CBCS SCHEME

USN 1CR22EC235

BIS654C

Sixth Semester B.E./B.Tech. Degree Examination, June/July 2025

Mobile Application Development

Time: 3 hrs. Max. Marks: 100

Note: 1. Answer any FIVE full questions, choosing ONE full question from each module.
2. M : Marks , L: Bloom's level , C: Course outcomes.

			M	L	C
		Module – 1			
Q.1	a.	Explain Android Architecture.	10	L2	CO1
	b.	List any Ten features of Android.	10	L2	CO1
		OR			
Q.2	a.	Describe Android Development Tools (ADT).	10	L2	CO2
	b.	Explain Dalvik Virtual Machine.	10	L2	CO2
		Module – 2			
Q.3	a.	Explain Directory Structure.	10	L2	CO3
	b.	Describe Fundamental UI Design.	10	L2	CO3
		OR			
Q.4	a.	Write a program to a relative layout.	10	L2	CO3
	b.	Explain Ten Attributes.	10	L2	CO3
		Module – 3			
Q.5	a.	Write a program of Time Picker with am pm.	10	L2	CO3
	b.	List properties of list view.	10	L2	CO3
		OR			
Q.6	a.	List properties of progress BOQ.	10	L2	CO3
	b.	Write a program for Toggle Button.	10	L2	CO3
		Module – 4			
Q.7	a.	Explain Activity life cycle with diagram.	10	L2	CO4
	b.	Explain with neat diagram, service life cycle.	10	L2	CO4
		OR			
Q.8	a.	Explain types of sensors.	10	L2	CO4
	b.	With diagram, explain multimedia framework architecture.	10	L2	CO4
		Module – 5			
Q.9	a.	Write a program getting location Data.	10	L2	CO5
	b.	Write a program for zooming In or Out of the Map.	10	L2	CO5
		OR			
Q.10	a.	Explain transactions with example.	10	L2	CO5
	b.	Explain how can we create the tables?	10	L2	CO5

* * * * *

Module–1

Q1(a). Explain Android Architecture.

Answer:
Android is a **layered software stack** that lets apps run safely and efficiently on many kinds of hardware.

1) Linux Kernel (Hardware Abstraction & Core Services)

- **Role:** Foundation of the stack; provides process scheduling, memory mgmt, file system, networking, power mgmt.
- **Drivers:** Display, camera, audio, binder (IPC), Wi-Fi, Bluetooth, sensors.
- **Why it matters:** Same kernel APIs across devices ⇒ portability; robust security primitives (UIDs, process isolation).

2) Native C/C++ Libraries

- **Graphics:** Skia (2D), OpenGL ES/Vulkan (3D).
- **Media:** libstagefright/MediaCodec (codecs), OpenMAX AL.
- **Database:** SQLite (embedded relational DB).
- **Web:** WebKit/Blink components for rendering.
- **Security:** SSL, zlib, etc.
- **Why:** High-performance components that apps access through the framework.

3) Android Runtime (ART; earlier Dalvik)

- **Core libraries:** Java/Kotlin standard libs + Android-specific APIs.
- **ART:** Ahead-of-Time (AOT) compilation + JIT + profile-guided optimization; garbage collection; better startup than Dalvik.
- **Per-app isolation:** Every app runs in its own process with a unique Linux UID (sandbox).

4) Application Framework (Java/Kotlin APIs)
Managers that apps use:

- **Activity/Fragment Manager:** screen navigation & back stack.
- **Window & View System:** draw UI and handle events.
- **Package Manager:** install/update apps & permissions.

- **Content Providers:** structured data sharing (e.g., Contacts).
- **Resource Manager:** load layouts, strings, drawables for localization.
- **Location/Notification/Sensor/Telephony managers** etc.
- **Why:** High-level, device-independent APIs = faster development.

5) Applications (System + User Apps)

- Preinstalled (Dialer, Settings) + third-party apps.
- **Intent system** lets apps request actions from other apps securely (e.g., share image, open URL).

Q1(b). List any Ten features of Android *(explain each briefly).*

Answer:

1. **Open Source (AOSP):** OEMs can customize; large community support.
2. **Application Sandbox:** Per-app UID, process isolation, SELinux—limits damage of compromised apps.
3. **Rich Component Model:** Activities, Services, Broadcast Receivers, Content Providers ⇒ composable apps.
4. **Intent-based IPC:** Decouples requesters and providers (e.g., "share" works with any app that can share).
5. **Resource System:** Alternate resources for screen sizes, locales, night mode ⇒ single APK adapts broadly.
6. **Notifications & Foreground Services:** Engage users outside the app; channels, actions, priorities.
7. **Connectivity Stack:** Wi-Fi, 5G, Bluetooth, NFC, USB, WebRTC—easy network programming.
8. **Location & Sensors:** GPS + fused provider, accelerometer, gyroscope, magnetometer, proximity, light ⇒ AR, fitness, maps.
9. **Graphics & Media:** Hardware-accelerated UI, OpenGL/Vulkan, high-quality audio/video codecs.
10. **App Distribution:** Google Play & other stores; in-app updates, billing, crash reporting ecosystem.

Q2(a). Describe Android Development Tools (ADT).

Answer :
Historically, **ADT** was an Eclipse plugin; today **Android Studio (IntelliJ-based)** is the official IDE. Exams often say ADT—explain features common to both.

1. **Project Wizards & Gradle Build System:** Create templates (Empty Activity, Master/Detail). Gradle handles compilation, resource merging, AAPT2 packaging, signing, product flavors, build types (debug/release).
2. **Layout & Resource Editors:** Drag-and-drop **Layout Editor**, ConstraintLayout tools, preview for multiple devices/locales; vector/drawable editors; translation editor.
3. **Emulator & Device Manager:** Virtual devices with different CPU/ABI, screen sizes, sensors, camera, location, network throttling; snapshot/quick boot.
4. **Debugging (Logcat, Breakpoints, Watches):** Inspect variables, step through code, filters for logs, structured logging.
5. **Profilers:** CPU, Memory, Network profilers; energy profiler to find battery drains; heap dumps & leak detection.
6. **APK Analyzer & App Inspection:** Inspect APK size, dex methods, resources, manifest; database inspector; network inspector.
7. **Testing Support:** JUnit, Instrumentation/Espresso UI tests, Robolectric; test orchestrator; coverage reports.
8. **Version Control Integration:** Git built-in (branching, merging, blame).
9. **Dependency Manager:** Gradle/Maven pulls libraries (material, lifecycle, maps, retrofit).
10. **Packaging & Signing:** Keystore mgmt, app bundles (AAB), split APKs, proguard/R8 minification & obfuscation.

Q2(b). Explain Dalvik Virtual Machine.

Answer:
Dalvik (pre-ART) executed Android apps until Lollipop.

- **Execution Format:** Converts JVM class files to a single **.dex** (Dalvik Executable) with constant-pool sharing ⇒ reduced size.

- **Register-based VM:** Uses 16-bit instructions operating on registers ⇒ fewer instructions than JVM's stack VM for many ops; optimized for low memory/CPU.
- **Per-Process Instance:** Each app has its own Dalvik instance → isolation + independent GC.
- **Just-In-Time (JIT) Compilation:** Later Dalvik versions used JIT to compile hot code paths at runtime.
- **Garbage Collection:** Stop-the-world GC tuned for mobile footprints.
- **Lifecycle:** Replaced by **ART**, which adds AOT + better GC & profiling, but Dalvik is important historically and for understanding .dex & register model.

Module–2

Q3(a). Explain Directory Structure of an Android Project.

Answer:

```
MyApp/
├─ app/
│  ├─ src/
│  │  ├─ main/
│  │  │  ├─ java/ or kotlin/     ← app source (packages)
│  │  │  ├─ res/                 ← resources
│  │  │  │  ├─ layout/           (XML UIs)
│  │  │  │  ├─ drawable/         (png/svg/xml shapes)
│  │  │  │  ├─ mipmap-*/         (launcher icons)
│  │  │  │  ├─ values/           (strings.xml, colors.xml, dimens.xml, styles.xml)
│  │  │  │  ├─ menu/             (menus)
│  │  │  │  ├─ xml/              (preferences, file_paths)
│  │  │  ├─ AndroidManifest.xml  ← components, permissions, intent-filters
│  │  ├─ test/ & androidTest/    ← unit & instrumented tests
│  ├─ build.gradle (module)      ← deps, minSdk/targetSdk
```

├─ build.gradle (project) ← repositories, plugins

├─ gradle/ & settings.gradle ← build config

- **assets/** (optional): raw bundled files accessible via AssetManager.
- **.gitignore, proguard-rules.pro:** version control & shrinker configuration.
- **Why:** Separation of code vs resources enables localization, theming, and easy reuse.

Q3(b). Describe Fundamental UI Design in Android.

Answer:

1. **Layout Systems:**
    - **ConstraintLayout:** modern, flexible, flat hierarchies; constraints to parent/siblings; chains, guidelines.
    - **LinearLayout:** rows/columns; weights for proportional space.
    - **RelativeLayout/FrameLayout:** relative positioning or stack children; good for overlays.
    - **Why:** choose based on complexity and performance.
2. **View & ViewGroup:**
    - **View:** basic UI element (TextView, Button, ImageView).
    - **ViewGroup:** containers that control child layout (RecyclerView, ScrollView).
    - **Measure–Layout–Draw** pass defines rendering order.
3. **Material Design Principles:**
    - **Hierarchy & Emphasis:** typography scale, color contrast, elevation.
    - **Motion/Feedback:** ripple effects, meaningful transitions.
    - **Adaptive UI:** responsive breakpoints, smallest-width (sw) qualifiers, orientation handling.
4. **State & Interaction:**
    - **Listeners:** onClick, onTouch, onFocusChange.
    - **State lists:** color/drawable selectors for pressed/focused/disabled.

- **Accessibility:** content descriptions, touch target ≥ 48dp.

5. **Theming & Styles:**
   - Define styles once (fonts, paddings); apply via themes (Theme.Material3) to keep UIs consistent and easily dark-mode-ready.
6. **Performance Tips:**
   - Minimize nested layouts, use RecyclerView for long lists, use vector drawables, enable layout inspector to catch overdraw.

Q4(a). Write a program using RelativeLayout.

Answer :

layout/activity_main.xml

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

  <ImageView
    android:id="@+id/avatar"
    android:layout_width="72dp"
    android:layout_height="72dp"
    android:src="@mipmap/ic_launcher"
    android:contentDescription="@string/app_name"
    android:layout_alignParentStart="true"
    android:layout_marginEnd="12dp"/>

  <TextView
    android:id="@+id/name"
```

```
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Rakesh Kumar"
    android:textStyle="bold"
    android:textSize="18sp"
    android:layout_toEndOf="@id/avatar"
    android:layout_alignTop="@id/avatar"/>

  <TextView
    android:id="@+id/subtitle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Senior Product Manager"
    android:layout_below="@id/name"
    android:layout_toEndOf="@id/avatar"/>

  <Button
    android:id="@+id/follow"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Follow"
    android:layout_below="@id/avatar"
    android:layout_alignParentEnd="true"/>
</RelativeLayout>
```

MainActivity.java (or Kotlin)

```
public class MainActivity extends AppCompatActivity {
```

```
 @Override protected void onCreate(Bundle b) {
  super.onCreate(b);
  setContentView(R.layout.activity_main);
  findViewById(R.id.follow).setOnClickListener(v ->
   Toast.makeText(this, "Following!", Toast.LENGTH_SHORT).show());
 }
}
```

Why RelativeLayout here? Easy to place name to the **right of** the avatar and **below/ aligned** constraints without deep nesting.

Q4(b). Explain Ten common Android view attributes *(with effect & example).*

Answer :

1. **android:id** – unique reference to use in Java/Kotlin and for relative rules.
   Ex: @+id/submitBtn.
2. **android:layout_width / android:layout_height** – size specifiers: wrap_content, match_parent, exact dp.
   Why: controls measure phase.
3. **android:layout_margin** – outer spacing; can be per-side (marginStart, marginTop …).
   Ex: avoid overlap, add breathing room.
4. **android:padding** – inner spacing between content and view bounds.
   Ex: TextView padding improves readability.
5. **android:gravity** – how content is placed **inside** the view (e.g., center, end).
   Ex: center text vertically in a button.
6. **android:text / android:hint** – display string and placeholder; typically from resources (@string/...) for localization.
7. **android:src / android:background** – image content vs background styling (colors, shape drawables).
   Tip: use backgroundTint to theme buttons.
8. **android:visibility** – visible, invisible (keeps space), gone (removes from layout).
   Why: dynamic UI changes without re-inflation.

9. **android:orientation** *(for LinearLayout)* – vertical or horizontal. Works with **layout_weight** to distribute space.
10. **android:onClick** – binds a public method in activity to the click event via XML; quick for demos.
    Better: use listeners or data binding in production.

Module–3

Q5(a). Write a program of TimePicker with AM/PM.

Answer :

layout/activity_time.xml

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

 <TimePicker
    android:id="@+id/timePicker"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:is24HourView="false"/>

 <TextView
    android:id="@+id/selected"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:paddingTop="12dp"/>
</LinearLayout>
```

Activity:

```
public class TimeActivity extends AppCompatActivity {
  @Override protected void onCreate(Bundle b) {
    super.onCreate(b);
    setContentView(R.layout.activity_time);

    TimePicker tp = findViewById(R.id.timePicker);
    TextView tv = findViewById(R.id.selected);
    tp.setIs24HourView(false); // AM/PM

    tp.setOnTimeChangedListener((view, hour, minute) -> {
      int displayHour = (hour == 0) ? 12 : (hour > 12 ? hour - 12 : hour);
      String ampm = (hour >= 12) ? "PM" : "AM";
      tv.setText(String.format(Locale.getDefault(), "%02d:%02d %s", displayHour, minute,
ampm));
    });
  }
}
```

Notes: Use android:is24HourView="false" for 12-hour format; convert hour to AM/PM for display.

Q5(b). List properties of ListView *(and how they affect UI).*

Answer :

- **android:divider / android:dividerHeight** – visual separation between rows; improves readability.
- **android:choiceMode** – none, singleChoice, multipleChoice; enables checked states for selection lists.
- **android:fastScrollEnabled** – adds a scrollbar thumb for long lists; UX boost for large datasets.

- **android:listSelector** – drawable used to highlight pressed/selected rows (ripple, color).
- **android:entries** – bind from a string-array resource for simple static lists.
- **android:stackFromBottom** – start list from bottom (e.g., chat history).
- **android:transcriptMode** – auto-scroll when items change; useful for chats/logs.
- **android:scrollbarStyle** – inside overlay vs outside inset; affects aesthetics.
- **android:footerDividersEnabled / headerDividersEnabled** – control extra dividers near header/footer views.
- **android:drawSelectorOnTop** – draw selector above content for visibility.

Adapter Example *(typical usage)*:

```
ListView lv = findViewById(R.id.list);
ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
    android.R.layout.simple_list_item_1,
    Arrays.asList("One","Two","Three"));
lv.setAdapter(adapter);
```

(Note: modern apps prefer RecyclerView for flexibility & performance.)

Q6(a). List properties of ProgressBar *(with meaning & use).*

Answer :

- **style** – ?android:attr/progressBarStyle (small, large), or **horizontal** for determinate bars.
- **android:indeterminate** – true for unknown duration (spinning wheel).
- **android:max** – maximum value for determinate bar (default 100).
- **android:progress** – current completed value.
- **android:secondaryProgress** – buffer state (e.g., media buffering).
- **android:visibility** – show/hide during tasks.
- **android:progressTint / indeterminateTint** – tint colors with theme.

- **android:layout_width/height & layout_margin** – placement and size.
- **android:animateLayoutChanges (parent)** – smooth appearance/disappearance.
- **setProgress(int, boolean) (code)** – animate progress updates.

Determinate example:

```
<ProgressBar
 style="@android:style/Widget.ProgressBar.Horizontal"
 android:id="@+id/pb"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:max="100"
 android:progress="0"/>
pb.setProgress(60, true); // 60% with animation
```

Q6(b). Write a program for ToggleButton.

Answer :

layout:

```
<ToggleButton
 android:id="@+id/toggle"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:textOn="Wi-Fi ON"
 android:textOff="Wi-Fi OFF"/>
<TextView
 android:id="@+id/status"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"/>
```

Activity:

```
public class ToggleActivity extends AppCompatActivity {
  @Override protected void onCreate(Bundle b) {
    super.onCreate(b);
    setContentView(R.layout.activity_toggle);

    ToggleButton t = findViewById(R.id.toggle);
    TextView status = findViewById(R.id.status);

    t.setOnCheckedChangeListener((button, checked) -> {
      status.setText(checked ? "Service enabled" : "Service disabled");
      // doWork(checked); // e.g., enable/disable feature
    });

    // restore state on rotation
    if (b != null) t.setChecked(b.getBoolean("state", false));
  }

  @Override protected void onSaveInstanceState(Bundle out) {
    super.onSaveInstanceState(out);
    out.putBoolean("state", ((ToggleButton)findViewById(R.id.toggle)).isChecked());
  }
}
```

Module–4

Q7(a). Explain Activity Life Cycle with diagram.

Answer :

States & Callbacks:

1. **onCreate()** – 1st time setup: inflate layout, init ViewModel/adapters, restore state from savedInstanceState.
2. **onStart()** – Activity becomes **visible** (UI exists, not yet foreground). Start lightweight UI updates.
3. **onResume()** – Activity in **foreground**; start camera, sensors, animations, listeners.
4. **onPause()** – Partially obscured (dialog/another activity). **Commit unsaved changes**, pause animations, heavy sensors. Keep state quick; this must be fast.
5. **onStop()** – Fully hidden. Release expensive resources (camera/GPS). Persist to DB.
6. **onRestart()** – Coming back from Stopped → **onStart()** called next.
7. **onDestroy()** – Final cleanup; may be called due to finish() or system destroying to reclaim memory.

Diagram (flow):

onCreate → onStart → onResume

↑ ↓

onRestart onPause → onStop → onDestroy

↘ (back to foreground) ↗

Why it matters: Correct placement of code prevents **memory leaks**, **ANR**, and **battery drain**.

Lifecycle logging snippet:

```
@Override protected void onResume(){ super.onResume(); Log.d("LC","resume"); }
```

Q7(b). Explain with neat diagram: Service Life Cycle.

Answer :

Types of services:

- **Started Service:** startService() / startForegroundService() – runs until it stops itself (stopSelf()/stopService()), good for long tasks (music).

- **Bound Service:** bindService() – provides an API to clients (e.g., IPC with Messenger/Binder). Runs as long as at least one client is bound.

Callbacks:

- **onCreate()** – one-time init (threads, notification channel).
- **onStartCommand(Intent, flags, startId)** – for started services; return value (START_STICKY, etc.) controls restart behavior if killed.
- **onBind(Intent)** – return IBinder for clients; called on first bind.
- **onUnbind()/onRebind()** – when clients disconnect/reconnect.
- **onDestroy()** – cleanup resources, stop threads, release wake locks.

Diagram (simplified):

startService() stopSelf()/stopService()

Client ----------> onStartCommand() ----------> onDestroy()

Client ---- bindService() --> onBind() -- use binder API -- unbindService() --> onUnbind() -> onDestroy()

Foreground service: shows a persistent notification (e.g., music playback, location). Required for long-running background tasks from Android O+.

Q8(a). Explain types of sensors in Android.

Answer :

1) Motion Sensors (measure acceleration/rotation):

- **Accelerometer:** linear acceleration including gravity (m/s^2).
- **Gyroscope:** rate of rotation (rad/s).
- **Gravity/Linear Accel:** virtual sensors derived from others.
  Use cases: shake to refresh, step detection, gaming, stabilization.

2) Environmental Sensors (ambient conditions):

- **Light (lux), Proximity (cm), Pressure (hPa), Temperature (°C), Humidity (%).**
  Use cases: auto-brightness, pocket detection, weather apps, altimeter.

3) Position Sensors:

- **Magnetometer:** ambient magnetic field; with accelerometer gives **compass**.
- **GPS (location provider):** satellite location; via Fused Location combines GPS + Wi-Fi + Cell for efficiency.
  Use cases: navigation, geo-fencing, AR alignment.

4) Biometric/Health (device-specific):

- **Fingerprint, face unlock, heart rate** (on wearables).
  Use cases: authentication, wellness.

Access pattern & best practices:

SensorManager sm = (SensorManager)getSystemService(SENSOR_SERVICE);

Sensor accel = sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

sm.registerListener(listener, accel, SensorManager.SENSOR_DELAY_GAME);

- Unregister in onPause() to save battery.
- Use **low-pass/high-pass filters** to separate gravity vs motion.
- Check sensor.getVendor() and getResolution() for quality.

Q8(b). With diagram, explain Multimedia Framework Architecture.

Answer :

High-level pipeline (playback):

App (MediaPlayer/ExoPlayer)

↓

Framework APIs (MediaPlayer, AudioManager, CameraX)

↓

Media Server (MediaCodec, Stagefright pipeline)

↓

HAL (AudioFlinger, Audio HAL / Camera HAL)

↓

Hardware (DSP, codecs, camera sensor, speakers)

Components:

- **Application Layer:** Uses MediaPlayer, AudioTrack, VideoView, or **ExoPlayer** (Google's library) to play/record.
- **Framework APIs:** Manage focus, routing, DRM, subtitles, audio attributes.
- **Native Media (Stagefright/MediaCodec):** Demuxers (MP4, MKV), decoders/encoders (H.264, AAC, Opus), time sync.
- **AudioFlinger:** Mixer for multiple audio streams, applies effects (equalizer).
- **OpenSL ES/AAudio:** Low-latency audio paths for games.
- **Camera stack:** App → Camera2/CameraX → Camera HAL → ISP → encoder.
- **Hardware Acceleration:** Many codecs run on DSP/GPU for battery efficiency.
- **DRM:** MediaDrm/MediaCrypto for protected content.

Why: Layers isolate apps from hardware specifics while delivering **low latency**, **power-efficient** playback/recording.

Module–5

Q9(a). Write a program getting Location Data.

Answer :
Modern approach uses **Fused Location Provider (FLP)** from Google Play services (better accuracy & battery). Include **runtime permissions**.

Manifest:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

Request permission at runtime (Activity):

```
ActivityCompat.requestPermissions(this,
   new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, 100);
```

Code to get updates:

```
public class LocationActivity extends AppCompatActivity {
  private FusedLocationProviderClient fused;
```

```
 private LocationCallback callback;

 @Override protected void onCreate(Bundle b) {
  super.onCreate(b);
  setContentView(R.layout.activity_main);

  fused = LocationServices.getFusedLocationProviderClient(this);

  LocationRequest req = new
LocationRequest.Builder(Priority.PRIORITY_HIGH_ACCURACY, 5000)
        .setMinUpdateIntervalMillis(2000)
        .setWaitForAccurateLocation(true)
        .build();

  callback = new LocationCallback() {
   @Override public void onLocationResult(LocationResult res) {
    if (res == null) return;
    Location loc = res.getLastLocation();
    if (loc != null) {
     String msg = "Lat=" + loc.getLatitude() + "  Lon=" + loc.getLongitude();
     Toast.makeText(LocationActivity.this, msg, Toast.LENGTH_SHORT).show();
    }
   }
  };
  startUpdates(req);
 }
```

```
 private void startUpdates(LocationRequest req) {
  if (ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION)
     != PackageManager.PERMISSION_GRANTED) return;
  fused.requestLocationUpdates(req, callback, Looper.getMainLooper());
 }

 @Override protected void onPause() {
  super.onPause();
  fused.removeLocationUpdates(callback); // save battery
 }
}
```

Notes:

- Choose accuracy priority wisely; for geofences or coarse UI, use PRIORITY_BALANCED_POWER_ACCURACY.
- Always stop updates when not needed.

Q9(b). Write a program for zooming In or Out of the Map.

Answer :

layout:

```
<fragment
 android:id="@+id/map"
 android:name="com.google.android.gms.maps.SupportMapFragment"
 android:layout_width="match_parent"
 android:layout_height="match_parent"/>
```

Activity:

```
public class MapsActivity extends FragmentActivity implements OnMapReadyCallback {
  private GoogleMap map;

  @Override protected void onCreate(Bundle b) {
    super.onCreate(b);
    setContentView(R.layout.activity_maps);

    SupportMapFragment f = (SupportMapFragment)getSupportFragmentManager()
        .findFragmentById(R.id.map);
    f.getMapAsync(this);
  }

  @Override public void onMapReady(GoogleMap googleMap) {
    map = googleMap;
    LatLng bangalore = new LatLng(12.9716, 77.5946);
    map.moveCamera(CameraUpdateFactory.newLatLngZoom(bangalore, 10f)); // initial
zoom

    // Zoom in/out programmatically:
    findViewById(R.id.zoomInBtn).setOnClickListener(v ->
        map.animateCamera(CameraUpdateFactory.zoomIn()));
    findViewById(R.id.zoomOutBtn).setOnClickListener(v ->
        map.animateCamera(CameraUpdateFactory.zoomOut()));
  }
}
```

Notes:

- Add Maps dependency + API key in manifest.
- You can also call CameraUpdateFactory.newLatLngZoom(latLng, zoomLevel) where zoom ~ 2 (world) to 21 (building).

Q10(a). Explain transactions with example.

Answer :
A **transaction** groups DB operations so they **all succeed or all fail**, ensuring **ACID**:

- **Atomicity:** all or nothing.
- **Consistency:** DB moves from one valid state to another.
- **Isolation:** concurrent transactions don't interfere.
- **Durability:** once committed, data survives crashes.

In SQLite on Android:

```
SQLiteDatabase db = helper.getWritableDatabase();

db.beginTransaction();

try {

  db.execSQL("INSERT INTO Accounts(id, balance) VALUES(1, 1000)");

  db.execSQL("UPDATE Accounts SET balance = balance - 200 WHERE id=1");

  db.execSQL("UPDATE Accounts SET balance = balance + 200 WHERE id=2");

  db.setTransactionSuccessful();  // COMMIT

} catch (Exception e) {

  // no setTransactionSuccessful() → ROLLBACK on endTransaction()

} finally {

  db.endTransaction();

}
```

Why use transactions:

- Prevents partial updates (e.g., money deducted but not credited).
- Faster: SQLite batches writes and fsyncs once at commit.

- Use for **bulk inserts** and multi-step updates.

Pitfalls:

- Don't do long I/O inside a transaction; keep it short.
- Handle exceptions and always call endTransaction() in finally.

Q10(b). Explain how we can create the tables.

Answer :
Use **SQLiteOpenHelper** to create/upgrade schema.

1) Define a Contract (good practice):

```
public final class StudentContract {
  private StudentContract() {}
  public static class Student implements BaseColumns {
    public static final String TABLE = "students";
    public static final String COL_NAME = "name";
    public static final String COL_AGE  = "age";
  }
}
```

2) Implement SQLiteOpenHelper:

```
public class DBHelper extends SQLiteOpenHelper {
  private static final String DB_NAME = "college.db";
  private static final int DB_VER = 1;

  public DBHelper(Context c) { super(c, DB_NAME, null, DB_VER); }

  @Override public void onCreate(SQLiteDatabase db) {
    String SQL = "CREATE TABLE " + StudentContract.Student.TABLE + " (" +
```

```
            StudentContract.Student._ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
            StudentContract.Student.COL_NAME + " TEXT NOT NULL, " +
            StudentContract.Student.COL_AGE  + " INTEGER CHECK(age>=0))";
    db.execSQL(SQL);

    // Optional: indexes for speed
    db.execSQL("CREATE INDEX idx_student_name ON " +
            StudentContract.Student.TABLE + "(" +
            StudentContract.Student.COL_NAME + ")");
  }

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

3) Use the helper:

```
DBHelper helper = new DBHelper(this);
SQLiteDatabase db = helper.getWritableDatabase();
ContentValues v = new ContentValues();
v.put(StudentContract.Student.COL_NAME, "Anita");
v.put(StudentContract.Student.COL_AGE, 21);
db.insert(StudentContract.Student.TABLE, null, v);
```

Notes & best practices:

- Prefer INTEGER PRIMARY KEY AUTOINCREMENT for ids.
- Define NOT NULL, UNIQUE, CHECK constraints to maintain integrity.

- Migrate with onUpgrade() using ALTER TABLE in production (avoid dropping user data).
- Keep schema versioned and write migration scripts when increasing DB_VER.

(Optional quick recap you can quote in answers)

- Draw small diagrams for **Q1(a)**, **Q7(a)**, **Q7(b)**, **Q8(b)**.
- For code questions, include **manifest permissions** where relevant.
- Always mention **why** each point matters (performance, security, battery).

If you want, I can package this into a clean **print-friendly PDF** with headings and diagram figures.

You said:

yes

ChatGPT said:

Perfect 👍 Thanks for confirming! Since you want **all 10 NAD modules with their 10 possible questions and very detailed answers (elaborated explanations for each point)**, I'll build them step by step.

Here's how I'll structure it for clarity:

- **Module name**
- **10 important questions (theoretical + practical mix)**
- **Elaborated answers (with explanations, real-world relevance, and easy-to-understand breakdowns)**

📘 Module 1 – Introduction to Data Communication & Networking

Q1. What is Data Communication? Explain the essential components of a communication system.

Answer:

- **Data Communication** refers to the exchange of data between two devices (like computers, mobiles, or servers) using a transmission medium such as cables or wireless signals.

- The **5 essential components** are:

    1. **Message** – The actual data (text, image, video). Without data, communication has no purpose.
    2. **Sender** – The device/system that generates and sends the message (e.g., a laptop).
    3. **Receiver** – The destination device/system that accepts the message (e.g., a server).
    4. **Transmission Medium** – The physical path or channel (e.g., copper cable, fiber optic, radio waves).
    5. **Protocol** – The set of rules governing communication (e.g., TCP/IP, HTTP).

💡 Example: When you send a WhatsApp message – your phone (sender) → WiFi network (medium) → server (protocol applied) → friend's phone (receiver).

Q2. What are the different types of data flows in communication?

Answer:

1. **Simplex** – One-way communication only. Example: TV broadcast (station → user).
2. **Half-duplex** – Both directions but one at a time. Example: Walkie-talkie.
3. **Full-duplex** – Both directions simultaneously. Example: Phone call or Zoom meeting.

Each flow type affects **speed, efficiency, and application suitability**. For instance, online gaming requires full-duplex, but an announcement system works fine with simplex.

Q3. Differentiate between Analog and Digital Signals.

Answer:

- **Analog Signal** – Continuous, smooth waveform (e.g., sound waves, radio signals).
- **Digital Signal** – Discrete steps (0s and 1s) representing binary data (used in computers).

💡 Example: A landline phone uses **analog**, while your computer network uses **digital**. Digital signals are less affected by noise and easier to encrypt.

Q4. Explain Transmission Modes with examples.

Answer:

- **Serial Transmission** – Data bits are sent one by one over a single channel (e.g., USB cable).
- **Parallel Transmission** – Multiple bits are sent simultaneously (e.g., printer cable, internal CPU communication).

👉 Serial is slower but works for long distances. Parallel is faster but works best for short distances.

Q5. What is a Network? Explain its types.

Answer:

- A **Network** is a collection of interconnected devices that share resources and communicate.
- Types:
    1. **LAN (Local Area Network)** – Small area (office, home).
    2. **MAN (Metropolitan Area Network)** – City-wide networks.
    3. **WAN (Wide Area Network)** – Large scale, across countries (e.g., Internet).

Q6. What are the advantages and disa

Module–1

Q1(a). Explain Android Architecture.

Answer :
Android is a **layered software stack** that lets apps run safely and efficiently on many kinds of hardware.

1) Linux Kernel (Hardware Abstraction & Core Services)

- **Role:** Foundation of the stack; provides process scheduling, memory mgmt, file system, networking, power mgmt.
- **Drivers:** Display, camera, audio, binder (IPC), Wi-Fi, Bluetooth, sensors.

- **Why it matters:** Same kernel APIs across devices ⇒ portability; robust security primitives (UIDs, process isolation).

2) Native C/C++ Libraries

- **Graphics:** Skia (2D), OpenGL ES/Vulkan (3D).
- **Media:** libstagefright/MediaCodec (codecs), OpenMAX AL.
- **Database:** SQLite (embedded relational DB).
- **Web:** WebKit/Blink components for rendering.
- **Security:** SSL, zlib, etc.
- **Why:** High-performance components that apps access through the framework.

3) Android Runtime (ART; earlier Dalvik)

- **Core libraries:** Java/Kotlin standard libs + Android-specific APIs.
- **ART:** Ahead-of-Time (AOT) compilation + JIT + profile-guided optimization; garbage collection; better startup than Dalvik.
- **Per-app isolation:** Every app runs in its own process with a unique Linux UID (sandbox).

4) Application Framework (Java/Kotlin APIs)
Managers that apps use:

- **Activity/Fragment Manager:** screen navigation & back stack.
- **Window & View System:** draw UI and handle events.
- **Package Manager:** install/update apps & permissions.
- **Content Providers:** structured data sharing (e.g., Contacts).
- **Resource Manager:** load layouts, strings, drawables for localization.
- **Location/Notification/Sensor/Telephony managers** etc.
- **Why:** High-level, device-independent APIs = faster development.

5) Applications (System + User Apps)

- Preinstalled (Dialer, Settings) + third-party apps.
- **Intent system** lets apps request actions from other apps securely (e.g., share image, open URL).

Key properties: modularity, reuse of components, security by sandbox & permissions, and efficiency via native libs + ART.

Q1(b). List any Ten features of Android *(explain each briefly).*

Answer :

1. **Open Source (AOSP):** OEMs can customize; large community support.
2. **Application Sandbox:** Per-app UID, process isolation, SELinux—limits damage of compromised apps.
3. **Rich Component Model:** Activities, Services, Broadcast Receivers, Content Providers ⇒ composable apps.
4. **Intent-based IPC:** Decouples requesters and providers (e.g., "share" works with any app that can share).
5. **Resource System:** Alternate resources for screen sizes, locales, night mode ⇒ single APK adapts broadly.
6. **Notifications & Foreground Services:** Engage users outside the app; channels, actions, priorities.
7. **Connectivity Stack:** Wi-Fi, 5G, Bluetooth, NFC, USB, WebRTC—easy network programming.
8. **Location & Sensors:** GPS + fused provider, accelerometer, gyroscope, magnetometer, proximity, light ⇒ AR, fitness, maps.
9. **Graphics & Media:** Hardware-accelerated UI, OpenGL/Vulkan, high-quality audio/video codecs.
10. **App Distribution:** Google Play & other stores; in-app updates, billing, crash reporting ecosystem.

(You can swap in other features—multitasking, widgets, backup/restore, accessibility—depending on what you remember best.)

Q2(a). Describe Android Development Tools (ADT).

Answer :
Historically, **ADT** was an Eclipse plugin; today **Android Studio (IntelliJ-based)** is the official IDE. Exams often say ADT—explain features common to both.

1. **Project Wizards & Gradle Build System:** Create templates (Empty Activity, Master/Detail). Gradle handles compilation, resource merging, AAPT2 packaging, signing, product flavors, build types (debug/release).
2. **Layout & Resource Editors:** Drag-and-drop **Layout Editor**, ConstraintLayout tools, preview for multiple devices/locales; vector/drawable editors; translation editor.
3. **Emulator & Device Manager:** Virtual devices with different CPU/ABI, screen sizes, sensors, camera, location, network throttling; snapshot/quick boot.
4. **Debugging (Logcat, Breakpoints, Watches):** Inspect variables, step through code, filters for logs, structured logging.
5. **Profilers:** CPU, Memory, Network profilers; energy profiler to find battery drains; heap dumps & leak detection.
6. **APK Analyzer & App Inspection:** Inspect APK size, dex methods, resources, manifest; database inspector; network inspector.
7. **Testing Support:** JUnit, Instrumentation/Espresso UI tests, Robolectric; test orchestrator; coverage reports.
8. **Version Control Integration:** Git built-in (branching, merging, blame).
9. **Dependency Manager:** Gradle/Maven pulls libraries (material, lifecycle, maps, retrofit).
10. **Packaging & Signing:** Keystore mgmt, app bundles (AAB), split APKs, proguard/R8 minification & obfuscation.

Q2(b). Explain Dalvik Virtual Machine.

Answer :
Dalvik (pre-ART) executed Android apps until Lollipop.

- **Execution Format:** Converts JVM class files to a single **.dex** (Dalvik Executable) with constant-pool sharing ⇒ reduced size.
- **Register-based VM:** Uses 16-bit instructions operating on registers ⇒ fewer instructions than JVM's stack VM for many ops; optimized for low memory/CPU.
- **Per-Process Instance:** Each app has its own Dalvik instance → isolation + independent GC.
- **Just-In-Time (JIT) Compilation:** Later Dalvik versions used JIT to compile hot code paths at runtime.

- **Garbage Collection:** Stop-the-world GC tuned for mobile footprints.
- **Lifecycle:** Replaced by **ART**, which adds AOT + better GC & profiling, but Dalvik is important historically and for understanding .dex & register model.

Module–2

Q3(a). Explain Directory Structure of an Android Project.

Answer :

```
MyApp/
├─ app/
│  ├─ src/
│  │  ├─ main/
│  │  │  ├─ java/ or kotlin/     ← app source (packages)
│  │  │  ├─ res/                 ← resources
│  │  │  │  ├─ layout/           (XML UIs)
│  │  │  │  ├─ drawable/         (png/svg/xml shapes)
│  │  │  │  ├─ mipmap-*/         (launcher icons)
│  │  │  │  ├─ values/           (strings.xml, colors.xml, dimens.xml, styles.xml)
│  │  │  │  ├─ menu/             (menus)
│  │  │  │  ├─ xml/              (preferences, file_paths)
│  │  │  ├─ AndroidManifest.xml  ← components, permissions, intent-filters
│  │  ├─ test/ & androidTest/    ← unit & instrumented tests
│  ├─ build.gradle (module)      ← deps, minSdk/targetSdk
├─ build.gradle (project)        ← repositories, plugins
├─ gradle/ & settings.gradle     ← build config
```

- **assets/** (optional): raw bundled files accessible via AssetManager.
- **.gitignore, proguard-rules.pro:** version control & shrinker configuration.

- **Why:** Separation of code vs resources enables localization, theming, and easy reuse.

Q3(b). Describe Fundamental UI Design in Android.

Answer :

1. **Layout Systems:**
    - **ConstraintLayout:** modern, flexible, flat hierarchies; constraints to parent/siblings; chains, guidelines.
    - **LinearLayout:** rows/columns; weights for proportional space.
    - **RelativeLayout/FrameLayout:** relative positioning or stack children; good for overlays.
    - **Why:** choose based on complexity and performance.
2. **View & ViewGroup:**
    - **View:** basic UI element (TextView, Button, ImageView).
    - **ViewGroup:** containers that control child layout (RecyclerView, ScrollView).
    - **Measure–Layout–Draw** pass defines rendering order.
3. **Material Design Principles:**
    - **Hierarchy & Emphasis:** typography scale, color contrast, elevation.
    - **Motion/Feedback:** ripple effects, meaningful transitions.
    - **Adaptive UI:** responsive breakpoints, smallest-width (sw) qualifiers, orientation handling.
4. **State & Interaction:**
    - **Listeners:** onClick, onTouch, onFocusChange.
    - **State lists:** color/drawable selectors for pressed/focused/disabled.
    - **Accessibility:** content descriptions, touch target ≥ 48dp.
5. **Theming & Styles:**
    - Define styles once (fonts, paddings); apply via themes (Theme.Material3) to keep UIs consistent and easily dark-mode-ready.
6. **Performance Tips:**

- Minimize nested layouts, use RecyclerView for long lists, use vector drawables, enable layout inspector to catch overdraw.

Q4(a). Write a program using RelativeLayout.

Answer :

layout/activity_main.xml

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

  <ImageView
    android:id="@+id/avatar"
    android:layout_width="72dp"
    android:layout_height="72dp"
    android:src="@mipmap/ic_launcher"
    android:contentDescription="@string/app_name"
    android:layout_alignParentStart="true"
    android:layout_marginEnd="12dp"/>

  <TextView
    android:id="@+id/name"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Rakesh Kumar"
    android:textStyle="bold"
```

```
    android:textSize="18sp"
    android:layout_toEndOf="@id/avatar"
    android:layout_alignTop="@id/avatar"/>

  <TextView
    android:id="@+id/subtitle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Senior Product Manager"
    android:layout_below="@id/name"
    android:layout_toEndOf="@id/avatar"/>

  <Button
    android:id="@+id/follow"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="Follow"
    android:layout_below="@id/avatar"
    android:layout_alignParentEnd="true"/>
</RelativeLayout>
```

MainActivity.java (or Kotlin)

```
public class MainActivity extends AppCompatActivity {
 @Override protected void onCreate(Bundle b) {
  super.onCreate(b);
  setContentView(R.layout.activity_main);
  findViewById(R.id.follow).setOnClickListener(v ->
```

```
    Toast.makeText(this, "Following!", Toast.LENGTH_SHORT).show());
  }
}
```

Why RelativeLayout here? Easy to place name to the **right of** the avatar and **below/ aligned** constraints without deep nesting.

Q4(b). Explain Ten common Android view attributes *(with effect & example).*

Answer :

1. **android:id** – unique reference to use in Java/Kotlin and for relative rules.
   Ex: @+id/submitBtn.
2. **android:layout_width / android:layout_height** – size specifiers: wrap_content, match_parent, exact dp.
   Why: controls measure phase.
3. **android:layout_margin** – outer spacing; can be per-side (marginStart, marginTop …).
   Ex: avoid overlap, add breathing room.
4. **android:padding** – inner spacing between content and view bounds.
   Ex: TextView padding improves readability.
5. **android:gravity** – how content is placed **inside** the view (e.g., center, end).
   Ex: center text vertically in a button.
6. **android:text / android:hint** – display string and placeholder; typically from resources (@string/...) for localization.
7. **android:src / android:background** – image content vs background styling (colors, shape drawables).
   Tip: use backgroundTint to theme buttons.
8. **android:visibility** – visible, invisible (keeps space), gone (removes from layout).
   Why: dynamic UI changes without re-inflation.
9. **android:orientation** *(for LinearLayout)* – vertical or horizontal. Works with **layout_weight** to distribute space.
10. **android:onClick** – binds a public method in activity to the click event via XML; quick for demos.
    Better: use listeners or data binding in production.

Module–3

Q5(a). Write a program of TimePicker with AM/PM.

Answer :

layout/activity_time.xml

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

 <TimePicker
    android:id="@+id/timePicker"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:is24HourView="false"/>

 <TextView
    android:id="@+id/selected"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:paddingTop="12dp"/>
</LinearLayout>
```

Activity:

```
public class TimeActivity extends AppCompatActivity {
 @Override protected void onCreate(Bundle b) {
  super.onCreate(b);
  setContentView(R.layout.activity_time);
```

```
   TimePicker tp = findViewById(R.id.timePicker);
   TextView tv = findViewById(R.id.selected);
   tp.setIs24HourView(false); // AM/PM

   tp.setOnTimeChangedListener((view, hour, minute) -> {
     int displayHour = (hour == 0) ? 12 : (hour > 12 ? hour - 12 : hour);
     String ampm = (hour >= 12) ? "PM" : "AM";
     tv.setText(String.format(Locale.getDefault(), "%02d:%02d %s", displayHour, minute, ampm));
   });
 }
}
```

Notes: Use android:is24HourView="false" for 12-hour format; convert hour to AM/PM for display.

Q5(b). List properties of ListView *(and how they affect UI).*

Answer :

- **android:divider / android:dividerHeight** – visual separation between rows; improves readability.
- **android:choiceMode** – none, singleChoice, multipleChoice; enables checked states for selection lists.
- **android:fastScrollEnabled** – adds a scrollbar thumb for long lists; UX boost for large datasets.
- **android:listSelector** – drawable used to highlight pressed/selected rows (ripple, color).
- **android:entries** – bind from a string-array resource for simple static lists.
- **android:stackFromBottom** – start list from bottom (e.g., chat history).

- **android:transcriptMode** – auto-scroll when items change; useful for chats/logs.
- **android:scrollbarStyle** – inside overlay vs outside inset; affects aesthetics.
- **android:footerDividersEnabled / headerDividersEnabled** – control extra dividers near header/footer views.
- **android:drawSelectorOnTop** – draw selector above content for visibility.

Adapter Example *(typical usage)*:

```
ListView lv = findViewById(R.id.list);
ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
    android.R.layout.simple_list_item_1,
    Arrays.asList("One","Two","Three"));
lv.setAdapter(adapter);
```

(Note: modern apps prefer RecyclerView for flexibility & performance.)

Q6(a). List properties of ProgressBar *(with meaning & use).*

Answer :

- **style** – ?android:attr/progressBarStyle (small, large), or **horizontal** for determinate bars.
- **android:indeterminate** – true for unknown duration (spinning wheel).
- **android:max** – maximum value for determinate bar (default 100).
- **android:progress** – current completed value.
- **android:secondaryProgress** – buffer state (e.g., media buffering).
- **android:visibility** – show/hide during tasks.
- **android:progressTint / indeterminateTint** – tint colors with theme.
- **android:layout_width/height & layout_margin** – placement and size.
- **android:animateLayoutChanges (parent)** – smooth appearance/disappearance.
- **setProgress(int, boolean) (code)** – animate progress updates.

Determinate example:

```
<ProgressBar
 style="@android:style/Widget.ProgressBar.Horizontal"
 android:id="@+id/pb"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:max="100"
 android:progress="0"/>
pb.setProgress(60, true); // 60% with animation
```

Q6(b). Write a program for ToggleButton.

Answer :

layout:

```
<ToggleButton
 android:id="@+id/toggle"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:textOn="Wi-Fi ON"
 android:textOff="Wi-Fi OFF"/>
<TextView
 android:id="@+id/status"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"/>
```

Activity:

```
public class ToggleActivity extends AppCompatActivity {
 @Override protected void onCreate(Bundle b) {
  super.onCreate(b);
```

```
  setContentView(R.layout.activity_toggle);

  ToggleButton t = findViewById(R.id.toggle);
  TextView status = findViewById(R.id.status);

  t.setOnCheckedChangeListener((button, checked) -> {
   status.setText(checked ? "Service enabled" : "Service disabled");
   // doWork(checked); // e.g., enable/disable feature
  });

  // restore state on rotation
  if (b != null) t.setChecked(b.getBoolean("state", false));
 }

 @Override protected void onSaveInstanceState(Bundle out) {
  super.onSaveInstanceState(out);
  out.putBoolean("state", ((ToggleButton)findViewById(R.id.toggle)).isChecked());
 }
}
```

Module–4

Q7(a). Explain Activity Life Cycle with diagram.

Answer :

States & Callbacks:

1. **onCreate()** – 1st time setup: inflate layout, init ViewModel/adapters, restore state from savedInstanceState.

2. **onStart()** – Activity becomes **visible** (UI exists, not yet foreground). Start lightweight UI updates.
3. **onResume()** – Activity in **foreground**; start camera, sensors, animations, listeners.
4. **onPause()** – Partially obscured (dialog/another activity). **Commit unsaved changes**, pause animations, heavy sensors. Keep state quick; this must be fast.
5. **onStop()** – Fully hidden. Release expensive resources (camera/GPS). Persist to DB.
6. **onRestart()** – Coming back from Stopped → **onStart()** called next.
7. **onDestroy()** – Final cleanup; may be called due to finish() or system destroying to reclaim memory.

Diagram (flow):

onCreate → onStart → onResume

↑ ↓

onRestart onPause → onStop → onDestroy

↘ (back to foreground) ↗

Why it matters: Correct placement of code prevents **memory leaks**, **ANR**, and **battery drain**.

Lifecycle logging snippet:

```
@Override protected void onResume(){ super.onResume(); Log.d("LC","resume"); }
```

Q7(b). Explain with neat diagram: Service Life Cycle.

Answer :

Types of services:

- **Started Service:** startService() / startForegroundService() – runs until it stops itself (stopSelf()/stopService()), good for long tasks (music).
- **Bound Service:** bindService() – provides an API to clients (e.g., IPC with Messenger/Binder). Runs as long as at least one client is bound.

Callbacks:

- **onCreate()** – one-time init (threads, notification channel).

- **onStartCommand(Intent, flags, startId)** – for started services; return value (START_STICKY, etc.) controls restart behavior if killed.
- **onBind(Intent)** – return IBinder for clients; called on first bind.
- **onUnbind()/onRebind()** – when clients disconnect/reconnect.
- **onDestroy()** – cleanup resources, stop threads, release wake locks.

Diagram (simplified):

startService() stopSelf()/stopService()

Client ----------> onStartCommand() ----------> onDestroy()

Client ---- bindService() --> onBind() -- use binder API -- unbindService() --> onUnbind() -> onDestroy()

Foreground service: shows a persistent notification (e.g., music playback, location). Required for long-running background tasks from Android O+.

Q8(a). Explain types of sensors in Android.

Answer :

1) Motion Sensors (measure acceleration/rotation):

- **Accelerometer:** linear acceleration including gravity (m/s^2).
- **Gyroscope:** rate of rotation (rad/s).
- **Gravity/Linear Accel:** virtual sensors derived from others.
  Use cases: shake to refresh, step detection, gaming, stabilization.

2) Environmental Sensors (ambient conditions):

- **Light (lux), Proximity (cm), Pressure (hPa), Temperature (°C), Humidity (%).**
  Use cases: auto-brightness, pocket detection, weather apps, altimeter.

3) Position Sensors:

- **Magnetometer:** ambient magnetic field; with accelerometer gives **compass**.
- **GPS (location provider):** satellite location; via Fused Location combines GPS + Wi-Fi + Cell for efficiency.
  Use cases: navigation, geo-fencing, AR alignment.

4) Biometric/Health (device-specific):

- **Fingerprint, face unlock, heart rate** (on wearables).
  Use cases: authentication, wellness.

Access pattern & best practices:

SensorManager sm = (SensorManager)getSystemService(SENSOR_SERVICE);

Sensor accel = sm.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);

sm.registerListener(listener, accel, SensorManager.SENSOR_DELAY_GAME);

- Unregister in onPause() to save battery.
- Use **low-pass/high-pass filters** to separate gravity vs motion.
- Check sensor.getVendor() and getResolution() for quality.

Q8(b). With diagram, explain Multimedia Framework Architecture.

Answer :

High-level pipeline (playback):

App (MediaPlayer/ExoPlayer)

↓

Framework APIs (MediaPlayer, AudioManager, CameraX)

↓

Media Server (MediaCodec, Stagefright pipeline)

↓

HAL (AudioFlinger, Audio HAL / Camera HAL)

↓

Hardware (DSP, codecs, camera sensor, speakers)

Components:

- **Application Layer:** Uses MediaPlayer, AudioTrack, VideoView, or **ExoPlayer** (Google's library) to play/record.
- **Framework APIs:** Manage focus, routing, DRM, subtitles, audio attributes.

- **Native Media (Stagefright/MediaCodec):** Demuxers (MP4, MKV), decoders/encoders (H.264, AAC, Opus), time sync.
- **AudioFlinger:** Mixer for multiple audio streams, applies effects (equalizer).
- **OpenSL ES/AAudio:** Low-latency audio paths for games.
- **Camera stack:** App → Camera2/CameraX → Camera HAL → ISP → encoder.
- **Hardware Acceleration:** Many codecs run on DSP/GPU for battery efficiency.
- **DRM:** MediaDrm/MediaCrypto for protected content.

Why: Layers isolate apps from hardware specifics while delivering **low latency**, **power-efficient** playback/recording.

Module–5

Q9(a). Write a program getting Location Data.

Answer :
Modern approach uses **Fused Location Provider (FLP)** from Google Play services (better accuracy & battery). Include **runtime permissions**.

Manifest:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

Request permission at runtime (Activity):

```
ActivityCompat.requestPermissions(this,
   new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, 100);
```

Code to get updates:

```
public class LocationActivity extends AppCompatActivity {
  private FusedLocationProviderClient fused;
  private LocationCallback callback;

  @Override protected void onCreate(Bundle b) {
    super.onCreate(b);
    setContentView(R.layout.activity_main);
```

```
   fused = LocationServices.getFusedLocationProviderClient(this);


   LocationRequest req = new
LocationRequest.Builder(Priority.PRIORITY_HIGH_ACCURACY, 5000)
         .setMinUpdateIntervalMillis(2000)
         .setWaitForAccurateLocation(true)
         .build();


   callback = new LocationCallback() {
    @Override public void onLocationResult(LocationResult res) {
     if (res == null) return;
     Location loc = res.getLastLocation();
     if (loc != null) {
      String msg = "Lat=" + loc.getLatitude() + "  Lon=" + loc.getLongitude();
      Toast.makeText(LocationActivity.this, msg, Toast.LENGTH_SHORT).show();
     }
    }
   };
   startUpdates(req);
  }


  private void startUpdates(LocationRequest req) {
   if (ActivityCompat.checkSelfPermission(this,
Manifest.permission.ACCESS_FINE_LOCATION)
      != PackageManager.PERMISSION_GRANTED) return;
   fused.requestLocationUpdates(req, callback, Looper.getMainLooper());
```

```
 }

 @Override protected void onPause() {
  super.onPause();
  fused.removeLocationUpdates(callback); // save battery
 }
}
```

Notes:

- Choose accuracy priority wisely; for geofences or coarse UI, use PRIORITY_BALANCED_POWER_ACCURACY.
- Always stop updates when not needed.

Q9(b). Write a program for zooming In or Out of the Map.

Answer :

layout:

```
<fragment
 android:id="@+id/map"
 android:name="com.google.android.gms.maps.SupportMapFragment"
 android:layout_width="match_parent"
 android:layout_height="match_parent"/>
```

Activity:

```
public class MapsActivity extends FragmentActivity implements OnMapReadyCallback {
 private GoogleMap map;

 @Override protected void onCreate(Bundle b) {
  super.onCreate(b);
```

```
  setContentView(R.layout.activity_maps);


  SupportMapFragment f = (SupportMapFragment)getSupportFragmentManager()
     .findFragmentById(R.id.map);
  f.getMapAsync(this);
 }


 @Override public void onMapReady(GoogleMap googleMap) {
  map = googleMap;
  LatLng bangalore = new LatLng(12.9716, 77.5946);
  map.moveCamera(CameraUpdateFactory.newLatLngZoom(bangalore, 10f)); // initial
zoom


  // Zoom in/out programmatically:
  findViewById(R.id.zoomInBtn).setOnClickListener(v ->
     map.animateCamera(CameraUpdateFactory.zoomIn()));
  findViewById(R.id.zoomOutBtn).setOnClickListener(v ->
     map.animateCamera(CameraUpdateFactory.zoomOut()));
 }
}
```

Notes:

- Add Maps dependency + API key in manifest.
- You can also call CameraUpdateFactory.newLatLngZoom(latLng, zoomLevel) where zoom ~ 2 (world) to 21 (building).

Q10(a). Explain transactions with example.

Answer :
A **transaction** groups DB operations so they **all succeed or all fail**, ensuring **ACID**:

- **Atomicity:** all or nothing.
- **Consistency:** DB moves from one valid state to another.
- **Isolation:** concurrent transactions don't interfere.
- **Durability:** once committed, data survives crashes.

In SQLite on Android:

```
SQLiteDatabase db = helper.getWritableDatabase();

db.beginTransaction();

try {

  db.execSQL("INSERT INTO Accounts(id, balance) VALUES(1, 1000)");

  db.execSQL("UPDATE Accounts SET balance = balance - 200 WHERE id=1");

  db.execSQL("UPDATE Accounts SET balance = balance + 200 WHERE id=2");

  db.setTransactionSuccessful();  // COMMIT

} catch (Exception e) {

  // no setTransactionSuccessful() → ROLLBACK on endTransaction()

} finally {

  db.endTransaction();

}
```

Why use transactions:

- Prevents partial updates (e.g., money deducted but not credited).
- Faster: SQLite batches writes and fsyncs once at commit.
- Use for **bulk inserts** and multi-step updates.

Pitfalls:

- Don't do long I/O inside a transaction; keep it short.
- Handle exceptions and always call endTransaction() in finally.

Q10(b). Explain how we can create the tables.

Answer :
Use **SQLiteOpenHelper** to create/upgrade schema.

1) Define a Contract (good practice):

```
public final class StudentContract {
  private StudentContract() {}
  public static class Student implements BaseColumns {
    public static final String TABLE = "students";
    public static final String COL_NAME = "name";
    public static final String COL_AGE  = "age";
  }
}
```

2) Implement SQLiteOpenHelper:

```
public class DBHelper extends SQLiteOpenHelper {
  private static final String DB_NAME = "college.db";
  private static final int DB_VER = 1;

  public DBHelper(Context c) { super(c, DB_NAME, null, DB_VER); }

  @Override public void onCreate(SQLiteDatabase db) {
    String SQL = "CREATE TABLE " + StudentContract.Student.TABLE + " (" +
            StudentContract.Student._ID + " INTEGER PRIMARY KEY AUTOINCREMENT, " +
            StudentContract.Student.COL_NAME + " TEXT NOT NULL, " +
            StudentContract.Student.COL_AGE  + " INTEGER CHECK(age>=0))";
    db.execSQL(SQL);
```

```
    // Optional: indexes for speed
    db.execSQL("CREATE INDEX idx_student_name ON " +
            StudentContract.Student.TABLE + "(" +
            StudentContract.Student.COL_NAME + ")");
  }

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

3) Use the helper:

```
DBHelper helper = new DBHelper(this);
SQLiteDatabase db = helper.getWritableDatabase();
ContentValues v = new ContentValues();
v.put(StudentContract.Student.COL_NAME, "Anita");
v.put(StudentContract.Student.COL_AGE, 21);
db.insert(StudentContract.Student.TABLE, null, v);
```

Notes & best practices:

- Prefer INTEGER PRIMARY KEY AUTOINCREMENT for ids.
- Define NOT NULL, UNIQUE, CHECK constraints to maintain integrity.
- Migrate with onUpgrade() using ALTER TABLE in production (avoid dropping user data).
- Keep schema versioned and write migration scripts when increasing DB_VER.
