### **1. Android Fundamentals**

#### **Activities and Intents**

**Activity Lifecycle**:

* **onCreate**: Called when the activity is first created. Initialize your activity here, set the content view, and bind views.
* **onStart**: The activity becomes visible to the user.
* **onResume**: The activity starts interacting with the user. It's at the top of the activity stack.
* **onPause**: Called when the activity is partially obscured by another activity (dialog or new activity). Use this method to pause ongoing tasks.
* **onStop**: The activity is no longer visible to the user.
* **onDestroy**: Called before the activity is destroyed. Clean up resources here.
* **onRestart**: Called after onStop when the activity is being restarted.

**Intents**:

**Explicit Intents**: Used to start a specific activity within your application.  
java  
Copy code  
Intent intent = new Intent(CurrentActivity.this, TargetActivity.class);

startActivity(intent);

**Implicit Intents**: Used to start an activity from another application or the same app.  
java  
Copy code  
Intent intent = new Intent(Intent.ACTION\_VIEW);

intent.setData(Uri.parse("http://www.example.com"));

startActivity(intent);

**Passing Data**: Using putExtra and getStringExtra methods.  
java  
Copy code  
Intent intent = new Intent(CurrentActivity.this, TargetActivity.class);

intent.putExtra("key", "value");

startActivity(intent);

#### **Fragments**

**Fragment Lifecycle**:

* **onAttach**: Called when the fragment is first attached to its context.
* **onCreate**: Called to initialize the fragment.
* **onCreateView**: Inflate the fragment's layout.
* **onViewCreated**: Called after the view is created.
* **onActivityCreated**: Called when the activity's onCreate method has returned.
* **onStart**: Fragment is visible.
* **onResume**: Fragment is visible and interacting with the user.
* **onPause**: Called when the fragment is no longer interacting with the user.
* **onStop**: Fragment is no longer visible.
* **onDestroyView**: Cleanup resources related to the view.
* **onDestroy**: Cleanup fragment resources.
* **onDetach**: Fragment is detached from its context.

**Communication**:

* Fragments communicate with each other through the hosting activity using interfaces.

### **2. Layouts and Views**

#### **XML Layouts**

**LinearLayout**: Arranges elements linearly (horizontally or vertically).  
xml  
Copy code  
<LinearLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent"

android:orientation="vertical">

<!-- Views here -->

</LinearLayout>

**RelativeLayout**: Position elements relative to each other or the parent.  
xml  
Copy code  
<RelativeLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<!-- Views here -->

</RelativeLayout>

**ConstraintLayout**: Most powerful layout for complex UIs.  
xml  
Copy code  
<ConstraintLayout

android:layout\_width="match\_parent"

android:layout\_height="match\_parent">

<!-- Views here -->

</ConstraintLayout>

**RecyclerView**:

* Replaces ListView and GridView.

Requires a LayoutManager, Adapter, and ViewHolder.  
java  
Copy code  
RecyclerView recyclerView = findViewById(R.id.recycler\_view);

recyclerView.setLayoutManager(new LinearLayoutManager(this));

recyclerView.setAdapter(new MyAdapter(data));

#### **Custom Views**

Extend View or ViewGroup and override methods like onDraw and onMeasure.  
java  
Copy code  
public class MyCustomView extends View {

public MyCustomView(Context context) {

super(context);

}

@Override

protected void onDraw(Canvas canvas) {

super.onDraw(canvas);

// Draw custom view here

}

}

### **3. Networking and Data Storage**

#### **Networking**

**Retrofit**: Create interfaces representing API endpoints.  
java  
Copy code  
public interface ApiService {

@GET("users")

Call<List<User>> getUsers();

}

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("https://api.example.com")

.addConverterFactory(GsonConverterFactory.create())

.build();

ApiService service = retrofit.create(ApiService.class);

Call<List<User>> call = service.getUsers();

**Volley**: For simpler networking operations.  
java  
Copy code  
RequestQueue queue = Volley.newRequestQueue(this);

String url = "https://www.example.com";

StringRequest stringRequest = new StringRequest(Request.Method.GET, url,

new Response.Listener<String>() {

@Override

public void onResponse(String response) {

// Handle response

}

}, new Response.ErrorListener() {

@Override

public void onErrorResponse(VolleyError error) {

// Handle error

}

});

queue.add(stringRequest);

**OkHttp**: For more control over network operations.  
java  
Copy code  
OkHttpClient client = new OkHttpClient();

Request request = new Request.Builder()

.url("https://www.example.com")

.build();

client.newCall(request).enqueue(new Callback() {

@Override

public void onFailure(Call call, IOException e) {

// Handle failure

}

@Override

public void onResponse(Call call, Response response) throws IOException {

// Handle response

}

});

#### **Data Storage**

**SharedPreferences**: For simple key-value pairs.  
java  
Copy code  
SharedPreferences sharedPreferences = getSharedPreferences("MyPrefs", Context.MODE\_PRIVATE);

SharedPreferences.Editor editor = sharedPreferences.edit();

editor.putString("key", "value");

editor.apply();

**SQLite**: For relational databases.  
java  
Copy code  
SQLiteDatabase db = openOrCreateDatabase("MyDB", MODE\_PRIVATE, null);

db.execSQL("CREATE TABLE IF NOT EXISTS users(id INTEGER PRIMARY KEY, name TEXT);");

db.execSQL("INSERT INTO users (name) VALUES ('John Doe');");

**Room**: Provides an abstraction layer over SQLite.  
java  
Copy code  
@Entity

public class User {

@PrimaryKey

public int id;

public String name;

}

@Dao

public interface UserDao {

@Insert

void insert(User user);

@Query("SELECT \* FROM user")

List<User> getAll();

}

@Database(entities = {User.class}, version = 1)

public abstract class AppDatabase extends RoomDatabase {

public abstract UserDao userDao();

}

AppDatabase db = Room.databaseBuilder(getApplicationContext(),

AppDatabase.class, "database-name").build();

UserDao userDao = db.userDao();

### **4. Concurrency**

#### **Threads and Handlers**

**Thread**: Create and start a new thread.  
java  
Copy code  
Thread thread = new Thread(new Runnable() {

@Override

public void run() {

// Background task

}

});

thread.start();

**Handler**: Communicate with the main thread.  
java  
Copy code  
Handler handler = new Handler(Looper.getMainLooper());

handler.post(new Runnable() {

@Override

public void run() {

// Update UI

}

});

#### **AsyncTask (Deprecated)**

Used for background operations.  
java  
Copy code  
private class MyAsyncTask extends AsyncTask<Void, Void, String> {

@Override

protected String doInBackground(Void... voids) {

// Background task

return "Result";

}

@Override

protected void onPostExecute(String result) {

// Update UI with result

}

}

#### **Modern Concurrency**

**Coroutines**: Kotlin's way to handle concurrency.  
kotlin  
Copy code  
GlobalScope.launch {

val result = async { doBackgroundTask() }.await()

withContext(Dispatchers.Main) {

// Update UI

}

}

**WorkManager**: For deferrable, guaranteed background work.  
java  
Copy code  
WorkRequest uploadWorkRequest = new OneTimeWorkRequest.Builder(UploadWorker.class)

.build();

WorkManager.getInstance(context).enqueue(uploadWorkRequest);

### **5. Dependency Injection**

#### **Dagger and Hilt**

**Dagger 2**: Set up modules and components.  
java  
Copy code  
@Module

public class NetworkModule {

@Provides

OkHttpClient provideOkHttpClient() {

return new OkHttpClient();

}

}

@Component(modules = {NetworkModule.class})

public interface AppComponent {

void inject(MyActivity activity);

}

**Hilt**: Simplifies Dagger setup.  
java  
Copy code  
@HiltAndroidApp

public class MyApplication extends Application {}

@AndroidEntryPoint

public class MyActivity extends AppCompatActivity {

@Inject OkHttpClient okHttpClient;

}

@Module

@InstallIn(SingletonComponent.class)

public class NetworkModule {

@Provides

OkHttpClient provideOkHttpClient() {

return new OkHttpClient();

}

}

### **6. Jetpack Components**

#### **ViewModel and LiveData**

**ViewModel**: Store and manage UI-related data.  
java  
Copy code  
public class MyViewModel extends ViewModel {

private MutableLiveData<String> data;

public LiveData<String> getData() {

if (data == null) {

data = new MutableLiveData<>();

loadData();

}

return data;

}

private void loadData() {

// Load data asynchronously

}

}

**LiveData**: Observable data holder.  
java  
Copy code  
myViewModel.getData().observe(this, new Observer<String>() {

@Override

public void onChanged(String s) {

// Update UI

}

});

#### **Navigation Component**

Handle navigation and passing data.  
xml  
Copy code  
<navigation xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:app="http://schemas.android.com/apk/res-auto"

xmlns:tools="http://schemas.android.com/tools"

app:startDestination="@id/homeFragment">

<fragment

android:id="@+id/homeFragment"

android:name="com.example.HomeFragment"

tools:layout="@layout/fragment\_home">

<action

android:id="@+id/action\_homeFragment\_to\_detailFragment"

app:destination="@id/detailFragment" />

</fragment>

<fragment

android:id="@+id/detailFragment"

android:name="com.example.DetailFragment"

tools:layout="@layout/fragment\_detail" />

</navigation>

### **7. Architecture Patterns**

#### **MVC, MVP, MVVM**

**MVC**: Model-View-Controller. The controller handles the logic.  
java  
Copy code  
// Simplified example

public class MainActivity extends AppCompatActivity {

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

MyModel model = new MyModel();

String data = model.getData();

TextView textView = findViewById(R.id.textView);

textView.setText(data);

}

}

**MVP**: Model-View-Presenter. The presenter handles the logic.  
java  
Copy code  
public class MyPresenter {

private MyView view;

private MyModel model;

public MyPresenter(MyView view) {

this.view = view;

this.model = new MyModel();

}

public void loadData() {

String data = model.getData();

view.updateView(data);

}

}

public interface MyView {

void updateView(String data);

}

public class MainActivity extends AppCompatActivity implements MyView {

private MyPresenter presenter;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

presenter = new MyPresenter(this);

presenter.loadData();

}

@Override

public void updateView(String data) {

TextView textView = findViewById(R.id.textView);

textView.setText(data);

}

}

**MVVM**: Model-View-ViewModel. The ViewModel handles the logic.  
java  
Copy code  
public class MyViewModel extends ViewModel {

private MutableLiveData<String> data;

public LiveData<String> getData() {

if (data == null) {

data = new MutableLiveData<>();

loadData();

}

return data;

}

private void loadData() {

// Load data asynchronously

}

}

public class MainActivity extends AppCompatActivity {

private MyViewModel viewModel;

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

viewModel = new ViewModelProvider(this).get(MyViewModel.class);

viewModel.getData().observe(this, new Observer<String>() {

@Override

public void onChanged(String data) {

TextView textView = findViewById(R.id.textView);

textView.setText(data);

}

});

}

}

#### **Clean Architecture**

* **Presentation Layer**: Contains UI components like activities and fragments.
* **Domain Layer**: Contains business logic and use cases.
* **Data Layer**: Handles data retrieval and persistence.

### **8. Testing**

#### **Unit Testing**

**JUnit**: For writing unit tests.  
java  
Copy code  
public class MyUnitTest {

@Test

public void addition\_isCorrect() {

assertEquals(4, 2 + 2);

}

}

**Mockito**: For mocking dependencies.  
java  
Copy code  
public class MyUnitTest {

@Mock

MyDependency dependency;

@Before

public void setUp() {

MockitoAnnotations.initMocks(this);

}

@Test

public void testDependency() {

when(dependency.getData()).thenReturn("Mock Data");

assertEquals("Mock Data", dependency.getData());

}

}

#### **UI Testing**

**Espresso**: For writing UI tests.  
java  
Copy code  
@RunWith(AndroidJUnit4.class)

public class MainActivityTest {

@Rule

public ActivityScenarioRule<MainActivity> activityScenarioRule =

new ActivityScenarioRule<>(MainActivity.class);

@Test

public void testButtonClick() {

onView(withId(R.id.button)).perform(click());

onView(withId(R.id.textView)).check(matches(withText("Hello World!")));

}

}

**Robolectric**: For running Android tests on the JVM.  
java  
Copy code  
@RunWith(RobolectricTestRunner.class)

public class MainActivityTest {

@Test

public void testActivity() {

MainActivity activity = Robolectric.buildActivity(MainActivity.class).create().get();

TextView textView = activity.findViewById(R.id.textView);

assertNotNull(textView);

}

}

### **9. Kotlin**

**Syntax**: Basic syntax and features.  
kotlin  
Copy code  
fun main() {

val name: String = "Kotlin"

println("Hello, $name!")

}

**Extension Functions**: Adding functions to existing classes.  
kotlin  
Copy code  
fun String.isPalindrome(): Boolean {

return this == this.reversed()

}

**Sealed Classes**: Represent restricted class hierarchies.  
kotlin  
Copy code  
sealed class Result {

class Success(val data: String) : Result()

class Error(val error: Throwable) : Result()

}

**Coroutines**: Handling concurrency.  
kotlin  
Copy code  
fun main() = runBlocking {

launch {

delay(1000L)

println("World!")

}

println("Hello,")

}

### **10. Performance and Optimization**

**Memory Leaks**: Identify and fix leaks using tools like LeakCanary.  
java  
Copy code  
// Example of fixing memory leaks

private static class MyHandler extends Handler {

private final WeakReference<MyActivity> mActivity;

MyHandler(MyActivity activity) {

mActivity = new WeakReference<>(activity);

}

@Override

public void handleMessage(Message msg) {

MyActivity activity = mActivity.get();

if (activity != null) {

// Handle message

}

}

}

* **Profiling Tools**: Use Android Profiler to monitor CPU, memory, and network usage.

### **Sample Questions**

#### **Basic Questions**

1. **What is an Activity?**
   * An Activity represents a single screen with a user interface in an Android application.
2. **Explain the Android application lifecycle.**
   * The lifecycle includes states such as onCreate, onStart, onResume, onPause, onStop, onDestroy, and onRestart.
3. **What are Intents? How do you use them?**
   * Intents are messaging objects used to request an action from another app component. They are used to start activities, services, and broadcast receivers.

#### **Intermediate Questions**

1. **What is a ContentProvider and how is it used?**
   * A ContentProvider manages access to a structured set of data and provides mechanisms for defining data security. They are used to share data between applications.
2. **How do you handle configuration changes in Android?**
   * By overriding the onConfigurationChanged method or by retaining fragments and using ViewModel to persist data across configuration changes.
3. **Explain the difference between Service and IntentService.**
   * Service runs in the main thread, while IntentService creates a worker thread to handle all start requests, ensuring that each request is handled in a separate thread.

#### **Advanced Questions**

1. **What is the difference between LiveData and StateFlow in Kotlin?**
   * LiveData is lifecycle-aware and is used primarily for observing UI-related data. StateFlow is a state-holder observable flow and can be used in a broader context outside the lifecycle-aware scope.
2. **How do you implement dependency injection in Android?**
   * Using Dagger or Hilt. Define modules and components to provide dependencies and inject them into classes.
3. **What is the importance of WorkManager and how does it differ from JobScheduler?**
   * WorkManager is a library that simplifies the management of deferrable and guaranteed background work. It provides compatibility down to API Level 14, whereas JobScheduler is available from API Level 21 onwards.