a new folder **menu** inside of our project resource directory (**res/menu/**) and add a new XML file to build the menu with the following elements.

Element	Description
<menu>	It's a root element to define a Menu in XML file and it will hold one or more and elements.
<item>	It is used to create a menu item and it represents a single item on the menu. This element may contain a nested <menu> element in order to create a submenu.
<group>	It's an optional and invisible for <item> elements. It is used to categorize the menu items so they share properties such as active state and visibility.

Once we are done with creation of menu, we need to load the menu resource from our activity using **MenuInflater.inflate()** like as shown below.

```
override fun onCreateOptionsMenu(menu: Menu?): Boolean {
   menuInflater.inflate(R.menu.menu_options, menu)
   return true
}
```

Android Options Menu

In android, **Options Menu** is a primary collection of menu items for an activity and it is useful to implement actions that have a global impact on the app, such as Settings, Search, etc.

Android Context Menu

In android, **Context Menu** is a floating menu that appears when the user performs a long click on an element and it is useful to implement actions that affect the selected content or context frame.

Android Popup Menu

In android, **Popup Menu** displays a list of items in a vertical list that's anchored to the view that invoked the menu and it's useful for providing an overflow of actions that related to specific content.

SHARED PREFERENCES

Shared Preferences is a built-in Android data storage mechanism designed to save and retrieve small amounts of private data in the form of key-value pairs. Shared Preferences is commonly used for scenarios like saving user preferences, application settings, or any other lightweight data that needs to persist between application launches.

Key Features of Shared Preferences

1. Stores data in an XML file located at: data/data/<package-name>/shared-prefs/<filename>.xml
2. Only primitive data types can be saved:
    - boolean, float, int, long, string, and stringSet.

Accessing SharedPreferences Object

1. getSharedPreferences(String name, int mode)

- Used for multiple preferences files.

```
val sharedPreferences = getSharedPreferences("myPreferences",
Context.MODE_PRIVATE)
```

Saving Data

```
val sharedPreferences = getSharedPreferences("myPreferences",
Context.MODE_PRIVATE)
val editor = sharedPreferences.edit()

editor.putString("username", "JohnDoe")
editor.putInt("age", 25)
editor.putBoolean("isLoggedIn", true)
editor.apply() // or editor.commit()
```

Retrieving Data

```
val sharedPreferences = getSharedPreferences("myPreferences",
Context.MODE_PRIVATE)

val username = sharedPreferences.getString("username", "DefaultUser")
val age = sharedPreferences.getInt("age", 0)
val isLoggedIn = sharedPreferences.getBoolean("isLoggedIn", false)

println("Username: $username, Age: $age, Is Logged In: $isLoggedIn")

```

Removing or Clearing Data

```
val sharedPreferences = getSharedPreferences("myPreferences",
Context.MODE_PRIVATE)
    val editor = sharedPreferences.edit()
    editor.remove("username").apply() //Removing Specific Key

// Clear all data
    editor.clear().apply()
```

Android Persistence with Preferences and Files

Persistence in Android refers to saving data locally on the device to ensure it is retained even when the app is closed or the device is restarted. Android provides multiple mechanisms for data persistence, depending on the nature and scope of the data.

File-Based Persistence in Android

Android creates a private storage directory for each application at the following location: /data/data/[application_package]/.

This directory contains three main subdirectories

1. **Files:** Stores general application data.
2. **Cache**: Stores temporary data (cleared when space is needed).
3. **Shared Preferences**: Stores key-value pairs for app preferences.

Options for Storing Data in Files

- **Files**: You can create, read, and update files directly.
- **Preferences**: Use Shared Preferences to store and retrieve lightweight, key-value-based data.
- **SQLite Database**: Use SQLite databases for structured and relational data storage.

Internal vs. External Storage

Internal Storage

- Private to the application.
- Data is saved in the application's directory (/data/data/[application_package]/files).
- Only accessible by the application (unless explicitly shared via a FileProvider).
- Ideal for storing sensitive data.
- Automatically cleared when the application is uninstalled.

```
   val fileName = "example.txt"
    val fileContent = "This is an internal file."
// Writing to internal storage
    openFileOutput(fileName, Context.MODE_PRIVATE).use {
      it.write(fileContent.toByteArray())
    }
// Reading from internal storage
    val fileContentRead = openFileInput(fileName).bufferedReader().useLines {
lines ->
      lines.joinToString("\n")
    }
    println("Read from file: $fileContentRead")
```

External Storage

- Publicly accessible.
- Not always available (e.g., when mounted via USB or missing an SD card).
- Requires runtime permission for reading and writing from Android 6.0 (API level 23) onward.

```
if (Environment.getExternalStorageState() ==
Environment.MEDIA_MOUNTED) {
   val externalDir = getExternalFilesDir(null) // App-specific external directory
   val file = File(externalDir, "example.txt")

   // Writing to external storage
   file.writeText("This is an external file.")

   // Reading from external storage
   val content = file.readText()
   println("Read from file: $content")
}
```

Creating and Using Databases Using SQLite in Android

SQLite is a lightweight, embedded SQL database engine designed for local data storage in mobile and desktop applications. It is self-contained, serverless, and requires zero configuration, making it ideal for Android applications. SQLite reads and writes directly to disk files, and all database objects (tables, indexes, etc.) are stored in a single file.

Key Components of SQLite in Android

1. **SQLiteOpenHelper**
   - A helper class to manage database creation, connection, and version management.
   - It abstracts away the complexity of handling raw SQL commands to create or update databases.
   - Implements the onCreate() method to define the schema and the onUpgrade() method to handle database version changes.
2. **SQLiteDatabase**
   - Represents the database instance.

- Provides methods like insert(), update(), delete(), and query() for performing database operations.
- Accessed via SQLiteOpenHelper using:
    - getWritableDatabase(): For write operations.
    - getReadableDatabase(): For read-only operations.

3. **Cursor**
    - A class that provides access to the results of database queries.
    - Acts as an iterator to navigate through query results row by row.
    - Optimized for handling large datasets by loading data in batches.

Steps to Use SQLite in Android

1. **Define a Database Schema**
    - Identify the tables, columns, and relationships required for your application.

```
CREATE TABLE Users (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  name TEXT NOT NULL,
  email TEXT UNIQUE NOT NULL
);
```

2. **Create a Database Helper Class**

- Extend SQLiteOpenHelper and override onCreate() and onUpgrade().

```

class MyDatabaseHelper(context: Context) : SQLiteOpenHelper(context,
"MyDatabase.db", null, 1) {
  override fun onCreate(db: SQLiteDatabase) {
    // Create tables
```

```
        db.execSQL(
            "CREATE TABLE Users (id INTEGER PRIMARY KEY
AUTOINCREMENT, name TEXT NOT NULL, email TEXT UNIQUE NOT
NULL)"
        )
    }

    override fun onUpgrade(db: SQLiteDatabase, oldVersion: Int, newVersion:
Int) {
        // Handle database upgrades
        db.execSQL("DROP TABLE IF EXISTS Users")
        onCreate(db)
    }
}
```

3. Open the Database

- Use getWritableDatabase() or getReadableDatabase() to interact with the database.

```
val dbHelper = MyDatabaseHelper(context)
val db = dbHelper.writableDatabase
```

4. Perform Database Operations

Insert Data

```
val values = ContentValues()
values.put("name", "Rithickshan")
values.put("email", "rithickshan@gmail.com")
val newRowId = db.insert("Users", null, values)
```

Read-Query Data

```
val cursor = db.query(
   "Users",
   arrayOf("id", "name", "email"), // Columns to return
   null, // Selection criteria
   null, // Selection arguments
   null, // Group by
   null, // Having
   null  // Order by
)

while (cursor.moveToNext()) {
   val userId = cursor.getInt(cursor.getColumnIndexOrThrow("id"))
   val userName = cursor.getString(cursor.getColumnIndexOrThrow("name"))
   val userEmail = cursor.getString(cursor.getColumnIndexOrThrow("email"))
   println("ID: $userId, Name: $userName, Email: $userEmail")
}
cursor.close()
```

Update Data

```
val updatedValues = ContentValues()
updatedValues.put("name", "Rithick")
val rowsUpdated = db.update(
   "Users",
   updatedValues,
   "email = ?",
   arrayOf("rithickshan@gmail.com")
)
```

Delete Data

```
val rowsDeleted = db.delete("Users", "email = ?",
arrayOf("rithickshan@gmail.com"))
```

5. **Close the Database**

- Always close the database to free up resources.

```
db.close()
```

Advantages of SQLite

- Lightweight and serverless.
- Fast and efficient for small to medium-sized datasets.
- Full-featured SQL implementation.
