UNIT -4:

Creating Your Own Content Providers - Using the Content Provider, SMS Messaging - Sending Email-Displaying Maps- Getting Location Data- Monitoring a Location

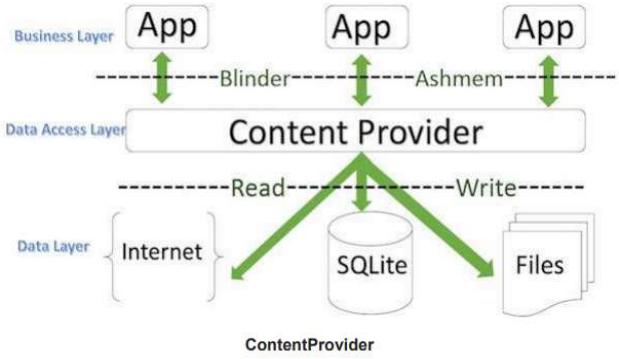
Putting SQL to work - Introducing SQLite, In and Out of SQLite, Hello Database, Data Binding, using content provider, implementing content provider.

Reading/writing local data, accessing the Internal File system, accessing the SD card. Preparing app for publishing, Deploying APK files, uploading in Market, Consuming Web Services Using HTTP-Consuming JSON Services- Creating Your Own Services Binding Activities to Services - Understanding Threading.

4.1 Using the Content Provider:

What is Content provider?

- A content provider component supplies data from one application to others on request. Such requests are handled by the methods of the ContentReslover class. It's a way to share data between different applications securely. A content provider can use different ways to store its data and data can be stored in a database, in files, or even over a network.



Here's a breakdown of the storage options:

1. **Database**: The content provider can store its data in a database, typically a SQLite database. This is a common method because databases are efficient for handling structured data and complex queries.
2. **Files**: Data can also be stored in files. This method might be used for handling unstructured data, such as media files (images, videos) or large datasets.
3. **Network**: A content provider can also store data over a network. This could mean the data is fetched from a remote server or cloud storage.

Android ships with many useful content providers, including the following:

- ➢ Browser—Stores data such as browser bookmarks, browser history, and so on
- ➢ CallLog—Stores data such as missed calls, call details, and so on
- ➢ Contacts—Stores contact details
- ➢ MediaStore—Stores media files such as audio, video, and images
- ➢ Settings—Stores the device's settings and preferences

Content URI is the key concept of Content Providers. To access the data from a content provider, Content URI is used as a query string.

Structure of a Content URI is **Content: //authority /optionalPath /optionalID**

1. **Content:** It represents that the given URI is content URI.
2. **Authority:** Name of the Content provider like contacts, browser, and etc.It must be unique for every content provider.
3. **OptionalPath:** Specifies the type of data provided by the Content provider. For example, if you are getting all the contacts from the **Contacts** content provider, then the data path would be people and URI would look like this **content://contacts/people**
4. **OptionalID:** It is a numeric Value it specifies the specific record requested. For example, example, if you are looking for a contact number 5 in the Contacts content provider then URI would look like this **content: //contacts/people/5.**

Steps To Create Content Provider:

- **Create the Content Provider Class**
- Create a class that extends the ContentProvider base class.
- **Define the Content URI**
- Set up a unique URI (Uniform Resource Identifier) that will be used to access your content.
- **Create the Database**
- Implement a local database (typically SQLite) to store your data.
- Override the onCreate() method of your Content Provider to initialize the database, often using SQLiteOpenHelper.
- Note: When the app starts, Android automatically calls onCreate() for each registered Content Provider on the main thread.
- **Implement CRUD Operations**
- Override methods like query(), insert(), update(), and delete() to interact with the database based on the incoming URI and request.
- **Register the Content Provider**
- Declare your Content Provider in the AndroidManifest.xml using the <provider> tag, specifying attributes like android:authorities, android:exported, and android:name.

Operations in Content provider: The fundamental operations are possible in content Provider namely Create, Read, update, get type, and delete. These operations are often termed as CRUD operations.

1. **OnCreate ( ):** This method is called when the provider is started.

2. **Query ( ):** This method receives a request from a client. The result is returned as a Cursor object.
3. **insert( ):** This method inserts a new record into the content provider.
4. **delete( ):** This method deletes an existing record from the content provider.
5. **update( ):** This method updates an existing record from the content provider.
6. **getType( ):** This method returns the MIME type of the data at the given URI.

Code for inserting a New Contact, update and delete:

```
Uri uri=uri.parse("content://com.android.contacts/contacts");
 ContentValues values = new ContentValues( );
 Values.put("display_name","srikanth");
 Values.put("Phone_number","9623457654");
Uri newUri=resolver.insert (uri, values)
Int rowsUpdated = resolver.update (uri, values,"contact_id=?" , new  String[]{"1"});
updates the Contact with ID  "1"
Int rowsDeleted = resolver.delete (uri, values,"contact_id=?" , new  String[]{"1"});
Deletes the Contact with ID "1"
```

4.2 Sending SMS:

Android provides a convenient way to send SMS with the help of its built-in API SmsManager. To use the API, you need to define it using the following code snippet:

SmsManager API:

```
SmsManager smsManager = SmsManager.getDefault ();

smsManager.sendTextMessage ("Phone number", null, "This is the sms message that you put here", null, null);
```

Five arguments to sendtextmessage () method

- destinationAddress - phone no of recipient
- scAddress - Service Center Address
    - **Service Center Address (SCA)**: This is the address of the Short Message Service Center (SMSC) that handles the delivery of the SMS
- Text – content of the text message
- sentIntent –Pending intent to invoke when the SMS message is sent (handed over to the SMSC).
- deliveryIntent –Pending intent to invoke when the SMS message is delivered to the recipient.

- To implement this, you need to have a SIM inserted in your android smartphone or a number equipped in your Android Emulator. To send a text SMS over the phone using the SMSManager class in an Android Application.

Built-in SMS Application:

- The built-in SMS application on an Android device is the default messaging app provided by the device manufacturer or the Android OS. This app is typically used for managing SMS and MMS messages and provides a user-friendly interface for sending and receiving messages.

Code:

```
protected void sendSMS () {
    Log.i("Send SMS", "");
    Intent smsIntent = new Intent (Intent.ACTION_VIEW);
    smsIntent.setData (Uri.parse ("smsto:"));
    smsIntent.setType ("vnd.android-dir/mms-sms");
    smsIntent.putExtra ("address", new String ("01234"));
   smsIntent.putExtra ("sms_body", "Test ");
   StartActivity (sendIntent);
```

Intent Object - Extra to send SMS

Android has built-in support to add phone number and text message to send an SMS as follows −

```
smsIntent.putExtra("address"  , new String("0123456789;3393993300"));
smsIntent.putExtra("sms_body"  , "Test SMS to Angilla");
```

1. **`Intent(Intent.ACTION_VIEW)`**: This line creates a new `Intent` object with the action `ACTION_VIEW`. Using `ACTION_VIEW` indicates that you want to view some data. In this case, you're preparing to view or interact with SMS or MMS data.
2. **`putExtra("sms_body", "default content")`**: This adds extra data to the `Intent`. Specifically, `"sms_body"` is a key used by many SMS/MMS applications to pre-fill the body of the message that will be sent. `"default content"` is the actual text that will be pre-filled in the message body.
3. **`setType("vnd.android-dir/mms-sms")`**:

- Sets the MIME type of the `Intent`. `vnd.android-dir/mms-sms` is a MIME type used to indicate that the `Intent` is intended to send an SMS or MMS message.
- MIME (Multipurpose Internet Mail Extensions) types are used to specify the nature and format of a file. For example, `text/html` for HTML files, `image/jpeg` for JPEG images, etc.
  - `vnd`: This prefix indicates a vendor-specific MIME type.
  - `android-dir`: Specifies that this MIME type is defined by Android.
  - `mms-sms`: Indicates that the MIME type is related to messaging, particularly MMS (Multimedia Messaging Service) and SMS (Short Message Service).

4. **`startActivity (sendIntent)`**: Finally, `startActivity()` method is called with `sendIntent` to start the activity that can handle the `Intent`. This will typically launch the default SMS/MMS application installed on the device, or prompt the user to choose which application to use if multiple are installed.

4.3 Sending E-mail:

Sending Emails can also be implemented using Android. Email is just like a message that can be sent from one user to another user via a network. In Android, you can send an email from your application without implementing it, just by using the default Email app provided by android like Email, Gmail, and Outlook etc. There are different Intent objects required to send an email that is described below:

Action required to send Email: To launch an email client on your android device, you will have to use ACTION_SEND action. The syntax given below is used to create intent:

```
Intent mailAction = new Intent(Intent.ACTION_SEND);
```

Type to send mail: As we are familiar with sending mail where you need to specify where you need to send the mail, i.e. 'mailto:' Here we provide the URI of the recipient with the help of the setData () method. You can also follow the code snippet shown below for your reference where we also defined the setType () method:

```
emailIntent.setData (Uri.parse ("mailto :"));
emailIntent.setType("text/plain");
```

Attachments in the mail or other extras: Android comes with default support where you can use TO, SUBJECT, CC, BCC etc. fields in your mail without sending it to the main recipient. Refer to the example below for a better understanding:

```
emailIntent.putExtra (Intent.EXTRA_EMAIL, new String[]{"Recipient_Add"});
emailIntent.putExtra(Intent.EXTRA_SUBJECT, "Subject_Here");
emailIntent.putExtra (Intent.EXTRA_TEXT   , "This is the main body of the mail.");
```

Step By Step sending an implementation of sending a email:

Step: 1 Project Setup

Create an android project by selecting "Empty Views activity" and naming the project "email-example" and Choosing java as a language.

Step: 2 Create the UI

Design the UI to input the recipients email address, subject, and message body. This UI will allow users to enter the necessary details to send an email.

activity main.xml code:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
```

```
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp">

<EditText
    android:id="@+id/email_address"
    android:layout width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Email Address"
    android: inputType="textEmailAddress" />

    <EditText
    android:id="@+id/email_subject"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Subject"
    android: inputType="text" />


    <EditText
    android:id="@+id/email_message"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:hint="Message"
    android: inputType="textMultiline" />

    <Button
    android:id="@+id/send_email"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="Send Email" />

    </LinearLayout>
```

Explanation:

* The LinearLayout' provides a vertical layout for the Ul components.
* The 'EditText' fields allow the user to input the recipient's email address, subject, and message content.
* The "Button' triggers the email-sending functionality when clicked.

Step3. Implement Main Activity

Implement the activity to handle sending emails using intents. This activity manages the user Interactions and handles the email-sending process.

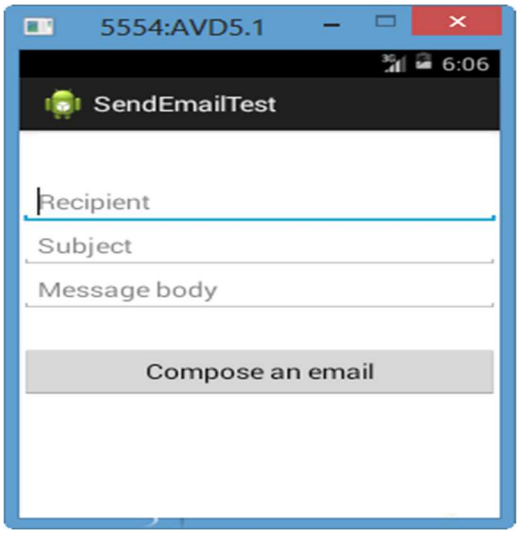
```
public class MainActivity extends AppCompatActivity (
private EditText emailAddress;
private EditText emailsubject;
private EditText emailMessage;
private Button sendEmail;
@Override
protected void onCreate(Bundle savedInstanceState)
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 emailAddress = findViewById(R.id.email_address);
 emailSubject = findViewById(R.id.email_subject);
 emailMessage = findViewById(R.id.email_message);
 sendEmail = findViewById(R.id.send_email);
 sendEmailButton.setOnClickListener(new View.OnClickListener() {
 @Override
  public void onClick(View v) {
   sendEmail();

                }
      });
}

private void sendEmail() {

String emailAddress = emailAddress.getText().toString();
String subject = emailSubject.getText().toString();
String message = emailMessageEditText.getText().toString();

Intent emailIntent = new Intent(Intent.ACTION_SEND);
emailIntent.setType("message/rfc822");
emailIntent.putExtra (Intent.EXTRA_EMAIL, new String[] {emailAddress});
emailIntent.putExtra(Intent.EXTRA_SUBJECT, subject);
emailIntent.putExtra (Intent.EXTRA_TEXT, message);
if (emailIntent.resolveActivity(getPackageManager()) != null) {
      startActivity(emailIntent);
 }
else {
      Toast.makeText (this, "Email Client Not Installed", Toast.LENGTH_SHORT).show ();
      }
}
```



Displaying Maps:

Android provides facility to integrate Google map in our application. Google map displays your current location, navigate location direction, search location etc. It allows users to interact with dynamic maps that provide detailed geographical information.Users can view their current location, search for specific places, get directions for navigation, and explore points of interest.

Key points when integrating Google Maps in an Android Application:

1. **API Key:** It is essential for utilizing Google Maps services within the application. This key is a unique identifier and authorization token for accessing the mapping functionalities.
2. **Permissions:** Configuring the necessary permissions in the AndroidManifest.xml file is crucial to enable the application to access location data and connect to the internet. Permissions like ACCESS_FINE_LOCATION' and 'INTERNET are commonly required for Google Maps integration.
3. **Google Maps SDK**: Google Maps SDK provides access to mapping tools, location services, and interactive functionalities within the application.

Step 1: Set Up Your Project

1. **Create a new project** in Android Studio or open an existing one.
2. **Add Google Play services** to your project by modifying your build.gradle files.

Step 2: Get a Google Maps API Key

1. Go to the Google Cloud Console.
2. Create a new project or select an existing project.
3. Enable the "Maps SDK for Android" API for your project.
4. Create an API key and restrict it to your Android app by specifying your app's package name and SHA-1 certificate fingerprint.

Step 3: Add the API Key to Your Project

Add the API key to your AndroidManifest.xml file:

<meta-data

```
android: name="com.google.android.geo.API_KEY"
android:value="YOUR_API_KEY_HERE"/>
```

Step 4: Add a Map Fragment to Your Layout

In your res/layout folder, create an XML layout file (e.g., activity_main.xml) and add a MapFragment or SupportMapFragment:

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

Step 5: Initialize the Map in Your Activity

In your MainActivity.java or MainActivity.java file, implement OnMapReadyCallback and initialize the map:

```
public class MainActivity extends FragmentActivity implements OnMapReadyCallback {
private GoogleMap mMap;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate (savedInstanceState);
      SetContentView (R.layout.activity_main);

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

   @Override
   public void onMapReady(GoogleMap googleMap) {
      mMap = googleMap;

      // Add a marker and move the camera
      LatLng sydney = new LatLng (-34, 151);
      mMap.addMarker (new MarkerOptions ().position (sydney).title("Marker in Sydney"));
      mMap.moveCamera (CameraUpdateFactory.newLatLng (sydney));
   }
}
```

Google Maps provide various types; each of these map types provides a unique way of viewing and interacting with the Map.

1. **Normal View:** This is the Standard roadmap view, which displays streets, road names and geographical features.
   googleMap.setMapType (GoogleMap.Map_TYPE_NORMAL);
2. **Satellite View:** This view displays the satellite images of the area, showing detailed and realistic images of the terrain and buildings.
   googleMap.setMapType (GoogleMap.Map_TYPE_SATELLITE);
3. **Terrain View:** This View emphasizes terrain information, showing topography and physical landscape features such as mountains, valleys, and bodies of water.

googleMap.setMapType (GoogleMap.Map_TYPE_TERRAIN);

4. **Hybrid View:** This View Combines the normal roadmap view with satellite images, providing street names and geographical features overlaid on real world imagery.
   googleMap.setMapType (GoogleMap.Map_TYPE_HYBRID);
5. **None:** Displays no base map tiles, allowing developers to overlay custom data on a blank canvas.
   googleMap.setMapType (GoogleMap.Map_TYPE_NONE);

Getting Location data:

Mobile devices today are equipped with multiple technologies to determine their location. These technologies enable various applications, such as navigation, location-based services, and geotagging, to function accurately and efficiently.

Location Manager in Android

On Android devices, the LocationManager class is used to determine the user's physical location. It allows the device to access different location providers such as GPS, network (cell tower and Wi-Fi). And passive providers to get the best available location data.

The primary technologies used to determine a device's location include GPS, cell tower triangulation and Wi-Fi triangulation.

Each of these technologies has its strengths and weakness

1. **GPS (Global Positioning System):**

Functionality: Utilizes satellites orbiting the Earth to pinpoint the device's location

Advantages: Highly accurate, especially in outdoor settings with a clear sky.

Disadvantages: Requires an unobstructed line of sight to satellites, rendering it Ineffective indoors or in areas where satellite signals are blocked (eg, tunnels, dense urban regions).

2. **Cell Tower Triangulation:**

Functionality: Estimates the device's location by identifying nearby cell towers. When a mobile phone is active, it continually communicates with surrounding base stations (cell towers).

Advantages: Effective indoors and doesn't rely on satellite signals. Particularly useful in densely populated areas with many cell towers.

Disadvantages: Less accurate than GPS, as accuracy depends on the density and overlapping coverage of cell towers.

3. **Wi-Fi Triangulation:**

Functionality: Connects to a Wi-Fi network and cross-references the service provider's information with databases to determine the location served by the provider.

Advantages: Effective indoors and usable when cell tower signals are unavailable

Disadvantages: Accuracy depends on the availability and proximity of Wi-Fi networks.

Monitoring a Location:

- To monitor a specific location in Android, developers can use the 'addProximityAlert() method provided by the LocationManager' class. This method sets up a proximity alert for a defined geographic point and triggers a specified action when the device enters or exits the proximity of that location.

The 'addProximityAlert ()' method takes the following parameters:

- ✓ **Latitude:** The latitude of the monitored location.
- ✓ **Longitude:** The longitude of the monitored location.
- ✓ **Radius:** The radius in meters around the location within which the alert will be triggered.
- ✓ **Expiration:** Time for the proximity alert to be valid, after which it will be deleted.
- ✓ **PendingIntent:** The intent to be triggered when the user enters or exits the defined location.

Android SQLite

SQLite is a C-language library that provides a relational database management system i.e. used to perform database operations on android devices such as storing, manipulating or retrieving persistent data from the database. It is self-contained, serverless, zero-configuration, and transactional.In Android, it is the default database engine used for local data storage.

In and Out of SQLite

Creating a Database and Tables

To create a database and tables in SQLite within an Android application, you typically use the `SQLiteOpenHelper` class.

Database Helper Class: Define the helper class for database creation and management.

```
public class MyDatabaseHelper extends SQLiteOpenHelper {
   private static final String DATABASE_NAME = "mydatabase.db";
   private static final int DATABASE_VERSION = 1;

   public MyDatabaseHelper(Context context) {
      super(context, DATABASE_NAME, null, DATABASE_VERSION);
   }

   @Override
   public void onCreate(SQLiteDatabase db) {
      String CREATE_TABLE_USERS = "CREATE TABLE users (id INTEGER PRIMARY
KEY, name TEXT, age INTEGER)";
      db.execSQL(CREATE_TABLE_USERS);
   }

   @Override
   public void onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion) {
      db.execSQL("DROP TABLE IF EXISTS users");
      onCreate(db);
```

```
    }
}
```

Inserting Data

To insert data into the database:

```
public void insertUser(String name, int age) {

    SQLiteDatabase db = this.getWritableDatabase();

    ContentValues values = new ContentValues();

    values.put("name", name);

    values.put("age", age);

    db.insert("users", null, values);

    db.close();

}
```

What is Data Binding?

Data Binding is a feature in Android that allows you to bind UI components in your layouts to data sources in your app using a declarative format rather than programmatically.

Enable Data Binding: Modify the `build.gradle` file to enable data binding.

- **Create a Layout with Data Binding**: Wrap the root layout element in a `<layout>` tag and define data variables.
- **Create a Data Model**: Create a class that represents the data to be bound to the layout.
- **Bind Data in Activity**: Use `DataBindingUtil` to set up data binding in the activity and bind the data model to the layout.

1. Enable Data Binding

In your project's `build.gradle` file, enable data binding:

```
android {
    ...
    buildFeatures {
        dataBinding true
    }
}
```

2. Create a Layout with Data Binding

Modify your layout XML file to use data binding by wrapping the root layout element in a `<layout>` tag. Define a data variable that will be bound to the layout.

activity_main.xml:

```
<layout xmlns:android="http://schemas.android.com/apk/res/android">
    <data>
        <variable
            name="user"
            type="com.example.myapp.User" />
    </data>
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:padding="16dp">

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{user.name}" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@{String.valueOf(user.age}" />

    </LinearLayout>
</layout>
```

3. Create a Data Model

Create a data model class that will be used in the data binding.

User.java:

```
package com.example.myapp;

public class User {
    private String name;
    private int age;

    public User(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
```

```
    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

4. Bind Data in Activity

In your activity, set up data binding and bind the data model to the layout.

MainActivity.java:

```
        // Set up data binding
    ActivityMainBinding binding=DataBindingUtil.setContentView (this, R.layout.activity_main);

        // Create a User object
        User user = new User("John Doe", 30);

        // Bind the User object to the layout
        binding.setUser(user);
    }
}
```

Mention the different event handlings?

In Android development, event handling refers to the mechanism of capturing and responding to user interactions or system events within an application. Here are some common types of event handling techniques used in Android:

- **Button Click Event Handling**:
    - Handles user clicks on buttons or other clickable views.
    - Defined in XML using the `android:onClick` attribute or programmatically using `setOnClickListener`.

1. Button Click Event Handling

This is the most common type of event handling where you handle user clicks on buttons or other clickable views.

- **XML Layout (activity_main.xml)**:

```
<Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
```

```
        android:text="Click Me"
        android:onClick="onButtonClick" />
```

- **Activity Code (MainActivity.java)**:

```
    public void onButtonClick(View view) {
        // Handle button click here
// Example: Toast.makeText(this, "Button Clicked", Toast.LENGTH_SHORT).show();
    }
```

- **Menu Item Click Event Handling**:
  - Handles clicks on options menu items or action bar items.
  - Implemented in the activity using `onOptionsItemSelected` method to detect menu item selection.

- **Touch Event Handling**:
  - Captures touch gestures like tap, swipe, and pinch.
  - Implemented by overriding `onTouchEvent` method in custom views or gesture detectors (`GestureDetector`).

- **List Item Click Event Handling**:
  - Handles clicks on items in a `ListView`, `RecyclerView`, or similar list-based views.
  - Implemented using `OnItemClickListener` for `ListView` or `RecyclerView` click listeners for `RecyclerView`.

- **Text Change Listener**:
  - Monitors changes to text input fields like `EditText`.
  - Implemented using `TextWatcher` interface methods (`beforeTextChanged`, `onTextChanged`, `afterTextChanged`) to track text changes.

- **Broadcast Receiver**:
  - Listens for system-wide broadcast messages sent by the system or other apps.
  - Registered in the manifest file or dynamically in the activity to handle specific broadcast intents (`IntentFilter`).

Mention the contents of an android package kit files.

The Android Package Kit (APK) file is the package file format used by the Android operating system for distribution and installation of mobile apps, including all the assets, resources, and compiled code necessary for the app to run on a device. The contents of an APK file typically include the following components:

1. **AndroidManifest.xml**:
    - The manifest file describes essential information about the app to the Android system, such as permissions required, hardware and software features used, components (activities, services, receivers, providers), and more.
2. **classes.dex**:
    - This file contains the compiled bytecode of the application's Java or Kotlin code. It includes all the classes and methods needed for the app to run on the Android platform.
3. **Resources**:
    - This directory contains all non-code resources used by the application, such as XML layouts (`res/layout/`), drawable images (`res/drawable/`), string values (`res/values/`), and other resources (`res/raw/`).
4. **Assets**:
    - This directory contains raw asset files that the application may need to access directly using input streams. Unlike resources, assets are not compiled into the binary format.
5. **META-INF**:
    - This directory contains the manifest file (`MANIFEST.MF`) and signature files (`CERT.RSA`, `CERT.SF`, `MANIFEST.MF`) used for signing and verifying the integrity of the APK.
6. **Libraries (Optional)**:
    - If the application includes native libraries for specific architectures (e.g., ARM, x86), these libraries are stored in separate directories (`lib/` or `lib/<architecture>/`) within the APK.
7. **Resources.arsc**:
    - This binary file contains compiled resources and is used by the Android system to efficiently load resources without parsing XML files at runtime.
8. **Android Resources Pack (ARSC)**:
    - A compiled file containing Android application resources (e.g., strings, styles, drawables).
9. **Manifest.xml**:
    - Configuration file that contains information about application components.

Simple Ways to Access the SD Card (External Storage) in Android

1. **Using Environment.getExternalStorageDirectory()**:

    This method provides access to the primary external storage directory. It's straightforward but note its behavior changes on Android 10 (API level 29) and above.

    ```
    File externalStorageDir = Environment.getExternalStorageDirectory();
    ```

    Note: On newer Android versions (API level 29+), this method returns a directory specific to your app, not shared storage. For shared storage, consider using Storage Access Framework (SAF).

2. **Using Storage Access Framework (SAF)**:

   SAF allows users to access documents and other files across different storage providers through a system UI. This is essential for Android 10 and above due to scoped storage restrictions.

   - **Opening a Directory Picker**:

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

   - **Accessing Documents**:

     Once a directory is chosen, you can access files using the content URI returned by SAF.

3. **Permissions**:

   Ensure your app has the necessary permissions declared in the manifest:

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

   Starting from Android 10, you also need to handle permissions for scoped storage and use SAF for accessing files outside your app's sandbox.