Android Interview Questions for FileViewer Project

1. Basic Android Concepts

Q1: What is an Activity, and how is it used in FileViewer?

Explanation: An `Activity` is a core Android component representing a single screen with a UI. In FileViewer, `MainActivity` serves as the main screen, hosting a `RecyclerView` for file listing and handling user interactions like search and dialogs.

Q2: Explain the Android Activity lifecycle.

Explanation: The lifecycle includes methods like `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, `onDestroy()`. In FileViewer, `onCreate()` initializes the UI and adapter, while permission checks occur during startup.

Q3: What is the purpose of `AndroidManifest.xml`?

Explanation: The manifest declares app components, permissions, and metadata. FileViewer uses it to declare `MainActivity`, request `READ_EXTERNAL_STORAGE`, and enable `requestLegacyExternalStorage`.

Q4: What are layouts, and why use `ConstraintLayout` in FileViewer?

Explanation: Layouts define UI structure. `ConstraintLayout` in `activity_main.xml`, `item_file.xml`, and `dialog_rename.xml` allows flexible, responsive designs with constraints, reducing view hierarchy depth.

Q5: What is the difference between `dp`, `sp`, and `px`?

Explanation: `dp` (density-independent pixels) ensures consistent sizing across screen densities; `sp` (scale-independent pixels) is for text, respecting user font size preferences; `px` (pixels) is device-specific. FileViewer uses `dp` in `dimens.xml` for padding and `sp` for text sizes.

2. UI Development

Q6: How does `RecyclerView` work, and why use `ListAdapter` in FileViewer?

Explanation: `RecyclerView` displays large datasets efficiently by recycling views. `ListAdapter` in `FileAdapter.kt` uses `DiffUtil` to compute list differences, optimizing updates for file list changes during navigation or search.

Q7: Explain the role of `FileAdapter.kt` in FileViewer.

Explanation: `FileAdapter.kt` binds `FileModel` data to `item_file.xml` views, handles clicks (file, rename, delete), and applies animations for directory expand/collapse, using `findViewById` for view access.

Q8: What are XML animations, and how are they used in FileViewer?

Explanation: XML animations define visual effects in `res/anim/`. FileViewer uses `expand.xml` and `collapse.xml` for 90° icon rotations in `FileAdapter.kt`, enhancing UX during directory toggling.

Q9: How do you ensure UI consistency across devices?

Explanation: Use `ConstraintLayout`, `dp`/`sp` units, and resource qualifiers (e.g., `values-sw600dp`). FileViewer's `dimens.xml` and vector drawables (`ic_folder.xml`, `ic_file.xml`) ensure consistent styling.

Q10: What is Material Design, and how does FileViewer leverage it?

Explanation: Material Design is Google's UI guideline for consistent, intuitive interfaces. FileViewer uses `material` library components (e.g., `ImageButton` in `item_file.xml`) and follows typography/spacing recommendations.

3. Data Handling

Q11: What is the purpose of `FileModel.kt`?

Explanation: `FileModel.kt` is a data class modeling a file/directory with properties (`file`, `isDirectory`, `children`, `isExpanded`). It supports hierarchical file navigation in FileViewer.

Q12: How does FileViewer handle file system access?

Explanation: Uses Java's `File` API (`Environment.getExternalStorageDirectory()`, `File.listFiles()`) in `MainActivity.kt` to load files, with sorting and filtering for display.

Q13: Explain the search functionality in FileViewer.

Explanation: `MainActivity.kt` uses `TextWatcher` on `searchEditText` to filter `fileList` by name, updating `FileAdapter` with `submitList`. `ListAdapter` ensures efficient UI updates.

Q14: How are file rename and delete operations implemented?

Explanation: `MainActivity.kt` uses `File.renameTo` for renaming (via `dialog_rename.xml`) and `File.delete` for deletion, with confirmation dialogs to prevent accidental actions.

Q15: What is `DiffUtil`, and why is it used in `FileAdapter.kt`?

Explanation: `DiffUtil` calculates differences between old and new lists, minimizing `RecyclerView` updates. `FileDiffCallback` in FileViewer compares `FileModel` by `absolutePath` and content for smooth list changes.

4. Permissions in Java (Detailed Explanation)

Concept: Android permissions protect sensitive data and features (e.g., storage, camera). In Java-based Android apps, permissions are declared in `AndroidManifest.xml` and requested at runtime for dangerous permissions (post-API 23). FileViewer uses `READ_EXTERNAL_STORAGE` to access files, leveraging Java's permission APIs.

Q16: What are Android permissions, and how are they categorized?

Explanation: Permissions are categorized as:

- **Normal**: Auto-granted (e.g., `INTERNET`).

- **Dangerous**: Require user approval (e.g., `READ_EXTERNAL_STORAGE`).
- **Special**: System-managed (e.g., `SYSTEM_ALERT_WINDOW`). FileViewer declares `READ_EXTERNAL_STORAGE` in `AndroidManifest.xml`.

Q17: How do you declare and request permissions in Java for FileViewer?

Explanation:

- **Declaration**: In `AndroidManifest.xml`:

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

- **Runtime Request**: In `MainActivity.java` (Java equivalent of `MainActivity.kt`):

```
if (ContextCompat.checkSelfPermission(this,
Manifest.permission.READ_EXTERNAL_STORAGE) !=
PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(this, new String[]
{Manifest.permission.READ_EXTERNAL_STORAGE}, STORAGE_PERMISSION_CODE);
} else {
    loadFiles();
}
```

FileViewer checks permission before loading files, requesting it if needed.

Q18: How do you handle permission results in Java?

Explanation: Override `onRequestPermissionsResult`:

```
@Override
public void onRequestPermissionsResult(int requestCode, String[] permissions,
int[] grantResults) {
    super.onRequestPermissionsResult(requestCode, permissions, grantResults);
    if (requestCode == STORAGE_PERMISSION_CODE && grantResults.length > 0 &&
grantResults[0] == PackageManager.PERMISSION_GRANTED) {
        loadFiles();
    } else {
        Toast.makeText(this, "Storage permission denied",
Toast.LENGTH_SHORT).show();
    }
}
```

FileViewer loads files if granted, else shows a denial message.

Q19: What is `requestLegacyExternalStorage`, and why is it used in FileViewer?

Explanation: `requestLegacyExternalStorage` (set in `AndroidManifest.xml`) enables pre-Android 11 file access for API 30, bypassing scoped storage. In Java, it's declared as:

```
<application android:requestLegacyExternalStorage="true">
```

FileViewer uses it to simplify `File` API access, though deprecated in API 33+.

Q20: How would you adapt FileViewer's permissions for scoped storage in Java?

Explanation: Scoped storage (API 30+) restricts direct `File` access. Use `Storage Access Framework` or `MediaStore`. In Java:

```
Intent intent = new Intent(Intent.ACTION_OPEN_DOCUMENT_TREE);
startActivityForResult(intent, REQUEST_CODE);
```

Override `onActivityResult` to handle the selected directory. FileViewer could replace `File` API with this for modern compliance.

Q21: What are common permission-related pitfalls in Java?

Explanation:

- Not checking permission before access (causes crashes).
- Missing `onRequestPermissionsResult` handling.
- Over-requesting permissions, reducing user trust. FileViewer avoids these by checking permissions and providing clear denial feedback.

5. Build Configuration

Q22: What is Gradle, and how is it used in FileViewer?

Explanation: Gradle is Android's build system. FileViewer uses Kotlin DSL in `build.gradle.kts` files for app configuration and `libs.versions.toml` for dependency management.

Q23: Why move repositories to `settings.gradle.kts`?

Explanation: Gradle 7.0+ prefers centralized repository declarations in `settings.gradle.kts` to avoid conflicts. FileViewer fixed a sync error by defining `google()` and `mavenCentral()` there.

Q24: What is `libs.versions.toml`, and why use it?

Explanation: A Gradle catalog file centralizing dependency versions. FileViewer uses it to manage versions (e.g., `androidxCoreKtx=1.13.1`), ensuring consistency and easy updates.

Q25: How do you configure `minSdk`, `compileSdk`, and `targetSdk`?

Explanation: Set in `build.gradle.kts`:

- `minSdk`: Minimum supported API (30 in FileViewer).
- `compileSdk`: API for compilation (35).
- `targetSdk`: Highest tested API (35). These balance compatibility and modern features.

Q26: What is the purpose of the `clean` task in FileViewer?

Explanation: Defined in project-level `build.gradle.kts`, it deletes the `build` directory to reset build artifacts, aiding in resolving build issues.

6. Performance and Optimization

Q27: How does FileViewer optimize `RecyclerView` performance?

Explanation: Uses `ListAdapter` with `DiffUtil` for minimal view updates and view recycling. Avoids heavy operations in `onBindViewHolder`.

Q28: What are memory leaks, and how can they be avoided in FileViewer?

Explanation: Memory leaks occur when objects (e.g., `Activity`) are retained after their lifecycle. FileViewer avoids leaks by not holding static references to views or contexts.

Q29: How would you optimize file loading in FileViewer?

Explanation: Load files asynchronously using `AsyncTask` or `Coroutines` (in Kotlin). Cache file metadata to reduce `File` API calls. FileViewer could improve by adding async loading.

Q30: What is ProGuard, and why is it disabled in FileViewer's release build?

Explanation: ProGuard shrinks and obfuscates code. FileViewer disables it (`isMinifyEnabled=false`) for simplicity but could enable it for production to reduce APK size.

Q31: How do you profile FileViewer's performance?

Explanation: Use Android Studio's Profiler to monitor CPU, memory, and network. Identify bottlenecks in file loading or `RecyclerView` rendering.

7. Advanced Android Concepts

Q32: What is the ViewModel, and how could FileViewer use it?

Explanation: `ViewModel` (from `androidx.lifecycle`) persists UI data across configuration changes. FileViewer could use it to store `fileList`, replacing the `mutableListOf` in `MainActivity`.

Q33: How would you implement navigation in FileViewer using Jetpack Navigation?

Explanation: Define a `NavGraph` with fragments for directory levels. Replace `MainActivity`'s manual list updates with navigation actions, improving back-stack handling.

Q34: What is WorkManager, and how could it enhance FileViewer?

Explanation: `WorkManager` schedules background tasks. FileViewer could use it for periodic file indexing or batch deletes, ensuring reliable execution.

Q35: How would you unit test FileViewer's `FileAdapter`?

Explanation: Use JUnit and Mockito to test `FileAdapter`'s `onBindViewHolder` logic. Mock `FileModel` and verify view updates (e.g., `fileName.text`).

Q36: What is Dependency Injection, and how could FileViewer benefit?

Explanation: DI (e.g., Hilt) provides dependencies externally. FileViewer could use Hilt to inject a file repository into `MainActivity`, improving testability.

8. Project-Specific Questions

Q37: Why did FileViewer remove data binding?

Explanation: Persistent `Unresolved reference: databinding` errors (e.g., `ItemFileBinding`) led to switching to `findViewById` in `FileAdapter.kt` and `MainActivity.kt` for reliability.

Q38: How does FileViewer handle directory expand/collapse?

Explanation: `MainActivity.kt` toggles `FileModel.isExpanded`, updating `fileList` with child files. `FileAdapter.kt` animates icon rotation and indents directories.

Q39: Why use `minSdk=30` in FileViewer?

Explanation: Simplifies storage access with `requestLegacyExternalStorage`. Lowering `minSdk` would require scoped storage adaptations, increasing complexity.

Q40: How would you make FileViewer compatible with API 33+?

Explanation: Replace `File` API with `Storage Access Framework` or `MediaStore`. Remove `requestLegacyExternalStorage`. Update permission requests for `READ_MEDIA_*`.

Q41: What are the limitations of FileViewer's search functionality?

Explanation: Only filters by name, not content or metadata. Lacks recursive search. Could enhance with full-text search or metadata indexing.

9. Architecture and Design

Q42: What is MVVM, and how could FileViewer adopt it?

Explanation: MVVM separates UI (`View`), data (`ViewModel`), and business logic (`Model`). FileViewer could use `ViewModel` for `fileList` and a repository for file operations.

Q43: How would you modularize FileViewer?

Explanation: Split into modules: `app` (UI), `data` (file access), `domain` (business logic). Improves scalability and testability.

Q44: What is Clean Architecture, and why is it relevant?

Explanation: Clean Architecture separates layers (presentation, domain, data). FileViewer could benefit for maintainability, especially with added features like file copying.

Q45: How do you handle configuration changes in FileViewer?

Explanation: Currently, `fileList` is lost on rotation. Using `ViewModel` or `onSaveInstanceState` would persist state.

Q46: What is the role of `FileDiffCallback` in FileViewer?

Explanation: Extends `DiffUtil.ItemCallback` to compare `FileModel` items, ensuring efficient `RecyclerView` updates during list changes.

10. Debugging and Maintenance

Q47: How did you resolve the Gradle sync error in FileViewer?

Explanation: Moved `google()` and `mavenCentral()` to `settings.gradle.kts`, fixing `InvalidUserCodeException` due to repository declarations in `build.gradle.kts`.

Q48: How do you debug FileViewer's UI issues?

Explanation: Use Layout Inspector to verify `ConstraintLayout` constraints and Logcat for runtime errors. Check `item_file.xml` for ID mismatches.

Q49: What tools do you use for code quality in FileViewer?

Explanation: Lint for static analysis, KtLint for Kotlin formatting, and Detekt for code smells. Ensures maintainable code.

Q50: How would you add logging to FileViewer for production?

Explanation: Use Timber or Logback with level-based logging. Log file operations and errors, avoiding sensitive data exposure.

Conclusion

These questions evaluate a candidate's ability to develop, optimize, and maintain an Android app like FileViewer. From basic UI and permissions to advanced architecture and debugging, they cover the full spectrum of Android development. The detailed focus on **permissions in Java** highlights their critical role in secure, user-friendly apps.