



KAMALDNP

kamaldnp.com • IT Training, Varanasi

PDF created by Raj Kamal using AI

Short Notes — Mobile App Development

BTEUP Diploma in Computer Science & Engineering • 5th Semester • Short Notes
Session 2026-27

Short Revision Notes — prepared by KAMALDNP Academy from the official BTEUP 2026-27 syllabus. This is NOT a BTEUP board paper or publication.

These short notes cover **Mobile App Development**, a programme elective in the BTEUP 2026-27 fifth-semester scheme for Computer Science and Engineering — students choose either this subject or Multimedia Technologies as one elective. In the 2026-27 scheme the subject is a **practicum**, so most marks come from practical work, but a clear theory base is still essential for the viva and for doing the practicals correctly.

Use the notes unit by unit: read the points, revise the one-line summary at the end, and try the likely question for each unit in your own words. Always match your final preparation with the official syllabus PDF and your lab records.

Unit I — Introduction to Android

Topics: The Android platform; Android SDK; Eclipse installation; Android installation; building your first Android application; understanding the anatomy of an Android application; Android manifest file.

The Android platform

The **Android platform** is an open-source software stack for mobile devices, based on the Linux kernel. It has layers: the **Linux kernel** at the bottom for drivers, security and memory management, then native libraries and the Android Runtime, then the application framework, and apps at the top.

Key features examiners expect:

- Open source and free, so any manufacturer can use it.
- Apps are normally written in Java or Kotlin and run on the Android Runtime.
- Supports touch screens, GPS, camera, sensors and telephony.
- Rich development environment with an emulator for testing without a real phone.

Because Android runs on phones, tablets and other devices with different screen sizes, an app must handle different hardware gracefully — this idea runs through the whole subject.

Android SDK and Eclipse installation

The **Android SDK (Software Development Kit)** is the set of tools needed to build Android apps. It contains platform tools, build tools, system images for the emulator, documentation and sample code. Without the SDK, source code cannot be compiled into an installable app package.

In the syllabus setup, **Eclipse** is used as the IDE with the ADT (Android Development Tools) plug-in. Installation steps in order:

- Install the JDK first, because Eclipse and the SDK tools need Java.
- Install Eclipse and then add the ADT plug-in from its update site.
- Install the Android SDK and point Eclipse to its folder in Preferences.
- Use the SDK Manager to download the required platform version and create an AVD (Android Virtual Device) for testing.

In viva, be ready to state what JDK, SDK, ADT and AVD each mean.

Android installation and building your first Android application

Android installation here means setting up the development system correctly and verifying it: JDK installed, Eclipse with ADT working, SDK platforms downloaded, and an emulator (AVD) or a real device with USB debugging enabled for running apps.

Building your first application follows a fixed sequence:

- Create a new Android Application Project — give the application name, project name and package name.
- Choose the minimum and target SDK versions and a blank activity.
- The IDE generates the main activity file, the layout XML and the manifest.
- Edit the layout and activity code, then run the project on the AVD or device.

The output is an **APK** file that Android installs. If the first simple app (for example, showing a welcome message) runs on the emulator, the whole toolchain is proved working.

Anatomy of an Android application and Android manifest file

The **anatomy** of an Android application means its standard project structure:

- **src** — Java source files, including activities.
- **res/layout** — XML files defining screens; **res/values** — strings, colours and styles; **res/drawable** — images.
- **gen** — automatically generated files, including R.java which links code to resources.
- **assets** — raw files bundled with the app, and **libs** for libraries.

The **Android manifest file (AndroidManifest.xml)** is present in every app. It declares the package name, the components (activities, services, receivers, providers), the permissions the app needs, and the minimum SDK. Android reads the manifest before running any code — an activity not declared in the manifest cannot be started at all. This declaration role of the manifest is a very common viva point.

Likely question: Describe the Android platform and the anatomy of an Android application. What is the role of the Android manifest file?

Unit II — Android Application Design Essentials

Topics: Anatomy of an Android application; Android terminologies; application context; activities; services; intents; receiving and broadcasting intents; Android manifest file and its common settings; using intent filters; permissions.

Anatomy of an Android application and Android terminologies

This unit revisits the **anatomy** of an app in design terms. The four building-block components are **Activity** (one screen with a user interface), **Service** (background work without a screen), **Broadcast Receiver** (responds to system or app broadcasts) and **Content Provider** (shares structured data between apps).

Important **Android terminologies** to define in one line each:

- **APK** — the installable application package file.
- **AVD** — Android Virtual Device, the emulator configuration.
- **Intent** — a message used to request an action from another component.
- **Context** — handle to information about the application environment.
- **Resources** — external files such as layouts, strings and images kept outside code.

An app is therefore not one program file but a set of declared components plus resources, assembled by the build tools into one APK.

Application context, activities and services

The **application context** is an object that gives access to application-wide information and resources — for example, to start an activity, access strings, or get a system service. It lives

as long as the app process lives, so it is safe for long-lived operations, unlike an activity context which is tied to one screen.

An **activity** represents one screen. Its life cycle moves through onCreate, onStart, onResume, onPause, onStop and onDestroy as the user opens, leaves and closes the screen. Code for setup belongs in onCreate.

A **service** performs long-running work in the background without a user interface, such as playing music or downloading a file. A service is started with startService or bound by another component. Difference to remember: an activity is visible and interactive; a service is invisible and keeps working when the screen changes.

Intents — receiving and broadcasting intents

An **intent** is a message object used to request an action — starting an activity, starting a service, or delivering a broadcast. It can carry data as extras (key-value pairs).

- **Explicit intent** — names the exact target component, used inside one app, e.g. opening the second screen of your own app.
- **Implicit intent** — declares only the general action (view, dial, share); Android chooses a suitable app, e.g. opening a web link in a browser.

Broadcasting an intent means sending a message that many receivers can hear, using sendBroadcast. **Receiving** is done by a broadcast receiver whose onReceive method runs when a matching broadcast arrives — for example, battery low or network changed. Receivers must be declared in the manifest or registered in code, and onReceive must finish quickly.

Manifest file settings, intent filters and permissions

Common **manifest settings** include the package name, version code and version name, minimum and target SDK, the application label and icon, and declarations of every activity, service, receiver and provider. The launcher activity is marked with the MAIN action and LAUNCHER category.

An **intent filter** is written inside a component's manifest entry. It states which actions and categories the component can handle, so Android can route matching implicit intents to it. A component without an intent filter can only be started by an explicit intent.

Permissions protect the user's privacy and device functions. Dangerous capabilities — internet, camera, location, contacts, SMS — must be declared with uses-permission entries in the manifest. Without the required permission, the operation fails at run time. In viva, always connect the three: manifest declares the component, the intent filter advertises what it can receive, and permissions declare what it is allowed to do.

Likely question: Differentiate between an activity and a service. Explain explicit and implicit intents with examples.

Unit III — Android User Interface Design Essentials

Topics: User interface screen elements; designing user interfaces with layouts; drawing and working with animation.

User interface screen elements

Android screens are built from **views** (single elements) arranged inside **view groups** (containers). Everything visible is a view.

Common screen elements:

- **TextView** — displays text; **EditText** — takes text input from the user.
- **Button** and **ImageButton** — tappable controls that raise click events.
- **CheckBox**, **RadioButton**, **Switch** — selection controls; radio buttons work in a group where only one is selected.
- **ImageView** — displays a picture; **ListView** / **RecyclerView** — scrollable lists of items.
- **ProgressBar**, **Spinner**, **DatePicker** — progress display and ready-made selection inputs.

Each element has attributes such as id, width, height, text, colour and padding, set in the layout XML or in code. Views are identified in code through their id. Good design keeps elements large enough to touch, labels clear, and input controls matched to the type of data expected.

Designing user interfaces with layouts

A **layout** is a view group that controls how its child views are positioned on the screen. Layouts are normally written in XML, separate from the Java code, so the design can change without changing logic.

- **LinearLayout** — arranges children in one row or one column.
- **RelativeLayout** — positions children relative to each other or to the parent.
- **ConstraintLayout** — positions views with constraints; flexible and recommended for complex screens.
- **TableLayout** — arranges children in rows and columns like a table.
- **FrameLayout** — stacks children on top of each other in one area.

Use **wrap_content** where the view should fit its content and **match_parent** where it should fill the available space. Because phone screens differ in size, prefer layouts and density-independent sizes over fixed pixel positions, and test the screen on more than one emulator size.

Drawing and working with animation

Drawing in Android can be done in three main ways: using ready drawable resources (images and shapes defined in XML), using views provided by the system, or drawing custom graphics on a **Canvas** inside a custom view's `onDraw` method, where lines, circles, rectangles and text are drawn with a `Paint` object.

Animation makes the interface feel alive and guides the user's attention. The main types are:

- **View animation** — tween effects on a view: move (translate), scale, rotate and fade (alpha), defined in XML or code.
- **Frame animation** — a sequence of images shown one after another, like a film strip.
- **Property animation** — animates any property of an object over time with smooth value changes.

Keep animations short and purposeful — feedback on a tap, a screen transition, or showing progress. Excessive animation slows the app and irritates the user.

Likely question: Describe briefly the common layout types in Android. How is animation used in an Android user interface?

Unit IV — Testing and Publishing

Topics: Testing Android applications; publishing Android applications; using Android preferences; managing application resources in a hierarchy; working with different types of resources.

Testing Android applications

Testing checks that the app works correctly before users receive it. Testing is done on the emulator and on real devices, because screen sizes, Android versions and hardware differ.

Main levels and checks:

- **Unit testing** — tests one small piece of logic, such as a calculation, in isolation.
- **Functional / UI testing** — checks that screens open, buttons work, input is validated and navigation is correct.
- **Compatibility testing** — runs the app on different screen sizes and Android versions.
- **Performance and stress checks** — behaviour with low memory, slow network, rotation of the screen and incoming calls.

The Android testing framework provides test cases that run on a device or emulator. Every defect found is fixed and the related test is repeated — this cycle is called regression testing. Untested apps fail in the market; tested apps earn better ratings.

Publishing Android applications

Publishing means preparing the app and releasing it so users can install it, normally through the Google Play Store. The steps are:

- Test the final version thoroughly and remove debug settings and test data.
- Set the final version code and version name in the manifest / build file.
- Build a signed release APK using a private keystore — signing proves ownership and is compulsory for updates.
- Create a developer account, prepare the store listing (title, description, screenshots, icon, category) and upload the signed APK.
- After review, release the app; later updates must be signed with the same key and carry a higher version code.

Losing the keystore means updates can never be published under the same app listing, so the key must be kept safely. Sideloaded an APK directly is possible but store publishing gives users trust, easy updates and wider reach.

Using Android preferences

Preferences (SharedPreferences) are Android's simple way to store small amounts of app settings as key-value pairs — for example, the user's name, a sound on/off choice, or the last selected language. The data is stored in an XML file in the app's private storage and survives closing the app.

Working pattern:

- Get the preferences object for the app.
- Read values with methods such as `getString` or `getBoolean`, giving a default value in case the key is absent.
- Write values through an Editor — put values, then apply or commit.

Preferences suit small, simple settings only. They are not a database: large records, lists of students or transaction data belong in SQLite. A Preference screen can also be built so the user edits settings on a standard settings page. In viva, state one example of data fit for preferences and one fit only for a database.

Managing application resources — hierarchy and types of resources

Resources are all non-code files of the app, kept under the `res` folder so Android can select the right version automatically. Resources are organised in a **hierarchy** of named sub-folders, and each resource is accessed from code or XML through the generated `R` class.

Main types of resources:

- **layout** — XML screen designs; **drawable** — images and shape graphics.
- **values** — strings, colours, dimensions and styles in XML form.
- **menu** — menu definitions; **anim** — animation definitions; **raw** — files such as audio kept as-is.

The hierarchy supports alternatives: folders such as drawable-hdpi, values-hi (Hindi strings) or layout-land let Android pick resources matching the device's density, language or orientation. Keeping text in strings.xml instead of hard-coding it makes translation and correction easy — a point frequently asked in viva.

Likely question: Describe the steps for publishing an Android application. Differentiate between preferences and a database for storing app data.

Unit V — Using Common Android APIs

Topics: Using Android data and storage APIs; managing data using SQLite; sharing data between applications with content providers; using Android networking APIs; using Android web APIs; using Android telephony APIs; deploying an Android application to the world.

Android data and storage APIs and managing data using SQLite

Android offers several **storage options**, and choosing correctly is a favourite exam topic:

- **SharedPreferences** — tiny key-value settings.
- **Internal storage** — private files only this app can read; **external storage** — files on shared storage such as media.
- **SQLite database** — structured records in tables with queries.
- **Network storage** — data kept on a server and fetched through APIs.

SQLite is a light, file-based relational database built into Android. A helper class (SQLiteOpenHelper) creates the database and tables in onCreate and handles version changes in onUpgrade. Data work uses SQL operations — insert, query, update, delete — and query results are read row by row through a Cursor. Use SQLite when the app must store many structured records, such as attendance or product lists, searchable and updatable on the device itself.

Sharing data between applications with content providers

A **content provider** is the standard Android component for **sharing structured data between applications**. One app owns the data; other apps access it only through the provider's published interface, so the storage details and safety stay under the owner's control.

Key ideas:

- Data is addressed by a **content URI**, in the form content://authority/path/id.
- Clients use a **ContentResolver** to call query, insert, update and delete on that URI.
- The provider decides which data to expose and can enforce permissions, so private data is not leaked.

Built-in providers exist for contacts, media and call logs — that is how a messaging app reads contacts without owning them. In short: preferences and files are private, SQLite is private structured storage, and a content provider is the controlled doorway through which data is shared with other apps.

Android networking APIs and web APIs

Networking APIs let an app exchange data with servers over the internet. The classic built-in option is `URLConnection`; connectivity is checked first with `ConnectivityManager` so the app reacts gracefully when the device is offline. The `INTERNET` permission must be declared in the manifest. Network work must run on a background thread, never on the main UI thread, or the app will freeze.

Web APIs in this syllabus context cover using web content and services inside the app:

- Calling a server's web service (often returning JSON or XML) and parsing the response into app data.
- Displaying web pages inside the app with a **WebView**, which can also run JavaScript when enabled and bridge between web content and app code.

Typical viva example: a weather or news app fetches fresh data from a web API at start-up, parses it, and shows it in a list — local SQLite keeps the last copy for offline viewing.

Android telephony APIs and deploying an application to the world

The **telephony APIs** give controlled access to phone features. `TelephonyManager` provides information such as network type and device identifiers (subject to strict permissions), while intents provide the common user actions: opening the dialler with a number filled in, placing a call where permitted, and composing an SMS. `SmsManager` can send text messages from code in permitted use-cases. Because telephony touches privacy and cost, these features need manifest permissions and, in modern Android, run-time permission from the user.

Deploying an application to the world repeats the publishing discipline of Unit IV at full scale: final testing on real devices, a signed release build with the permanent key, a complete store listing, and a plan for updates and user feedback. After release, monitor crashes and ratings, fix defects, and publish updates with higher version codes signed with the same key — deployment is a continuing responsibility, not a one-day event.

Likely question: Explain the storage options available in Android. How does a content provider share data between applications?

Quick Revision — One Line Each

- Android is an open-source, Linux-based mobile platform; apps run as components plus resources packed in an APK.

- SDK = development tools; ADT plug-in connects the SDK to Eclipse; AVD is the emulator device.
- AndroidManifest.xml declares package, components and permissions — nothing runs unless it is declared.
- Four components: Activity (screen), Service (background), Broadcast Receiver (events), Content Provider (shared data).
- Explicit intent names its target; implicit intent names only the action and Android chooses the app.
- Intent filters in the manifest advertise which implicit intents a component can handle.
- Layouts: Linear (row/column), Relative, Constraint, Table, Frame (stacked).
- Animation types: view (tween), frame-by-frame, and property animation.
- Testing before publishing: unit, functional, compatibility — then sign the release APK with a permanent key.
- SharedPreferences = small key-value settings; SQLite = structured records in tables.
- Content providers share data through content URIs, accessed by other apps via ContentResolver.
- Network work needs the INTERNET permission and must run off the main thread.

KAMALDNP • kamaldnp.com • support@kamaldnp.com • +91 8318013442 • Varanasi, UP

Prepared by KAMALDNP Academy from the official BTEUP 2026-27 syllabus topics. Short explanations are the author's own plain-English summaries, not syllabus text. Please confirm with your own polytechnic for any local changes.