1. **What are the advantages of Android operating system?**
   Android's biggest strengths are openness, flexibility, and a huge ecosystem. It's open-source (AOSP), runs on many device types, and gives manufacturers and developers deep control over the platform; combined with Google Play and other app stores it delivers a huge user base and fast innovation. For developers this means abundant libraries, tools (Android Studio, Gradle, Jetpack), and multiple distribution channels; for users it means device and UI choice, wide app availability, and frequent hardware variety.

- **Open source (AOSP):** Source code available; vendors can customize.
- **Large market share & device variety:** Phones, tablets, TVs, wearables, cars, IoT.
- **Rich app ecosystem:** Google Play + third-party stores; many libraries and frameworks.
- **Flexible UI & customization:** Launchers, themes, OEM customizations, widgets.
- **Powerful developer tooling:** Android Studio, emulator, profilers, Jetpack libraries.
- **Multiple languages & runtimes:** Kotlin, Java, C/C++ (NDK), with ART performing ahead-of-time compilation.
- **Integration with Google services:** Maps, Firebase, Play services (optional).
- **Cost-effective hardware choices:** Wide price range for devices and OEMs.

2. **List and Explain components of Android SDK**
   The Android SDK bundle contains the tools, platforms, and system images you need to build, debug, test and package Android apps. It's organized so developers can install platform versions, emulator images, and command-line tools.

- **SDK Platform(s):** API libraries for a specific Android OS release (android-xx). Use these to compile against particular Android versions.
- **Platform Tools:** Tools that interact with devices: adb (Android Debug Bridge), fastboot. Essential for debugging and device management.
- **Build Tools & Gradle Plug-ins:** aapt, dx/d8, r8 for packaging, dexing and shrinking/obfuscation; Gradle plug-ins to compile and assemble APK/AAB.
- **Android Studio & IDE integration:** The primary IDE (based on IntelliJ) with profilers, layout editor, Lint, refactoring and device manager.
- **System Images:** Emulator system images (Google APIs, Google Play, AOSP images) for running AVDs.
- **SDK Tools:** Legacy and extra utilities, e.g., emulator binary, traceview, monkeyrunner (some deprecated).
- **Support/Jetpack Libraries:** AndroidX libraries (AppCompat, Lifecycle, Room, WorkManager, Navigation, etc.) for modern app patterns.
- **Emulator & Device Profiles:** AVD manager and hardware profiles to emulate different devices and configurations.

- **Sample Projects & Documentation:** Example apps and API docs included or available online.

3. **What is the use of Content Provider in Android?**
   A Content Provider is the standardized Android component to encapsulate and expose structured app data to other apps (or to your own app components) via a ContentResolver and URIs. Content providers implement CRUD semantics and permission checks, providing a consistent API for data access (e.g., contacts, media). They're especially important when multiple apps need to share a common store safely.

- **Primary purpose:** Share data across apps with a well-defined URI-based API.
- **URI + authority:** Each provider registers an authority and exposes resources via content:// URIs.
- **CRUD operations:** query(), insert(), update(), delete() methods backed by SQLite, files, etc.
- **ContentResolver:** Clients use ContentResolver to perform operations (no direct DB access).
- **MIME types:** Providers return MIME types for URIs to identify data shape.
- **Permissions & Grants:** You can restrict access with android:readPermission/writePermission or grant temporary URI permissions.
- **Examples:** ContactsProvider, MediaStore, Calendar; custom providers for shared app data.

Small conceptual example (Kotlin):

```
// Querying contacts via ContentResolver
val cursor = contentResolver.query(
   ContactsContract.Contacts.CONTENT_URI,
   arrayOf(ContactsContract.Contacts._ID, ContactsContract.Contacts.DISPLAY_NAME),
   null, null, null
)
```

4. **Briefly explain: Shared Preferences**
   SharedPreferences is a lightweight key-value storage mechanism for primitive data types (strings, ints, booleans, floats, long). It's ideal for app settings, flags, or small bits of state that must persist across launches. For more structured or larger data use databases (Room) or DataStore (Jetpack) which is the modern replacement option for preferences.

- **Use cases:** App settings, user preferences, login flags, small state.
- **Data types:** String, int, long, float, boolean, stringSet.
- **APIs:** getSharedPreferences(name, mode) or PreferenceManager.getDefaultSharedPreferences(context).
- **Writing:** Use edit() → putX() → apply() (async) or commit() (sync). apply() is preferred.

- **Security:** Not encrypted by default — consider EncryptedSharedPreferences (Jetpack Security) for sensitive data.
- **Modern alternative:** Jetpack **DataStore** (Preferences & Proto) — safer and coroutine friendly.

Kotlin example:

```
val prefs = context.getSharedPreferences("settings", Context.MODE_PRIVATE)

prefs.edit().putBoolean("isFirstRun", false).apply()

val isFirstRun = prefs.getBoolean("isFirstRun", true)
```

5. **How many ways data stored in Android?**
   Android supports multiple persistent storage options. The right choice depends on size, structure, sharing needs, performance, and security. Below are the main storage mechanisms with brief notes.

- **SharedPreferences / DataStore:** Key-value pairs for small data (settings). DataStore (Preferences or Proto) is the modern coroutine-friendly alternative.
- **Internal Storage (Files):** Private files stored in app-specific directories (filesDir, cacheDir) — accessible only to your app.
- **External Storage / MediaStore:** For user-visible files (photos, downloads). Since Android 10+ use **scoped storage** and MediaStore APIs.
- **SQLite Database / Room:** Relational local DB. Room is the recommended abstraction over SQLite (type-safety, compile-time checks, LiveData/Flow support).
- **Content Providers:** Expose and/or share structured data with other apps.
- **Network / Cloud Storage:** Firebase Realtime DB, Cloud Firestore, REST APIs — for remote/synced data.
- **KeyStore / AccountManager:** For storing cryptographic keys and user credentials securely.
- **Cache & Temporary Storage:** cacheDir, externalCacheDir for transient data that can be removed by the system.

6. **List out different layouts available in Android and explain any one in detail**
   Android provides multiple layout containers to position UI elements. Common ones: LinearLayout, RelativeLayout (older), FrameLayout, ConstraintLayout, TableLayout, GridLayout, CoordinatorLayout, ScrollView, etc. *ConstraintLayout* is the modern, recommended layout for complex screens because it offers powerful constraints, better performance and flatter view hierarchies.

- **LinearLayout:** Arranges children in a single row or column; supports weight to distribute space.
- **FrameLayout:** Simple placeholder — stacks children; good for single-child layouts or overlays.

- **RelativeLayout:** Position children relative to sibling or parent positions (older pattern).
- **ConstraintLayout (detailed):**
    - **Paragraph:** ConstraintLayout lets you create large, complex layouts with a flat view hierarchy by positioning widgets using constraints instead of nesting multiple layouts. You place each view by creating constraints to other views or to the parent (e.g., top-toTopOf, start-toEndOf). It supports chains, barriers, guidelines, groups, and percent positioning which makes responsive UIs easier and often more performant than deep nested layouts. The Android Studio layout editor has a visual constraint editor that helps create constraints visually.
    - **Key features:** Anchors/constraints (start/end/top/bottom), chains (spread, packed, weighted), guidelines (vertical/horizontal), barriers (dynamic helper for groups), ConstraintSet for runtime changes, support for aspect ratio and percent.
    - **When to use:** Complex or responsive UIs where nested LinearLayouts would be inefficient.
    - **Short XML example:**

```
<androidx.constraintlayout.widget.ConstraintLayout ...>
 <TextView
  android:id="@+id/title"
  android:layout_width="0dp"
  android:layout_height="wrap_content"
  app:layout_constraintStart_toStartOf="parent"
  app:layout_constraintEnd_toEndOf="parent"
  app:layout_constraintTop_toTopOf="parent"/>
 <Button
  android:id="@+id/btn"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  app:layout_constraintTop_toBottomOf="@id/title"
  app:layout_constraintEnd_toEndOf="parent"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

7. **Explain Checkbox**
   A CheckBox is a CompoundButton UI widget that represents a boolean choice: checked/unchecked. Checkboxes allow one or more selections from a set (in contrast to radio buttons where only one choice is allowed in a group). They are accessible, standard across Android, and can be styled.

- **Behavior:** Toggle on/off. User-driven binary state.
- **Use case:** Multi-select lists (e.g., interests, filters).
- **API:** isChecked, setChecked(boolean), setOnCheckedChangeListener(CompoundButton.OnCheckedChangeListener).
- **Accessibility:** Provide contentDescription if needed; group related checkboxes with proper labels.
- **Example (Kotlin):**

```
checkBox.setOnCheckedChangeListener { _, isChecked ->
   if (isChecked) doSomething() else undoSomething()
}
```

Visuals: You can use android:buttonTint or custom drawable to change appearance.

8. **What is Intent? Explain with suitable example**
   An Intent is a messaging object used to request an action from another app component. It's how Activities, Services, and BroadcastReceivers are started and how data is passed between components. Intents are either **explicit** (targeting a specific component/class) or **implicit** (declaring a general action that the system resolves to an appropriate component).

- **Explicit Intent:** Directly specifies the component (class) to start. Used within the same app typically.
- **Implicit Intent:** Declares an action (e.g., ACTION_VIEW), data and categories; the system resolves the best component(s) that can handle it.
- **Extras:** Key-value pairs attached to an Intent for passing data (putExtra, getStringExtra).
- **Flags:** Control new task, clearing tasks, singleTop behavior (FLAG_ACTIVITY_NEW_TASK, etc.).
- **PendingIntent:** A wrapper for an Intent that allows another app (e.g., AlarmManager, NotificationManager) to execute it on your behalf.
- **Result APIs:** startActivityForResult is deprecated; use Activity Result APIs (registerForActivityResult) for getting results.

Examples (Kotlin):

- *Explicit* — start DetailActivity:

```
val intent = Intent(this, DetailActivity::class.java)
intent.putExtra("itemId", 123)
startActivity(intent)
```

- *Implicit* — open a web URL:

```
val intent = Intent(Intent.ACTION_VIEW, Uri.parse("https://example.com"))
```

startActivity(intent)

9. **Explain Android Virtual Devices (AVD)**
   An Android Virtual Device (AVD) is an emulator configuration which simulates a physical Android device on your development machine. AVDs let you test apps on different Android versions, screen sizes, densities, CPU architectures and system images (with/without Google Play). They're essential for reproducing device-specific behaviors without needing real hardware.

- **Components:** Hardware profile, system image, memory/SD card settings, skin, and emulator options (GPU, network).
- **System images:** Google Play images (allow Play services), Google APIs images, and AOSP images.
- **Boot modes:** Cold boot vs quick-boot snapshot (faster start).
- **Performance:** Use hardware accelerated virtualization (Intel HAXM, Hypervisor Framework) and GPU acceleration for better performance.
- **Testing benefits:** Multiple API levels, different locales, screen sizes, camera, GPS simulation, network throttling.
- **Limitations:** Some hardware features (e.g., exact sensors, chipset behavior) might differ from real devices; always verify on actual hardware before release.

10. **Explain Fragment Life Cycle in Detail**
    A Fragment is a reusable portion of UI and behavior that lives inside an Activity. Unlike Activities, a Fragment has two related lifecycles: the **fragment lifecycle** (attachment to activity, create/destroy) and the **view lifecycle** (create/destroy the fragment's view). Proper handling ensures resources are allocated and released at correct times and avoids memory leaks (e.g., cleaning view bindings in onDestroyView).

Lifecycle flow (main callbacks & when to use them):

- onAttach(context): Fragment attached to Activity — safe to access context but not activity views.
- onCreate(savedInstanceState): Initialize non-view state (variables, retained data). Don't touch view hierarchy.
- onCreateView(inflater, container, savedInstanceState): Inflate and return the fragment's view. Keep view creation here.
- onViewCreated(view, savedInstanceState): View setup (bind views, set adapters, listeners). Recommended place to access view objects.
- onStart(): Fragment becomes visible.
- onResume(): Fragment is in foreground and user can interact.
- onPause(): Persist changes, stop animations, release exclusive resources.

- onStop(): Fragment no longer visible.
- onDestroyView(): Clean up references to the view (e.g., set binding = null).
- onDestroy(): Final cleanup (non-view resources).
- onDetach(): Fragment detached from Activity.

Important notes & best practices:

- Prefer onViewCreated for view logic; clear view binding in onDestroyView to prevent leaks.
- For retaining UI-related data across config changes use ViewModel (rather than setRetainInstance, which is deprecated).
- Lifecycle-aware components (LiveData, LifecycleScope, repeatOnLifecycle) help avoid memory leaks.
- Fragment transactions (add/replace/remove) affect lifecycle transitions; addToBackStack() influences their destroy/resume behavior.

ASCII-ish simplified flow:

onAttach → onCreate → onCreateView → onViewCreated → onStart → onResume

(onPause → onStop → onDestroyView → onDestroy → onDetach)

---

11. **Explain the Android architecture in detail**
    Android uses a layered architecture that isolates hardware and platform capabilities from app-level code. This modular, layered design improves portability, security and developer productivity. The typical layers (bottom-up):

- **Linux Kernel:** Foundation for process management, drivers (display, camera, Bluetooth, Wi-Fi), power management, security (SELinux), and networking. Android uses a modified Linux kernel providing hardware abstraction and low-level services.
- **Hardware Abstraction Layer (HAL):** C interfaces that allow the Android platform to communicate with device-specific hardware implementations. HAL modules let the system use hardware without depending on vendor-specific drivers directly.
- **Native Libraries & Android Runtime (ART):**
    - **Native Libraries:** C/C++ libraries like libc (Bionic), libsqlite, OpenGL ES, libwebp, media codecs, WebKit/Chromium components. Apps and system services can use native code via NDK.
    - **Android Runtime (ART):** Runs app bytecode compiled to ART format; provides core libraries (java.* and android.*). ART replaces the older Dalvik VM and uses ahead-of-time / just-in-time compilation for performance and memory efficiency.
- **Framework / Application Framework:** Java/Kotlin APIs and managers that app developers use: ActivityManager, WindowManager, ContentProviders, NotificationManager, LocationManager, PackageManager, etc. Provides high-level building blocks for apps.

- **Applications:** End-user apps and system apps (home, phone, contacts). Each app runs in its own process, sandboxed with unique UID, limited access to other apps unless explicitly shared (ContentProvider, Intents, permissions).

Key architecture concepts:

- **Sandboxing & permissions:** Apps run in isolated sandboxes; permissions guard sensitive APIs.
- **Binder IPC:** Core IPC mechanism (Binder) enables fast, secure communication between processes.
- **APK packaging:** Apps packaged as APK/AAB (ZIP-based), with manifest declaring components and permissions.
- **Component model:** Activities, Services, BroadcastReceivers, ContentProviders — each with well-defined responsibilities.
- **Jetpack & AndroidX:** Official libraries for modern app architecture (Lifecycle, Room, WorkManager, Navigation, Compose UI).
- **Modern app architecture:** Encourage separation of concerns — UI layer (Activity/Fragment/Compose), ViewModel for state, Repository for data, Room/Network for persistence, WorkManager for background tasks.

12. **How to use Spinner in android application**
    Spinner is a dropdown selector widget for choosing one item from a list. It requires an Adapter (commonly ArrayAdapter or a custom adapter) to supply the items and an OnItemSelectedListener to receive selection events. There are two display modes: dropdown and dialog.

- **Basic steps:**
    1. Add Spinner to layout XML.
    2. Prepare data source (array or list).
    3. Create an ArrayAdapter (or custom RecyclerView-based spinner if complex).
    4. Set adapter on spinner: spinner.adapter = adapter.
    5. Handle selection via spinner.onItemSelectedListener.
- **Example (Kotlin):**

```
val spinner: Spinner = findViewById(R.id.spinner)

val items = listOf("India", "USA", "UK")

val adapter = ArrayAdapter(this, android.R.layout.simple_spinner_item, items)

adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)

spinner.adapter = adapter
```

```
spinner.onItemSelectedListener = object : AdapterView.OnItemSelectedListener {
   override fun onItemSelected(parent: AdapterView<*>, view: View?, pos: Int, id: Long) {
      val selected = parent.getItemAtPosition(pos) as String
   }
   override fun onNothingSelected(parent: AdapterView<*>) { /* no-op */ }
}
```

- **Customization:** Provide custom layout for rows or use themes; spinner mode android:spinnerMode="dropdown|dialog".
- **Accessibility:** Provide content descriptions and meaningful labels.

13. **Explain the significance of Software Development Life Cycle (SDLC)**
    SDLC is a structured process for planning, developing, testing, delivering and maintaining software. It brings predictability, quality checks, stakeholder alignment and risk management to software projects. SDLC models (Waterfall, Agile, Spiral, DevOps) differ in iteration, but all aim to improve quality, on-time delivery, and manage costs.

- **Phases & significance:**
    - **Requirements / Planning:** Gather functional and non-functional requirements; reduces scope creep.
    - **Analysis / Feasibility:** Technical feasibility, risk analysis, and cost estimation.
    - **Design:** Architecture, data models, UI/UX — sets blueprint for implementation.
    - **Implementation (Coding):** Actual development against designs and standards.
    - **Testing / QA:** Unit, integration, system and acceptance testing — ensures correctness and reliability.
    - **Deployment:** Release to production with roll-out strategy and rollback plan.
    - **Maintenance:** Bug fixes, updates, and feature enhancements.
- **Benefits:** Improved quality, predictable timelines, better resource planning, risk mitigation, stakeholder visibility, easier maintenance.
- **Modern practices:** Agile/CI-CD/DevOps emphasize continuous integration, fast feedback loops, and incremental delivery.

14. **What is Activity? Draw and explain Activity lifecycle in detail**
    An **Activity** represents a single focused screen with which the user interacts. Activities manage UI, handle user events, and are the primary entry points for performing tasks. The Activity lifecycle is crucial because Android may create, pause, stop, destroy or re-create Activities in response to user actions or system conditions; correctly handling lifecycle callbacks ensures state preservation and resource management.

Activity lifecycle callbacks & explanation:

- onCreate(savedInstanceState): Initialize UI, restore state from savedInstanceState bundle, set content view. Heavy setup here but avoid long blocking work.
- onStart(): Activity is visible but not in foreground interactive state.
- onResume(): Activity is in foreground and interactive — start animations, acquire exclusive resources.
- onPause(): Called when another activity comes into foreground (partially obscured). Save transient state and stop animations; quick operations only.
- onStop(): Activity no longer visible — release resources (e.g., camera) and save persistent data.
- onRestart(): Activity restarting after stop — usually leads to onStart.
- onDestroy(): Final cleanup before process reclaims the Activity; could be called for configuration changes or final finish.
- onSaveInstanceState(Bundle): Called before possible destruction to save UI state (transient UI state) — restore in onCreate or onRestoreInstanceState.

Key practices:

- Use onSaveInstanceState to save UI state for process death (short-lived).
- Persist durable data in onPause/onStop or via ViewModel + persistent store.
- Avoid long-running work on the main thread; use background threads, coroutines, or WorkManager.
- For receiving results from another Activity, use Activity Result APIs (registerForActivityResult) (recommended over deprecated startActivityForResult).
- Understand process death: the OS can kill your process when the Activity is stopped; restore state from saved bundles rather than relying on background fields.

15. **Explain Access web data with JSON**
    JSON (JavaScript Object Notation) is the most common format for REST APIs. To access web data: perform HTTP requests, parse the JSON into model objects, and update the UI. Use robust HTTP clients (Retrofit + OkHttp, Volley) and modern async mechanisms (Kotlin coroutines, Flow, LiveData) to keep the UI responsive.

- **Steps:**

    1. **Choose HTTP client:** Retrofit (with OkHttp) is the most common for REST; Volley for simpler cases.
    2. **Define models:** Data classes matching JSON structure (use Gson, Moshi, or Kotlinx.serialization).
    3. **Define API interface:** For Retrofit, annotate endpoints (@GET, @POST).

4. **Execute request off main thread:** Coroutines, Dispatchers.IO, or Retrofit suspend functions.
5. **Parse JSON:** Automatic with converters (Gson/Moshi) or manually with JSONObject.
6. **Update UI on main thread:** Post parsed models to UI via LiveData/StateFlow/ViewModel.
7. **Error handling & retries:** Handle network errors, timeouts, parse errors; show fallback UI.

Retrofit + Kotlin coroutine example:

```
// Data class
data class Post(val id: Int, val title: String, val body: String)


// Retrofit interface
interface ApiService {
 @GET("posts")
 suspend fun getPosts(): List<Post>
}


// Usage inside ViewModel (with coroutine)
viewModelScope.launch {
 try {
  val posts = apiService.getPosts()
  _uiState.value = UiState.Success(posts)
 } catch (e: Exception) {
  _uiState.value = UiState.Error(e)
 }
}
```

16. **Briefly explain AsyncTaskLoader**
    AsyncTaskLoader is a Loader that performs asynchronous loading of data, typically from a database or content provider, without blocking the UI. It's lifecycle-aware through the LoaderManager and was commonly used before architecture components matured. Loaders automatically reconnect to the last loader's results after configuration changes. However, modern Android recommends using ViewModel + coroutines/LiveData or WorkManager for most background tasks.

- **How it worked:** Subclass AsyncTaskLoader<T>, implement loadInBackground() and call deliverResult() or forceLoad(); use LoaderManager to initialize and manage it.
- **Advantages:** Lifecycle-aware, automatically reconnects after config changes, runs off main thread.
- **Drawbacks / Current status:** Loaders and AsyncTask are considered legacy; AsyncTask is deprecated. Prefer **ViewModel + LiveData/StateFlow** and Kotlin coroutines, or WorkManager for deferrable background work.
- **Example skeleton:**
- class MyLoader(context: Context) : AsyncTaskLoader<List<Item>>(context) {
- override fun onStartLoading() { forceLoad() }
- override fun loadInBackground(): List<Item>? {
- // Blocking I/O; return result
- }
- }

17. **Write the steps for CRUD operation to use Firebase database in android application**
Below steps assume Firebase Realtime Database (Firestore has similar but different APIs). Use Firebase console to configure the project and google-services.json in app. Use Kotlin coroutines / listeners appropriately.

- **Steps:**
    1. **Create Firebase project:** In Firebase console create project and add Android app package name.
    2. **Add google-services.json:** Download from console and place in app/.
    3. **Update Gradle:** Add classpath 'com.google.gms:google-services:... and apply com.google.gms.google-services; add implementation 'com.google.firebase:firebase-database-ktx:...' (or Firestore).
    4. **Initialize (optional):** FirebaseApp.initializeApp(context) — typically auto-initialized.
    5. **Get DatabaseReference:**
    6. val db = Firebase.database.reference // Realtime DB
    7. val usersRef = db.child("users")
    8. **Create (C):** Use push() for auto-id or set a key:
    9. val id = usersRef.push().key
    10. val user = User("Alice", 25)
    11. usersRef.child(id!!).setValue(user)

12. **Read (R):** Attach a listener:
13. usersRef.addValueEventListener(object : ValueEventListener {
14. override fun onDataChange(snapshot: DataSnapshot) { /* parse snapshot */ }
15. override fun onCancelled(error: DatabaseError) { /* handle */ }
16. })
17. **Update (U):** updateChildren() or setValue() on a child:
18. usersRef.child(userId).updateChildren(mapOf("age" to 26))
19. **Delete (D):** removeValue():
20. usersRef.child(userId).removeValue()
21. **Security rules & auth:** Define DB rules in Firebase console; use Firebase Auth to control per-user access.
22. **Offline & sync:** Realtime DB supports persistence (FirebaseDatabase.getInstance().setPersistenceEnabled(true)), and Firestore has similar features.

18. **Describe shared preference with example**
SharedPreferences stores simple key-value pairs in XML files. It's synchronous for commit() and asynchronous for apply(). Use EncryptedSharedPreferences when storing secrets (requires Jetpack Security). For structured or larger datasets, prefer Room or DataStore.

- **API summary:** getSharedPreferences(name, mode) → edit() → putX() → apply()/commit().
- **apply vs commit:** apply() writes asynchronously (no return result), commit() writes synchronously and returns a boolean. Prefer apply() for UI thread.
- **Scopes:** MODE_PRIVATE is default (private to app).
- **EncryptedSharedPreferences:** Wraps prefs in AES encryption (Jetpack Security) — recommended for sensitive items like tokens.

Kotlin example (simple and Encrypted):

```
// Simple
val prefs = context.getSharedPreferences("app_prefs", Context.MODE_PRIVATE)
prefs.edit().putString("userName", "Raj").apply()
val name = prefs.getString("userName", "")

// Encrypted (requires dependency and MasterKey)
val masterKey =
MasterKey.Builder(context).setKeyScheme(MasterKey.KeyScheme.AES256_GCM).build()
```

```
val encPrefs = EncryptedSharedPreferences.create(
  context, "secret_prefs", masterKey,
  EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256_SIV,
  EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256_GCM
)
encPrefs.edit().putString("token", "s3cr3t").apply()
```

19. **List out UI Components in Android Application. Explain three in brief.**
    Android provides many built-in UI widgets for building user interfaces. Common UI components include TextView, EditText, Button, ImageView, CheckBox, RadioButton, Switch, Spinner, RecyclerView, ListView, WebView, ProgressBar, SeekBar, CardView, Toolbar/AppBar, ConstraintLayout, and many Material Components. Below I explain three widely used ones.

- **TextView:** Displays text to the user. Supports styling (size, color, font), spans for rich text, and autoLink features. Use TextView for read-only text, and TextSwitcher/Marquee for special effects.
- **EditText:** User-editable text field derived from TextView. Supports input types (number, textEmailAddress), hint text, input filters, IME options, and password transformation. Typically used inside TextInputLayout (Material) for better UX.
- **RecyclerView:** A flexible, efficient list/grid widget that reuses ViewHolder objects for scrolling large datasets. Unlike ListView, RecyclerView decouples layout (via LayoutManager), decorations (via ItemDecoration), and data diffing (via ListAdapter/DiffUtil). Recommended for any scrollable collection of items.

20. **Explain Recycler View and Card View in android.**
    RecyclerView is the modern, flexible replacement for ListView/GridView providing efficient view recycling and many extension points. CardView is a view group that provides a consistent card-like container with elevation, rounded corners, and content padding — commonly used as item container inside RecyclerView to achieve Material design cards.

RecyclerView (detailed):

- **Core parts:** RecyclerView widget, Adapter (binds data to view holders), ViewHolder (holds references to views), LayoutManager (positions items: LinearLayoutManager, GridLayoutManager, StaggeredGridLayoutManager).
- **Performance:** Recycles views via ViewHolder pattern; DiffUtil and ListAdapter help efficient list updates.
- **Extensibility:** Supports ItemDecoration (dividers), ItemAnimator (animations), and custom layout managers.
- **Best practices:** Use ListAdapter + DiffUtil, keep ViewHolder lightweight, avoid findViewById each bind (use view binding), offload heavy work from bind (use Glide/Picasso for images).

CardView (detailed):

- **Purpose:** Simple container that provides rounded corners and elevation (shadow), matching Material card metaphor.
- **Usage:** Typically wrap item content inside a CardView inside an item layout used by a RecyclerView adapter.
- **Attributes:** cardCornerRadius, cardElevation, cardUseCompatPadding. Works well with Material components.

Small RecyclerView item example (Kotlin adapter + item layout using CardView):

item_layout.xml

```
<androidx.cardview.widget.CardView ... >
  <LinearLayout ... >
    <ImageView android:id="@+id/iv"/>
    <TextView android:id="@+id/title"/>
  </LinearLayout>
</androidx.cardview.widget.CardView>
```

Kotlin Adapter skeleton:

```
class MyAdapter(private val items: List<Item>) : RecyclerView.Adapter<MyAdapter.VH>() {
  class VH(val binding: ItemLayoutBinding) : RecyclerView.ViewHolder(binding.root)
  override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) =
     VH(ItemLayoutBinding.inflate(LayoutInflater.from(parent.context), parent, false))
  override fun onBindViewHolder(holder: VH, position: Int) {
     val item = items[position]
     holder.binding.title.text = item.title
     // load image with Glide/Picasso
  }
  override fun getItemCount() = items.size
}
```