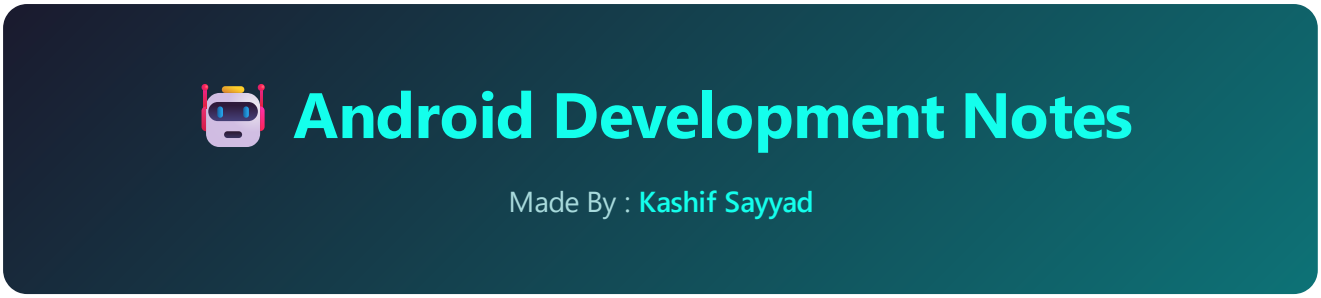
Android1



🟡 Module I — Activities, Fragments, Intents & Android UI

🔹 Introduction to Activities

An **Activity** in Android is a single screen with a user interface — it is one of the fundamental building blocks of an Android application. Every screen a user sees in an Android app is typically an Activity. Activities are Java/Kotlin classes that extend `android.app.Activity` or more commonly `AppCompatActivity` (from AndroidX) which provides backward compatibility. Each activity has its own layout XML file that defines the UI. An Android app can have multiple activities — for example, a login screen, a home screen, and a settings screen are each separate activities. Activities are declared in the `AndroidManifest.xml` file using the `<activity>` tag — the launcher activity (the first screen) is identified by the `MAIN` action and `LAUNCHER` category in the intent filter. Activities interact with each other through **Intents**. Managing activities properly is crucial for providing a smooth and responsive user experience.

🔹 Activity Lifecycle

The **Activity Lifecycle** defines the various states an activity goes through from creation to destruction, managed by the Android operating system. Understanding the lifecycle is critical for managing resources, saving data, and ensuring smooth user experience. The lifecycle is controlled through seven callback methods that the system calls at specific points.

`onCreate()` — Called when the activity is first created. This is where you initialize the activity — set the content view with `setContentView()`, initialize UI components, and set up data. Called only once per activity instance.

`onStart()` — Called when the activity becomes visible to the user but not yet interactive. The activity is about to come to the foreground.

`onResume()` — Called when the activity starts interacting with the user. The activity is now in the foreground and has focus. This is where you start animations, open camera, and resume paused operations.

`onPause()` — Called when the system is about to resume another activity. The activity is partially visible. Save lightweight data here — do NOT do heavy operations as it delays the transition.

`onStop()` — Called when the activity is no longer visible to the user. Release heavy resources, stop animations, and save app data here.

`onRestart()` — Called when a stopped activity is about to restart before calling `onStart()`.

`onDestroy()` — Called before the activity is destroyed. Final cleanup — release all resources, stop background threads.

```
onCreate() → onStart() → onResume() → [Running]
                                          ↓
                                      onPause()
                                          ↓
                                       onStop()
                                       ↙      ↘
                                onRestart()   onDestroy()
                                    ↓
                                onStart()
```

🔹 Introduction to Intents

An **Intent** in Android is a messaging object used to request an action from another app component — it is the primary mechanism for communication between activities, services, and broadcast receivers. Intents carry information about what operation should be performed and optionally the data to operate on. There are two types of intents — **Explicit Intents** specify the exact component (class name) to start — used to start a specific activity within your own app. **Implicit Intents** do not specify a component but declare a general action to perform — the Android system finds the appropriate component to handle it based on intent filters declared in `AndroidManifest.xml`. For example, an implicit intent to view a URL allows the system to choose the installed browser. Intents can carry extra data as key-value pairs using `putExtra()` and retrieve them with `getIntent().getStringExtra()` etc. Intents are also used to start services, deliver broadcasts, and return results from activities using `startActivityForResult()` (deprecated in favor of Activity Result API).

🔹 Linking Activities Using Intents

Linking activities using intents is the standard way to navigate between screens in an Android application. To start a new activity, you create an explicit Intent specifying the current context and the target Activity class, then call `startActivity(intent)`. Data can be passed between activities by attaching extras to the intent using `intent.putExtra("key", value)` before calling `startActivity()`. The receiving activity retrieves the data using `getIntent().getStringExtra("key")` (or appropriate typed method) in its `onCreate()` method. To receive a result back from a started activity, you use the **Activity Result API** — register a launcher with `registerForActivityResult()`, start the activity with the launcher, and handle the result in the callback. The back stack manages the history of activities — pressing the back button pops the current activity and returns to the previous one. Activity transitions can be animated using `overridePendingTransition()` for custom enter/exit animations.

```
// Starting a new activity with data
Intent intent = new Intent(MainActivity.this, ProfileActivity.class);
intent.putExtra("username", "Kashif");
intent.putExtra("age", 21);
startActivity(intent);

// In ProfileActivity.java
String name = getIntent().getStringExtra("username");
int age = getIntent().getIntExtra("age", 0);
```

🔹 Calling Built-in Applications Using Intents

One of the most powerful features of Android's Intent system is the ability to invoke built-in system applications without writing any of their functionality yourself. Using **implicit intents**, your app can request system apps to perform specific actions. To make a phone call, use `Intent(Intent.ACTION_CALL, Uri.parse("tel:+91XXXXXXXXXX"))` — requires `CALL_PHONE` permission. To open a website, use `Intent(Intent.ACTION_VIEW, Uri.parse("https://www.example.com"))` which opens the default browser. To send an email, use `Intent(Intent.ACTION_SEND)` with type `"message/rfc822"` and email extras. To open Google Maps at a location, use `Intent(Intent.ACTION_VIEW, Uri.parse("geo:latitude,longitude"))`. To capture a photo using the camera app, use `MediaStore.ACTION_IMAGE_CAPTURE`. To share content with other apps, use `Intent.ACTION_SEND` with the appropriate MIME type. Always check if an app can handle your intent using `intent.resolveActivity(getPackageManager()) != null` before calling `startActivity()` to avoid crashes if no app can handle the intent.

🔹 Introduction to Fragments

A **Fragment** represents a reusable portion of your app's UI — it is like a sub-activity that has its own layout, lifecycle, and behavior, but must be hosted within an Activity. Fragments were introduced in Android 3.0 (API 11) to support flexible UI designs for tablets and large screens, where you might want to show multiple UI panels side by side. A Fragment extends `Fragment` class (from AndroidX) and has its own lifecycle callbacks. The fragment's layout is defined in XML and inflated in the `onCreateView()` method. Fragments communicate with their host Activity through interfaces or shared ViewModels (modern approach). Multiple fragments can be displayed in a single Activity simultaneously — for example, a list fragment on the left and a detail fragment on the right on tablets, while on phones they each take the full screen. Fragments are managed by the **FragmentManager** and can be added to a **back stack**, allowing users to navigate back through fragment transactions with the back button.

🔹 Adding Fragments Dynamically

Fragments can be added to an Activity either **statically** (defined directly in the Activity's XML layout using `<fragment>` tag) or **dynamically** (added, replaced, or removed at runtime through code). Dynamic fragment management uses the **FragmentManager** and **FragmentTransaction**. You obtain the FragmentManager with `getSupportFragmentManager()`, begin a transaction with `beginTransaction()`, perform operations like `add()`, `replace()`, or `remove()`, optionally call `addToBackStack(null)` to allow back navigation, and commit the transaction with `commit()`. The `replace()` operation removes the current fragment in a container and replaces it with a new one — used for navigation. The `add()` operation adds a fragment on top without removing the existing one. Container views (like `FrameLayout`) in the Activity layout serve as placeholders where fragments are placed. Dynamic fragments are essential for building flexible, responsive UIs that adapt to different screen sizes and user interactions.

```
// Dynamically adding a fragment
FragmentManager fm = getSupportFragmentManager();
FragmentTransaction ft = fm.beginTransaction();
ft.replace(R.id.fragment_container, new ProfileFragment());
ft.addToBackStack(null);  // allows back navigation
ft.commit();
```

🔹 Lifecycle of a Fragment

The **Fragment Lifecycle** is more complex than the Activity lifecycle because a fragment has its own lifecycle that is also tied to the lifecycle of its host Activity. Understanding fragment lifecycle is essential for proper resource management and avoiding memory leaks.

`onAttach()` — Fragment is attached to its host Activity. The Activity reference becomes available here.

`onCreate()` — Fragment is created. Initialize non-UI components here. The fragment's view has NOT been created yet.

`onCreateView()` — Fragment creates and returns its UI layout. Inflate the fragment's XML layout here and return the root view.

`onViewCreated()` — Called after `onCreateView()`. This is the ideal place to initialize UI components and set up observers since the view hierarchy is fully created.

`onStart()` — Fragment becomes visible.

`onResume()` — Fragment is interactive and in the foreground.

`onPause()` — Fragment is partially obscured or losing focus.

`onStop()` — Fragment is no longer visible.

`onDestroyView()` — Fragment's view is being destroyed. Clean up view-related resources and references here to avoid memory leaks.

`onDestroy()` — Fragment itself is being destroyed.

`onDetach()` — Fragment is detached from its Activity.

🔹 Toast in Android

A **Toast** is a small, non-interactive popup message that appears briefly on the screen to provide simple feedback to the user and then automatically disappears without requiring any user action. Toasts do not steal focus from the current activity and are perfect for showing quick status messages like "File saved successfully", "No internet connection", or "Item deleted". A Toast is created using `Toast.makeText(context, "message", duration)` where duration is either `Toast.LENGTH_SHORT` (2 seconds) or `Toast.LENGTH_LONG` (3.5 seconds). The `show()` method displays the toast. By default, toasts appear at the bottom center of the screen, but you can reposition them using `setGravity(Gravity.TOP, xOffset, yOffset)`. Custom toasts with custom layouts can be created by inflating a custom XML layout and setting it with `setView()`. However, **custom toasts are deprecated in Android 11 (API 30)** for apps targeting API 30+, and only text-based toasts from the background are restricted in API 33+.

```
// Simple Toast
Toast.makeText(this, "Hello Kashif!", Toast.LENGTH_SHORT).show();

// Toast with custom position
Toast toast = Toast.makeText(this, "Saved!", Toast.LENGTH_LONG);
```

```
toast.setGravity(Gravity.TOP | Gravity.CENTER_HORIZONTAL, 0, 100);
toast.show();
```

🔹 Android User Interface — Understanding Screen Components

The Android user interface is built on a hierarchy of **View** and **ViewGroup** objects. A **View** is the basic building block of UI — it occupies a rectangular area on the screen and is responsible for drawing and event handling. Common View subclasses include `TextView`, `ImageView`, `Button`, `EditText`, `CheckBox`, `RadioButton`, and `Switch`. A **ViewGroup** is a container that holds other Views or ViewGroups and defines their layout — common ViewGroup subclasses include `LinearLayout`, `RelativeLayout`, `ConstraintLayout`, and `FrameLayout`. The UI is defined in **XML layout files** stored in the `res/layout/` directory and inflated at runtime using `setContentView()` in Activities or `inflater.inflate()` in Fragments. Every view has attributes for **position** (margins, padding), **size** (`match_parent`, `wrap_content`, specific dp values), **appearance** (background, text color, font), and **behavior** (click listeners, visibility). The **dp (density-independent pixels)** unit ensures consistent sizing across different screen densities.

🔹 Views and View Groups

Views are the fundamental UI elements in Android — each one draws something on the screen and can respond to user input. Views have a rich set of XML attributes and corresponding Java/Kotlin methods for customization. Key view properties include `id` (unique identifier for finding views in code), `layout_width` and `layout_height` (size), `padding` (space inside the view), `margin` (space outside the view), `background` (color or drawable), `visibility` (VISIBLE, INVISIBLE, GONE), and `onClick` (click handler method name). **ViewGroups** are invisible containers that arrange their child views according to specific rules. The `addView()` method adds views dynamically at runtime. **View binding** (modern approach) and `findViewById()` are used to get references to views in code. The view hierarchy should be kept as **shallow as possible** for better rendering performance — deeply nested layouts cause slow UI rendering. Android's layout inspector tool in Android Studio helps visualize and optimize the view hierarchy.

🔹 Linear Layout

LinearLayout is one of the simplest and most commonly used ViewGroups in Android — it arranges its child views in a single row or column, one after another. The `orientation` attribute controls the direction — `horizontal` places children side by side (left to right), while `vertical`

stacks children one below the other. The `layout_weight` attribute allows children to proportionally share the available space — a child with `layout_weight="1"` in a horizontal LinearLayout with all children having equal weights gets equal width. Setting `layout_width="0dp"` with a weight tells LinearLayout to distribute remaining space by weight ratio. `gravity` controls how the LinearLayout aligns its content within itself, while `layout_gravity` controls how a child positions itself within the LinearLayout. LinearLayouts can be nested — a vertical LinearLayout containing horizontal LinearLayouts creates a grid-like structure. However, deeply nested LinearLayouts are performance-intensive and should be replaced with ConstraintLayout for complex designs.

🔹 Absolute Layout

AbsoluteLayout is a layout in Android that positions its children at **exact X and Y coordinates** on the screen. Each child view specifies its position using `layout_x` and `layout_y` attributes which define the distance from the top-left corner of the layout. While this gives pixel-perfect control over element positioning, it is extremely problematic for responsive design because the fixed positions look correct only on the specific screen size they were designed for — on a different screen size or orientation, elements may overlap, go off-screen, or leave unwanted gaps. AbsoluteLayout was **deprecated in Android API level 3** (very early in Android's history) and should never be used in modern Android development. It has been completely replaced by more flexible layouts like **RelativeLayout** and **ConstraintLayout** that use relationship-based positioning to create designs that adapt to different screen sizes. AbsoluteLayout is mentioned in syllabuses for historical context and understanding of Android layout evolution.

🔹 Table Layout

TableLayout is a ViewGroup that arranges its children in a grid of rows and columns, similar to an HTML table. It contains `TableRow` children, and each `TableRow` contains the actual cell views (TextViews, Buttons, etc.). Columns are created implicitly based on the number of cells in each row. The `layout_column` attribute specifies which column a view occupies, and `layout_span` allows a view to span multiple columns. The `stretchColumns` attribute specifies which column indices should stretch to fill available width — `stretchColumns="*"` stretches all columns equally. The `shrinkColumns` attribute specifies which columns can be shrunk if the table is too wide. `collapseColumns` hides specific columns. TableLayout is well-suited for displaying form-like data, comparison tables, and structured data with labeled rows. However, for complex responsive grids, **GridLayout** or **RecyclerView with GridLayoutManager** are more flexible and performant alternatives in modern Android development.

🔹 Relative Layout

RelativeLayout is a flexible ViewGroup where each child view positions itself relative to other views or relative to the parent layout boundaries. Instead of using absolute coordinates, views specify their position using relationship attributes. Common attributes include `layout_alignParentTop`, `layout_alignParentBottom`, `layout_alignParentLeft`, `layout_alignParentRight` (position relative to parent edges), `layout_centerInParent`, `layout_centerHorizontal`, `layout_centerVertical` (centering options), `layout_below`, `layout_above`, `layout_toRightOf`, `layout_toLeftOf` (position relative to other views — referenced by their IDs), and `layout_alignTop`, `layout_alignBottom`, `layout_alignLeft`, `layout_alignRight` (alignment with other view edges). RelativeLayout eliminates the need for deeply nested LinearLayouts, resulting in flatter, more efficient view hierarchies. While largely superseded by **ConstraintLayout** (which is more powerful and performant), RelativeLayout is still widely used and understanding it is fundamental to Android UI development.

🔹 Frame Layout

FrameLayout is the simplest ViewGroup in Android — it is designed to hold a **single child view**, though it can contain multiple children. When multiple children are added, they are all drawn on top of each other (layered) starting from the top-left corner of the FrameLayout. The most recently added child is drawn on top. The `layout_gravity` attribute controls how each child is positioned within the FrameLayout — options include `top`, `bottom`, `left`, `right`, `center`, `center_horizontal`, and `center_vertical`. FrameLayout is most commonly used as a **container for Fragments** — a `FrameLayout` in an Activity's layout serves as the placeholder into which Fragments are dynamically added and replaced. It is also used for overlapping UI elements — for example, placing a badge count on top of an icon, or displaying a loading spinner overlay over content. Its simplicity and layering behavior make it the go-to choice for fragment containers and overlay effects.

🔹 Scroll View

ScrollView is a ViewGroup that allows its content to be scrolled vertically when the content is too tall to fit on the screen. It can contain only **one direct child** — typically a LinearLayout that holds all the scrollable content. The single child can itself contain many child views, creating a long scrollable page. ScrollView extends **FrameLayout**. For horizontal scrolling, **HorizontalScrollView** is used instead. Key attributes include `fillViewport` (set to `true` to stretch the child to fill the scroll view when content is shorter than the screen) and `scrollbars` (to control scrollbar visibility). Programmatic scrolling is done using `scrollTo(x, y)` (absolute) or `scrollBy(dx, dy)` (relative), and `smoothScrollTo()` for animated scrolling. **NestedScrollView** (from AndroidX) is the modern replacement that supports nested scrolling

behaviors and works better with CoordinatorLayout for collapsing toolbars and other Material Design patterns. For long lists of items, always use **RecyclerView** instead of ScrollView for much better performance.

🔹 Constraint Layout

ConstraintLayout is the most powerful and flexible layout in Android, introduced in 2016 and now the default layout for new Android projects. It allows creating large, complex layouts with a **flat view hierarchy** (no nested ViewGroups) by defining constraints between views and between views and the parent. Each view must have at least one horizontal and one vertical constraint. Constraints can be set to **parent edges**, **other view edges**, or **guidelines** (invisible helper lines). **Bias** (0.0 to 1.0) controls the position of a view between two constrained edges. **Chains** link multiple views together with various chain styles — `spread`, `spread_inside`, and `packed`. **Guidelines** are invisible reference lines at a fixed position or percentage for aligning multiple views. **Barriers** adjust dynamically based on the size of referenced views. ConstraintLayout is deeply integrated with Android Studio's **Layout Editor** which provides a WYSIWYG drag-and-drop design surface. Using ConstraintLayout results in faster layout inflation and better scroll performance compared to nested layouts.

🔹 Adapting to Display Orientation

Android apps must handle **screen orientation changes** gracefully — by default, when the device rotates, Android destroys and recreates the current Activity, which causes loss of any data not properly saved. There are several approaches to handle orientation changes. **ViewModel** (from Android Architecture Components) stores UI-related data that survives configuration changes — it is the modern recommended approach. `onSaveInstanceState(Bundle)` and `onRestoreInstanceState(Bundle)` allow saving and restoring lightweight data (primitives, strings) across Activity recreations. You can also configure specific Activities to handle orientation changes themselves by adding `android:configChanges="orientation|screenSize"` to the activity tag in `AndroidManifest.xml` — this prevents recreation but requires you to manually update the UI in `onConfigurationChanged()`. Providing **separate layout files** for portrait (`res/layout/`) and landscape (`res/layout-land/`) orientations allows you to design optimized UIs for each orientation. `getResources().getConfiguration().orientation` checks the current orientation programmatically.

🔹 Anchoring Views, Resizing, Repositioning & Split Screen

Anchoring views refers to connecting UI elements to fixed reference points — in ConstraintLayout, this is done through constraints to parent edges, other views, or guidelines, ensuring views maintain their relative positions across different screen sizes. **Resizing views** dynamically is done using `ViewGroup.LayoutParams` — you can change width and height programmatically by getting a view's `layoutParams`, modifying them, and calling `requestLayout()`. **Repositioning views** programmatically in RelativeLayout involves updating `RelativeLayout.LayoutParams` with new alignment rules. In ConstraintLayout, you can animate constraint changes using `ConstraintSet` and `TransitionManager.beginDelayedTransition()`. **Split Screen (Multi-Window)** was introduced in Android 7.0 (API 24) — it allows two apps (or two activities) to run side by side. Apps support multi-window by default in API 24+ unless `android:resizeableActivity="false"` is set. The `onMultiWindowModeChanged()` callback notifies your activity of split-screen mode changes. Apps should handle configuration changes properly to work well in split-screen mode.

🔴 Module II — Part 2A: Fragments, Images, Menus & Video

🔹 Using a DialogFragment

DialogFragment is a fragment subclass that displays a floating dialog window on top of the current activity. It is the modern recommended way to show dialogs in Android — replacing the older `Dialog` class used directly. DialogFragment properly handles configuration changes (like rotation) that would crash a plain Dialog. You create a DialogFragment by extending `DialogFragment` and overriding `onCreateDialog()` to return an `AlertDialog` built with `AlertDialog.Builder`, or overriding `onCreateView()` to return a fully custom layout for more complex dialogs. The dialog is shown by calling `dialogFragment.show(supportFragmentManager, "tag")`. DialogFragment has its own lifecycle tied to the fragment lifecycle. `setCancelable(false)` prevents dismissal by tapping outside or pressing back. Results are communicated back to the host Activity/Fragment using the **Fragment Result API** (`setFragmentResult()` and `setFragmentResultListener()`) — the modern replacement for interfaces and `setTargetFragment()`. Common uses include confirmation dialogs, input forms, and progress dialogs.

🔹 Displaying Pictures and Menus Using Image Views

ImageView is the standard view for displaying images in Android. It supports various image sources — drawable resources (`setImageResource(R.drawable.image)`), Bitmap objects (`setImageBitmap(bitmap)`), URIs (`setImageURI(uri)`), and drawable objects. The `scaleType` attribute controls how the image is scaled and positioned within the ImageView

bounds — common values include `centerCrop` (fills bounds, crops excess), `fitCenter` (fits inside bounds maintaining ratio), `centerInside` (similar but never scales up), `fitXY` (stretches to fill, may distort), and `center` (no scaling, centered). For loading images from URLs, always use image loading libraries — **Glide** and **Picasso** are the most popular, handling caching, threading, placeholder images, error images, and transformations automatically. Never load network images on the main thread. **Menus** in Android (Options Menu, Context Menu, Popup Menu) can display icons using ImageView-like menu item icons set via `android:icon` attribute in the menu XML resource file.

🔹 Gallery and ImageView Views

The **Gallery widget** was a horizontally scrolling view that displayed items as a centered selection — it was used for image galleries and carousels in early Android versions. However, **Gallery was deprecated in API 16** and should not be used in modern apps. The modern replacements are **RecyclerView with LinearLayoutManager (HORIZONTAL)** for horizontal lists, **ViewPager2** for swipeable full-screen image galleries, and **RecyclerView with GridLayoutManager** for grid photo galleries. **ImageView** remains the core widget for displaying individual images. For building a photo gallery feature, the typical modern approach combines a RecyclerView grid showing thumbnail ImageViews (loaded with Glide/Picasso), with clicking an item opening a full-screen ViewPager2 for swiping through images. **Picasso** simplifies image loading with one line: `Picasso.get().load(url).into(imageView)`. **Glide** offers more features including GIF support and video thumbnails: `Glide.with(context).load(url).placeholder(R.drawable.loading).into(imageView)`.

🔹 Image Switcher

ImageSwitcher is a ViewSwitcher subclass specifically designed for switching between two ImageViews with an animation — it provides smooth animated transitions when changing the displayed image. ImageSwitcher implements the `ViewSwitcher.ViewFactory` interface, requiring you to implement `makeView()` which returns a new ImageView for the switcher to manage. You set images using `setImageResource()`, `setImageDrawable()`, or `setImageURI()` — each call switches to the next image with the specified animation. In and Out animations are set using `setInAnimation()` and `setOutAnimation()` — common animations include slide-in-left, slide-in-right, fade-in, and fade-out from `android.R.anim`. ImageSwitcher is useful for simple image slideshows, product image cycling, and avatar selection screens. For more complex scenarios with swipe gestures and better performance, **ViewPager2** is the preferred modern alternative. ImageSwitcher is typically combined with a Gallery or navigation buttons to create a complete image browsing experience.

🔹 Grid View

GridView is a ViewGroup that displays items in a two-dimensional scrollable grid — each cell in the grid displays one item from an adapter. Key attributes include `numColumns` (number of columns — can be `auto_fit` to automatically calculate based on `columnWidth`), `columnWidth` (width of each column), `horizontalSpacing` and `verticalSpacing` (gaps between cells), and `stretchMode` (how columns stretch to fill available width). Like ListView, GridView uses an **Adapter** (typically ArrayAdapter or custom BaseAdapter) to supply views for each cell. The `setOnItemClickListener()` handles cell taps. GridView also benefits from the **ViewHolder pattern** for performance. **Note:** GridView is legacy — **RecyclerView with GridLayoutManager** is the modern replacement. RecyclerView with GridLayoutManager is far more flexible — it supports variable span sizes (some items spanning multiple columns), item animations, drag-and-drop reordering, and swipe-to-dismiss gestures out of the box. GridView is still commonly used in older apps and is important to understand for maintaining legacy code.

🔹 Using Menus with Views — Options Menu & Context Menu

Android provides three types of menus for presenting actions to users. The **Options Menu** appears in the app bar (toolbar) when the user taps the overflow icon (three dots) or presses the Menu button on older devices. It is created by overriding `onCreateOptionsMenu()` and inflating a menu XML resource. Menu item selections are handled in `onOptionsItemSelected()`. Menu items can appear directly in the toolbar as action icons (using `app:showAsAction="always"` or `"ifRoom"`) or in the overflow dropdown. The **Context Menu** appears as a floating list when the user performs a long press on a view. Register a view for context menu with `registerForContextMenu(view)`, create it in `onCreateContextMenu()`, and handle selections in `onContextItemSelected()`. **Popup Menu** shows a dropdown menu anchored to a specific view — created with `new PopupMenu(context, anchorView)`, inflated with `popupMenu.inflate(R.menu.menu_resource)`, and shown with `popupMenu.show()`. Menus are defined in XML files in the `res/menu/` directory.

🔹 Creating the Helper Methods — Options Menu & Context Menu

Helper methods for menus are utility methods that keep menu-related code organized and reusable. For the Options Menu, helper methods can handle complex logic like dynamically changing menu item visibility, updating menu item text/icons based on app state, or grouping related menu actions. `menu.findItem(R.id.menu_item_id)` retrieves a specific menu item programmatically — you can then call `.setVisible()`, `.setEnabled()`, `.setTitle()`, or `.setIcon()` on it. Call `invalidateOptionsMenu()` to trigger `onPrepareOptionsMenu()` which allows updating menu items before they are displayed. For the Context Menu, `AdapterContextMenuInfo` (retrieved from `item.getMenuInfo()`) provides the position and ID

of the item that was long-pressed in a ListView or GridView — this is essential for knowing which data item the context action applies to. Helper methods that encapsulate "perform action on item at position X" logic make context menu handling clean and maintainable. Always use `MenuCompat` and `MenuItemCompat` for backward compatibility.

🔹 VideoView

VideoView is a View that displays video content and provides basic playback controls. It handles all the complexities of video decoding and rendering internally. Videos can be loaded from local storage using a URI (`videoView.setVideoURI(Uri.fromFile(file))`), from raw resources (`videoView.setVideoURI(Uri.parse("android.resource://" + packageName + "/" + R.raw.video))`), or from a network URL (`videoView.setVideoURI(Uri.parse("https://..."))`). A **MediaController** can be attached to provide playback controls (play, pause, seek bar, fast-forward, rewind) — `mediaController.setAnchorView(videoView)` anchors the controls to the VideoView. Playback is controlled with `videoView.start()` , `videoView.pause()` , `videoView.stopPlayback()` , and `videoView.seekTo(milliseconds)` . `setOnPreparedListener()` is called when the video is ready to play, `setOnCompletionListener()` when playback finishes, and `setOnErrorListener()` for error handling. For more advanced video playback needs (adaptive streaming, DRM, custom UI), **ExoPlayer** (now part of Android's Media3 library) is the professional choice used by YouTube and other major apps.

🔹 Play Video from URL Using VideoView

Playing a video from a URL using VideoView requires internet permission declared in `AndroidManifest.xml` (`<uses-permission android:name="android.permission.INTERNET"/>`). The video URL is set using `videoView.setVideoURI(Uri.parse(videoUrl))` . Since network operations involve loading time, you should show a ProgressBar while the video buffers — hide it in `onPreparedListener` when the video is ready. Always call `videoView.requestFocus()` before starting playback. The VideoView automatically handles buffering but does not provide built-in buffering progress feedback. To save and restore playback position across configuration changes (like screen rotation), save the current position with `videoView.getCurrentPosition()` in `onSaveInstanceState()` and restore it with `videoView.seekTo(savedPosition)` in `onRestoreInstanceState()` . For streaming from URLs, VideoView uses the platform's built-in `MediaPlayer` which supports HTTP/HTTPS streams but has limitations with advanced streaming formats like HLS and DASH — for these, **ExoPlayer** is strongly recommended.

```
String videoUrl = "https://www.example.com/sample.mp4";
VideoView videoView = findViewById(R.id.videoView);
MediaController mc = new MediaController(this);
mc.setAnchorView(videoView);
videoView.setMediaController(mc);
videoView.setVideoURI(Uri.parse(videoUrl));
videoView.setOnPreparedListener(mp -> {
    progressBar.setVisibility(View.GONE);
    videoView.start();
});
```

🔹 VideoView Create & Optimized VideoView in ListView

Creating a VideoView properly involves careful lifecycle management to avoid memory leaks and battery drain. In `onPause()`, always call `videoView.pause()` and save the current position. In `onStop()`, call `videoView.stopPlayback()` to release the MediaPlayer. In `onResume()`, restore the position with `seekTo()` and resume playback. **Optimized VideoView in ListView** is a particularly challenging scenario because ListView recycles views — a VideoView in a list item can cause issues like wrong videos playing in wrong cells, multiple videos playing simultaneously, and severe performance degradation. The standard solution is to play video only for the **currently visible/selected item** and use thumbnail images for all others. Clicking a thumbnail replaces it with a VideoView that starts playing. **RecyclerView** handles this better than ListView. For true video-in-list scenarios (like Instagram Reels or TikTok style), **ExoPlayer** with a shared player instance and proper attach/detach management per visible item is the professional approach used in production apps.

🔴 Module II — Part 2B: SQLite, SMS Messaging & Email

🔹 Introduction to SQLite in Android

SQLite is a lightweight, serverless, self-contained relational database engine that is built directly into the Android operating system — every Android device has SQLite available without any additional installation or configuration. Unlike server-based databases like MySQL or PostgreSQL, SQLite stores the entire database as a single file on the device's internal storage, making it perfect for mobile applications. Android provides the `android.database.sqlite` package with classes to create, manage, and interact with SQLite databases. SQLite supports standard SQL syntax including CREATE, INSERT, UPDATE, DELETE, and SELECT statements. It supports most SQL data types — INTEGER, TEXT, REAL, BLOB, and NULL. SQLite databases in Android are stored in the app's private directory at

`/data/data/com.your.package/databases/` and are not accessible by other apps by default. SQLite is ideal for storing structured, relational data that needs to persist between app sessions — user profiles, cached data, offline content, settings, and transaction records.

🔹 SQLiteOpenHelper and SQLiteDatabase

`SQLiteOpenHelper` is the key class in Android for managing SQLite database creation and version management. You create a database helper by extending `SQLiteOpenHelper` and implementing two essential callback methods. `onCreate(SQLiteDatabase db)` is called only once when the database is first created — this is where you execute `CREATE TABLE` SQL statements to set up your database schema. `onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)` is called when the database version number increases — here you handle schema migrations like adding new columns, creating new tables, or dropping old ones. The helper is instantiated with `new MyDatabaseHelper(context, "dbname.db", null, version)`. `SQLiteDatabase` is the class that actually represents the database and provides methods for all database operations — `execSQL()` for DDL statements, `insert()`, `update()`, `delete()`, and `query()` for DML operations, and `rawQuery()` for executing raw SELECT statements. Get a writable database instance with `helper.getWritableDatabase()` and readable with `helper.getReadableDatabase()`.

```
public class DatabaseHelper extends SQLiteOpenHelper {
    private static final String DB_NAME = "StudentDB";
    private static final int DB_VERSION = 1;

    public DatabaseHelper(Context context) {
        super(context, DB_NAME, null, DB_VERSION);
    }

    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL("CREATE TABLE students (id INTEGER PRIMARY KEY
AUTOINCREMENT, " +
                   "name TEXT, marks INTEGER)");
    }

    @Override
    public void onUpgrade(SQLiteDatabase db, int oldV, int newV) {
        db.execSQL("DROP TABLE IF EXISTS students");
        onCreate(db);
    }
}
```

🔹 Creating, Opening and Closing Database

Creating a SQLite database in Android is handled automatically by `SQLiteOpenHelper` — the database file is created the first time `getWritableDatabase()` or `getReadableDatabase()` is called. The `onCreate()` callback fires only if the database doesn't already exist, executing your `CREATE TABLE` statements. **Opening** the database is done by calling `getWritableDatabase()` (returns a `SQLiteDatabase` object with read-write access) or `getReadableDatabase()` (returns read-only access when disk is full, otherwise same as writable). These methods should not be called on the main thread in production apps — use background threads, `AsyncTask` (deprecated), `ExecutorService`, or **Room** (modern approach) to avoid blocking the UI. **Closing** the database is important to release resources — call `db.close()` after operations are complete, or `helper.close()` to close the helper. In practice, many apps keep the database open for the app's lifetime and close it in `onDestroy()`. Always use try-finally blocks to ensure the database is closed even if an exception occurs.

🔹 Working with Cursors

A **Cursor** in Android SQLite is an interface that provides random read-write access to the result set of a database query — it acts as a pointer to rows in the query results, similar to cursors in PL/SQL. A cursor is obtained by calling `db.rawQuery("SELECT * FROM students", null)` or `db.query(table, columns, selection, selectionArgs, groupBy, having, orderBy)`. The cursor initially points **before the first row** — you must call `cursor.moveToFirst()` to move to the first row. Navigation methods include `moveToFirst()`, `moveToNext()`, `moveToPrevious()`, `moveToLast()`, `moveToPosition(index)`, and `isAfterLast()`. Data is retrieved using typed getter methods — `cursor.getString(columnIndex)`, `cursor.getInt(columnIndex)`, `cursor.getDouble(columnIndex)` — where column index is obtained with `cursor.getColumnIndex("columnName")`. Always close the cursor after use with `cursor.close()` to avoid memory leaks. The `cursor.getCount()` method returns the total number of rows in the result set.

```
SQLiteDatabase db = helper.getReadableDatabase();
Cursor cursor = db.rawQuery("SELECT * FROM students", null);

if(cursor.moveToFirst()) {
    do {
        String name = cursor.getString(cursor.getColumnIndex("name"));
        int marks = cursor.getInt(cursor.getColumnIndex("marks"));
        Log.d("DB", name + " - " + marks);
    } while(cursor.moveToNext());
}
cursor.close();
db.close();
```

🔹 Insert, Update, Delete Operations

Android provides both **high-level convenience methods** and **raw SQL execution** for database manipulation. For **Insert**, use `db.insert(tableName, null, contentValues)` where `ContentValues` is a key-value map of column names to values. It returns the row ID of the newly inserted row (-1 if failed). For **Update**, use `db.update(tableName, contentValues, whereClause, whereArgs)` — the `whereClause` uses `?` placeholders replaced by `whereArgs` strings, preventing SQL injection. It returns the number of rows affected. For **Delete**, use `db.delete(tableName, whereClause, whereArgs)` which returns the number of deleted rows. The `whereArgs` are always passed as String arrays. Alternatively, `db.execSQL("INSERT INTO...")` executes raw SQL but doesn't return results or row counts. **ContentValues** is similar to a HashMap but specifically designed for SQLite operations — `values.put("name", "Kashif")`, `values.put("marks", 95)`.

```
// INSERT
ContentValues values = new ContentValues();
values.put("name", "Kashif");
values.put("marks", 95);
long id = db.insert("students", null, values);

// UPDATE
ContentValues updateVals = new ContentValues();
updateVals.put("marks", 98);
int rows = db.update("students", updateVals, "name=?", new String[]
{"Kashif"});

// DELETE
int deleted = db.delete("students", "id=?", new String[]{"1"});
```

🔹 Building and Executing Queries

Android provides the `db.query()` method as a structured way to build SELECT queries without writing raw SQL strings, reducing SQL injection risks. The parameters are `table` (table name), `columns` (array of column names to retrieve, null for all), `selection` (WHERE clause with `?` placeholders), `selectionArgs` (values for `?` placeholders as String array), `groupBy` (GROUP BY clause), `having` (HAVING clause), `orderBy` (ORDER BY clause), and `limit` (LIMIT clause). For complex queries involving JOINs, subqueries, or aggregate functions, `db.rawQuery(sql, selectionArgs)` is more appropriate. **SQLiteQueryBuilder** provides an even more structured approach for building complex queries. Always use parameterized queries (with `?` placeholders) instead of string concatenation to prevent **SQL Injection attacks**. For production Android apps, **Room Persistence Library** (part of Android Jetpack)

is strongly recommended — it provides compile-time SQL verification, LiveData integration, and eliminates most boilerplate code while using SQLite under the hood.

🔹 SMS Messaging — Sending SMS Programmatically

Android provides two ways to send SMS messages from your app. The first approach uses the **built-in SMS app** via an implicit Intent — `Intent(Intent.ACTION_SENDTO, Uri.parse("smsto:phoneNumber"))` with `intent.putExtra("sms_body", message)` opens the default SMS app pre-filled with the number and message, letting the user confirm and send. This requires **no special permissions**. The second approach sends SMS **directly from your app** using `SmsManager` — `SmsManager.getDefault().sendTextMessage(phoneNumber, null, message, sentIntent, deliveredIntent)`. This requires the `SEND_SMS` permission declared in `AndroidManifest.xml` and **runtime permission request** (since Android 6.0 / API 23). Long messages exceeding 160 characters must be split using `SmsManager.divideMessage(message)` and sent with `sendMultipartTextMessage()`. The `sentIntent` and `deliveredIntent` are PendingIntents that are broadcast when the message is sent and delivered respectively — allowing you to track message status.

```
// Method 1: Using built-in SMS app (no permission needed)
Intent smsIntent = new Intent(Intent.ACTION_SENDTO,
Uri.parse("smsto:+91XXXXXXXXXX"));
smsIntent.putExtra("sms_body", "Hello from Kashif's app!");
startActivity(smsIntent);

// Method 2: Send directly (requires SEND_SMS permission)
SmsManager smsManager = SmsManager.getDefault();
smsManager.sendTextMessage("+91XXXXXXXXXX", null, "Hello!", null, null);
```

🔹 Getting Feedback After Sending a Message

When sending SMS programmatically using `SmsManager`, Android provides a feedback mechanism through **PendingIntents** and **BroadcastReceivers** to track whether the message was successfully sent and delivered. For **sent feedback**, create a `PendingIntent` with a custom action string and pass it as the `sentIntent` parameter to `sendTextMessage()`. Register a `BroadcastReceiver` with the same action that receives the broadcast when the SMS send attempt completes. In the receiver's `onReceive()`, check `getResultCode()` — `Activity.RESULT_OK` means sent successfully, `SmsManager.RESULT_ERROR_GENERIC_FAILURE` means generic failure, `RESULT_ERROR_NO_SERVICE` means no network service, and `RESULT_ERROR_RADIO_OFF` means the device radio is turned off. For **delivery feedback**, pass a `deliveredIntent` PendingIntent — this fires when the recipient's device acknowledges

receipt of the message. Always **unregister BroadcastReceivers** in `onPause()` or `onDestroy()` to avoid memory leaks and unnecessary processing after the activity is gone.

```
// Sent feedback setup
String SENT = "SMS_SENT";
PendingIntent sentPI = PendingIntent.getBroadcast(this, 0, new Intent(SENT),
    PendingIntent.FLAG_IMMUTABLE);

registerReceiver(new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        if(getResultCode() == Activity.RESULT_OK)
            Toast.makeText(context, "SMS Sent!", Toast.LENGTH_SHORT).show();
        else
            Toast.makeText(context, "SMS Failed!", Toast.LENGTH_SHORT).show();
    }
}, new IntentFilter(SENT));

SmsManager.getDefault().sendTextMessage(number, null, message, sentPI, null);
```

🔹 Sending SMS Messages Using Intents

Using **Intents to send SMS** is the safest and most user-friendly approach — it delegates the actual sending to the user's default SMS app, avoiding the need for `SEND_SMS` permission entirely. The `ACTION_SENDTO` action with a `smsto:` URI scheme opens the default messaging app with the recipient number pre-filled. The message body is attached using `putExtra("sms_body", messageText)`. For sending to multiple recipients, separate numbers with semicolons in the URI — `Uri.parse("smsto:num1;num2")`. The `ACTION_VIEW` with `sms:phoneNumber` URI also works similarly. Always wrap `startActivity()` with intent resolution checking using `intent.resolveActivity(getPackageManager()) != null` to handle devices where no SMS app is installed. This Intent approach is also used to integrate with **WhatsApp** and other messaging apps using their specific URI schemes. The intent-based approach respects user privacy and gives users control over their messages — it is the recommended approach for most apps that only occasionally need to send SMS.

🔹 Receiving SMS Messages

To **receive incoming SMS messages**, your app must declare the `RECEIVE_SMS` permission in `AndroidManifest.xml` and also request it at runtime on API 23+. You register a **BroadcastReceiver** with the `android.provider.Telephony.SMS_RECEIVED` action. In the receiver's `onReceive()` method, retrieve the SMS data from the intent using `Telephony.Sms.Intents.getMessagesFromIntent(intent)` which returns an array of

`SmsMessage` objects. From each `SmsMessage`, retrieve the sender's phone number with `getOriginatingAddress()` and the message body with `getMessageBody()`. Long messages may be split across multiple `SmsMessage` objects — concatenate their bodies. The receiver must be declared in `AndroidManifest.xml` with the appropriate intent-filter, or registered dynamically in code. **Note:** Since Android 8.0 (API 26), background broadcast receivers for implicit broadcasts are restricted — SMS_RECEIVED is an exception and still works. Your app must be the **default SMS app** to reliably receive all SMS on Android 4.4+ — otherwise it receives SMS in a read-only capacity.

🔹 Caveats and Warnings for SMS

Working with SMS in Android comes with several important limitations and considerations that developers must understand. **Permission requirements** are significant — `SEND_SMS` and `RECEIVE_SMS` are dangerous permissions that users must explicitly grant at runtime on API 23+, and many users are reluctant to grant these sensitive permissions. **Default SMS app requirements** — since Android 4.4 (KitKat), only the default SMS app can write to the SMS database and reliably intercept incoming SMS. Third-party apps receive SMS broadcasts but cannot mark them as read or delete them. **Character limits** — standard SMS supports 160 characters in GSM-7 encoding; Unicode messages (containing emoji, non-Latin characters) are limited to 70 characters per part. **Cost awareness** — sending SMS incurs charges on the user's mobile plan — always inform users before sending programmatic SMS. **Rate limiting** — Android may throttle or block apps that send large volumes of SMS. **Privacy concerns** — reading SMS content is highly sensitive and apps should request only the minimum necessary permissions and clearly explain why they need them.

🔹 Sending Email from Android

Android provides multiple approaches for sending emails from your app. The **Intent-based approach** (recommended for most cases) uses `Intent.ACTION_SEND` or `Intent.ACTION_SENDTO` to open the user's default email client pre-filled with recipient, subject, and body — requiring no special permissions. `ACTION_SENDTO` with `mailto:` URI is preferred as it specifically targets email apps. Multiple recipients are supported by passing a String array to `putExtra(Intent.EXTRA_EMAIL, new String[]{"email1", "email2"})`. CC and BCC are set with `EXTRA_CC` and `EXTRA_BCC`. Attachments are added with `EXTRA_STREAM` using a content URI. For **programmatic email sending** without user interaction (e.g., automated error reports), you need a third-party library like **JavaMail API** or use a backend service/API — Android does not have built-in SMTP client capabilities. Services like **SendGrid**, **Firebase Extensions**, or your own backend API are the professional approaches for server-side email sending triggered by Android apps.

```java
// Intent-based email (opens email app)
Intent emailIntent = new Intent(Intent.ACTION_SENDTO);
emailIntent.setData(Uri.parse("mailto:recipient@example.com"));
emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Hello from Kashif's App");
emailIntent.putExtra(Intent.EXTRA_TEXT, "This is the email body.");

if(emailIntent.resolveActivity(getPackageManager()) != null) {
    startActivity(emailIntent);
} else {
    Toast.makeText(this, "No email app found!", Toast.LENGTH_SHORT).show();
}
```
