**Section 18: Volume Calculator App - Mastering Adapters & GridView**

**Key Concepts Taught**

1. **GridView**: UI widget for displaying items in a 2D grid.
2. **Custom Adapters**: Bridge between data source and GridView.
3. **Model Classes**: Structure data for adapter items.
4. **ViewHolder Pattern**: Optimizes view recycling.
5. **Layout Inflation**: Dynamically creating UI elements.
6. **Intents**: Navigate between activities.
7. **Volume Calculation**: Math operations for shapes.

**Implementation Steps**

**1. Project Setup**

* **Step 1**: Create a new Android Studio project (Empty Views Activity).
* **Step 2**: Name the project "Volume Area App".

**2. Main Layout (**activity\_main.xml**)**

xml

<GridView

android:id="@+id/grid"

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:numColumns="2" /> *<!-- Set columns programmatically or in XML -->*

* **Tools**: Android Studio Layout Editor.
* **Best Practice**: Use numColumns to define grid structure.

**3. Grid Item Layout (**grid\_item\_layout.xml**)**

xml

<androidx.constraintlayout.widget.ConstraintLayout

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content">

<ImageView

android:id="@+id/imageView"

android:layout\_width="120dp"

android:layout\_height="120dp"

android:paddingStart="16dp"

android:paddingTop="16dp" />

<TextView

android:id="@+id/textView"

android:layout\_width="wrap\_content"

android:layout\_height="wrap\_content"

app:layout\_constraintTop\_toBottomOf="@id/imageView"

app:layout\_constraintStart\_toStartOf="@id/imageView" />

</androidx.constraintlayout.widget.ConstraintLayout>

* **Key**: Root layout uses wrap\_content to avoid overflow.

**4. Model Class (**Shape.java**)**

java

public class Shape {

private int shapeImage; *// int for drawable resource ID*

private String shapeName;

*// Constructor*

public Shape(int shapeImage, String shapeName) {

this.shapeImage = shapeImage;

this.shapeName = shapeName;

}

*// Getters*

public int getShapeImage() { return shapeImage; }

public String getShapeName() { return shapeName; }

}

* **Purpose**: Encapsulates data for grid items.

**5. Custom Adapter (**MyCustomAdapter.java**)**

java

public class MyCustomAdapter extends ArrayAdapter<Shape> {

private ArrayList<Shape> shapesArray;

private Context context;

*// Constructor*

public MyCustomAdapter(ArrayList<Shape> shapesArray, Context context) {

super(context, R.layout.grid\_item\_layout, shapesArray);

this.shapesArray = shapesArray;

this.context = context; }

*// ViewHolder Class*

private static class MyViewHolder {

ImageView shapeImage;

TextView shapeName;

}

@Override

public View getView(int position, View convertView, ViewGroup parent) {

Shape shape = getItem(position); *// Get current shape*

MyViewHolder holder;

*// Inflate new view if no recycled view exists*

if (convertView == null) {

holder = new MyViewHolder();

LayoutInflater inflater = LayoutInflater.from(getContext());

convertView = inflater.inflate(R.layout.grid\_item\_layout, parent, false);

*// Bind views to ViewHolder*

holder.shapeImage = convertView.findViewById(R.id.imageView);

holder.shapeName = convertView.findViewById(R.id.textView);

convertView.setTag(holder); *// Store holder in tag*

} else {

holder = (MyViewHolder) convertView.getTag(); *// Reuse holder*

}

*// Set data*

holder.shapeImage.setImageResource(shape.getShapeImage());

holder.shapeName.setText(shape.getShapeName());

return convertView;

}

}

* **Key APIs**: LayoutInflater, View.setTag(), View.getTag().
* **Best Practices**:
  + Use ViewHolder pattern for smooth scrolling.
  + Recycle views with convertView to optimize performance.

**6. MainActivity Setup**

java

*// Initialize GridView and data*

GridView gridView = findViewById(R.id.grid);

ArrayList<Shape> shapesArray = new ArrayList<>();

*// Add shapes (images must be in res/drawable with lowercase names)*

shapesArray.add(new Shape(R.drawable.sphere, "Sphere"));

shapesArray.add(new Shape(R.drawable.cylinder, "Cylinder"));

shapesArray.add(new Shape(R.drawable.cube, "Cube"));

shapesArray.add(new Shape(R.drawable.prism, "Prism"));

*// Set adapter*

MyCustomAdapter adapter = new MyCustomAdapter(shapesArray, getApplicationContext());

gridView.setAdapter(adapter);

*// Handle item clicks*

gridView.setOnItemClickListener((parent, view, position, id) -> {

Intent intent;

switch (shapesArray.get(position).getShapeName()) {

case "Sphere":

intent = new Intent(MainActivity.this, SphereActivity.class);

break;

*// Add cases for other shapes (Cube, Cylinder, Prism)*

default:

return;

}

startActivity(intent);

});

**7. Shape Activity (e.g.,**SphereActivity.java**)**

java

*// activity\_sphere.xml contains:*

*// - EditText (id: editTextSphere)*

*// - Button (id: buttonCalculate)*

*// - TextView (id: textViewResult)*

public class SphereActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_sphere);

EditText radiusInput = findViewById(R.id.editTextSphere);

Button calculateBtn = findViewById(R.id.buttonCalculate);

TextView resultTv = findViewById(R.id.textViewResult);

calculateBtn.setOnClickListener(v -> {

*// Get input and convert to double*

String radiusStr = radiusInput.getText().toString();

double radius = Double.parseDouble(radiusStr);

*// Calculate volume: V = (4/3) \* π \* r³*

double volume = (4.0 / 3.0) \* Math.PI \* Math.pow(radius, 3);

*// Display result (rounded to 2 decimals)*

resultTv.setText("Volume = " + String.format("%.2f", volume) + " m³");

});

}

}

* **Key APIs**: Math.PI, Double.parseDouble().
* **Best Practices**:
  + Use String.format() for readable output.
  + Validate input (e.g., check for empty values).

**Tools & APIs Used**

* **Android APIs**: GridView, ArrayAdapter, LayoutInflater, Intent, Math.
* **Tools**: Android Studio Layout Editor, Resource Manager (for images).

**Best Practices & Alternatives**

1. **Adapter Optimization**:
   * Use RecyclerView with GridLayoutManager (modern alternative to GridView).
   * Implement DiffUtil for efficient data updates.
2. **Input Handling**:
   * Add input validation (e.g., TextWatcher to prevent invalid entries).
   * Use TextInputLayout for better UX.
3. **Navigation**:
   * Use NavComponent for simplified navigation (alternative to manual Intent).
4. **Math Operations**:
   * Use BigDecimal for precise calculations (e.g., financial apps).
5. **Image Loading**:
   * Use Glide or Picasso for dynamic image loading (not covered).

**Part B: Important Topics Not Covered**

1. **RecyclerView**:
   * Modern replacement for GridView/ListView.
   * Better performance with built-in view recycling.
   * Steps:
     + Extend RecyclerView.Adapter.
     + Use GridLayoutManager for grid layouts.
     + Implement ViewHolder pattern (similar to adapter example).
2. **Input Validation**:
   * Prevent crashes from invalid inputs (e.g., non-numeric values).
   * Example:

java

if (radiusStr.isEmpty()) {

radiusInput.setError("Enter radius");

return;

}

1. **Fragments**:
   * Use fragments instead of activities for shape calculations.
   * Reduces memory overhead and simplifies navigation.
2. **ViewModel & LiveData**:
   * Retain data during configuration changes (e.g., screen rotation).
   * Follow MVVM architecture for separation of concerns.
3. **Dependency Injection (Dagger/Hilt)**:
   * Manage dependencies (e.g., math libraries) cleanly.
4. **Unit Testing**:
   * Test volume calculations with JUnit.
   * Example:

java

@Test

public void testSphereVolume() {

double volume = calculateSphereVolume(10.0);

assertEquals(4188.79, volume, 0.01); *// Allow delta for floating-point precision*

}

1. **Localization**:
   * Support multiple languages via res/values-<lang> strings.
2. **Accessibility**:
   * Add contentDescription to ImageViews for screen readers.
3. **Material Design**:
   * Use MaterialButton, MaterialTextView for consistent UI.

**Summary**

This section covered building a volume calculator app using GridView and custom adapters. Key takeaways include creating model classes, optimizing adapters with the ViewHolder pattern, handling click events, and navigating between activities. For production apps, modern approaches like RecyclerView, input validation, and architecture components (ViewModel, LiveData) are recommended.