1)How can you create a custom list item layout and adapter?

LISTVIEW IN ANDROID

ListView is used when you have to show items in a vertically scrolling list. Best example of it is our device's Contact List. With ListView, user can easily browse the information, while scrolling up and down. You can set divider between every item and set its height and coloredg as per your UI design.

Inside a ListView, we can show list of Text items by using TextView, or pictures using ImageView, or any other view or a combination of views.

As ListView is generally used to display a large set of data, hence it is not feasible to manually create list items for the complete data, hence Android provides us with special Adapter classes which can be used to supply data from datasets to ListView. Following are some of the main attributes that are most commonly used:

Attribute	Description
android:divider	Using this attribute we can specify a divider between List items. A drawable or any color can be specified as a value for this attribute.
android:dividerHeight	Used to specify height of the divider.

Below we have shown how you can add a ListView to your android app using the layout XML.

SAMPLE CODE:

<*ListView*

```
android:id="@+id/listView"
android:layout_width="match_parent"
android:layout_height="match_parent"
android:divider="@android:color/black"
android:dividerHeight="1dp"/>
```

Adapter and Adapter View

Adapter and Adapter View are so popular, that every time you see any app with a List of items or Grid of items, you can say for sure that it is using Adapter and Adapter View.

Generally, when we create any List or Grid of data, we think we can use a loop to iterate over the data and then set the data to create the list or grid.

But what if the data is a set of 1 million products. Then using a loop will not only consume a lot of time, making the app slow, also it might end up eating all the runtime memory.

All these problems are solved by using Adapter and Adapter View.Adapter View, is a **View** object, and can be used just like we use any other interface widget. The only catch here is, that it needs an **Adapter** to provide content to it as it is incapable of displaying data on its own.

What is an Adapter?

An adapter acts like a bridge between a data source and the user interface. It reads data from various data sources, coverts it into View objects and provide it to the linked Adapter view to create UI components.

The data source or dataset can be an Array object, a List object etc.

You can create your own Adapter class by extending the BaseAdapter class, which is the parent class for all other adapter class. Android SDK also provides some ready-to-use adapter classes, such as ArrayAdapter, SimpleAdapter etc.

What is an Adapter View?

An Adapter View can be used to display large sets of data efficiently in form of List or Grid etc, provided to it by an Adapter.

An Adapter View is capable of displaying millions of items on the User Interface, while keeping the memory and CPU usage very low and without any noticeable lag. Different Adapters follow different strategies for this, but the default Adapter provided in Android SDK follow the following tricks:

1. It only renders those **View** objects which are currently on-screen or are about to some on-screen. Hence no matter how big your data set is, the Adapter View will always load only 5 or 6 or maybe 7 items at once, depending upon the display size. Hence saving memory.

2. It also reuses the already created layout to populate data items as the user scrolls, hence saving the CPU usage.

Suppose you have a dataset, like a String array with the following contents.

```
String days[] = {"Monday", "Tuesday", "Wednesday",
"Thursday", "Friday", "Saturday", "Sunday"};
```

Now, what does an **Adapter** do is that it takes the data from this array and creates a View from this data and then, it gives this **View** to an **AdapterView**.

The AdapterView then displays the data in the way you want.

2)Explain the Date Picker and Time Picker views in Android with Example.

PICKER VIEW IN ANDROID

Android provides controls for the user to pick a time or pick a date as ready-touse dialogs. Each picker provides controls for selecting each part of the time(hour, minute, AM/PM) or date (month, day, year). Using these pickers helps ensure that your users can pick a time or date that is valid, formatted correctly, and adjusted to the user's locale.

1. DatePicker

The DatePicker allows the user to select a date (year, month, day). It can be used

either as a widget in a layout or within a DatePickerDialog.

SAMPLE CODE:

```
<DatePicker
android:id="@+id/datePicker"
android:layout_width="wrap_content"15
android:layout_height="wrap_content"
android:datePickerMode="spinner"/>
```

2. TimePicker

The TimePicker allows users to select time (hours and minutes). Like the DatePicker, it can be a widget or a dialog.

```
<TimePicker
```

```
android:id="@+id/timePicker"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
android:timePickerMode="spinner"
android:is24HourView="true" />
```

SAMPLE CODE:

```
class MainActivity : AppCompatActivity() {
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
val selectDateTimeButton: Button = findViewById(R.id.select_datetime_btn)
val dateTimeText: TextView = findViewById(R.id.datetime_text)
selectDateTimeButton.setOnClickListener {
val calendar = Calendar.getInstance()
val year = calendar.get(Calendar.YEAR)
val month = calendar.get(Calendar.MONTH)
val day = calendar.get(Calendar.DAY_OF_MONTH)
val hour = calendar.get(Calendar.HOUR_OF_DAY)
val minute = calendar.get(Calendar.MINUTE)
val datePickerDialog = DatePickerDialog(this,16
{ _, selectedYear, selectedMonth, selectedDay ->
val timePickerDialog = TimePickerDialog(this,
{ _, selectedHour, selectedMinute ->
```

```
val selectedDateTime = "$selectedDay/${selectedMonth +
1}/$selectedYear $selectedHour:$selectedMinute"
dateTimeText.text = selectedDateTime
}, hour, minute, true)
timePickerDialog.show()
}, year, month, day)
datePickerDialog.show()
}
}
}
```

The Button triggers both DatePickerDialog and TimePickerDialog.

When the user selects a date using DatePickerDialog, the app immediately shows a TimePickerDialog.

The selected date and time are combined into a string.

The TextView is updated with the selected date and time.

This code handles both date and time selection sequentially using two dialogs.

3) What are the methods for persisting data to files in Android?

ANDROID PERSISTENCE WITH PREFERENCES AND FILES

File based persistence:

Android allows to persists application data via the file system. For each application the Android system creates a *data/data/[application package]* directory.

Android supports the following ways of storing data in the local file system:

- Files - You can create and update files
- Preferences - Android allows you to save and retrieve persistent key value pairs of primitive data type.
- SQLite database - instances of SQLite databases are also stored on the local file system.

Files are saved in the *files* folder and application settings are saved as XML files in the *shared_prefs* folder.If your application creates an SQLite database this database is saved in the main application directory under the *databases* folder.

The following screenshot shows a file system which contains file, cache files and preferences.

Only the application can write into its application directory. It can create additional sub-directories in this application directory. For these sub directories, the application can grant read or write permissions for other applications.

INTERNAL VS. EXTERNAL STORAGE

Android has internal storage and external storage. External storage is not

private and may not always be available. If, for example, the Android device is
connected with a computer, the computer may mount the external system via USB and that makes this external storage not available for Android applications.

APPLICATION ON EXTERNAL STORAGE

As of Android 8 SDK level it is possible to define that the application can or should be placed on external storage. For this set the android:installLocation to
preferExternal or auto.

25In this case certain application components may be stored on an encrypted external mount point. Database and other private data will still be stored in the
internal storage system.

4) How do you create and use an SQLite database in an Android application?

REATING AND USING DATABASES USING SQLite

Introduction to SQLite-Definition

SQLite is an in-process library that implements a self-contained, serverless,

zero-configuration, transactional SQL database engine. SQLite is an embedded
SQL database engine. Unlike most other SQL databases, SQLite does not have
a separate server process. SQLite reads and writes directly to ordinary disk files.
The below are three components of SQLite that you need to understand because
they are central to working with SQLite in Android.

1. **SQLiteOpenHelper** — this is the most important class that you will work with in Android SQLite. You will use SQLiteOpenHelper to create and upgrade your SQLite Database. In other words, SQLiteOpenHelper removes the effort required to install and configure database in other systems.

2. **SQLiteDatabase** — this is the actual database where your data is stored. When you created your database with SQLiteOpenHelper class, it sets everything in motion to create your database but holds off until you are ready to use that database. And the way SQLiteOpenHelper knows that you are ready to use your database is when you access that database either with getReadableDatabase() or getWritableDatabase() for read and write operations respectively.

3. **Cursor** — The reason you store your data in a database is so you can

access them later. That access is called a query and a successful query will return a list of the items you queried for. If that list is so long, your Android device may choke if you want to access all of the items in the returned result. This is where the Cursor comes in, the list of the items that you query for are wrapped in a Cursor and the Cursor hands them over to you in batches of any number.

Sample code:

```
package com.example.notescrud
import android.content.ContentValues
import android.content.Context
import android.database.sqlite.SQLiteDatabase
import android.database.sqlite.SQLiteOpenHelper
class NotesDataBaseHelper(context: Context) :
SQLiteOpenHelper(context, DATABASE_NAME, null,
DATABASE_VERSION) {
companion object {
private const val DATABASE_NAME = "notesapp.db"
private const val DATABASE_VERSION = 1
private const val TABLE_NAME = "allnotes"
private const val COLUMN_ID = "id"
private const val COLUMN_TITLE = "title"
private const val COLUMN_DESCRIPTION = "description"
}
```

```
override fun onCreate(db: SQLiteDatabase?) {
val createTableQuery =
"CREATE TABLE $TABLE_NAME($COLUMN_ID INTEGER PRIMARY
KEY, $COLUMN_TITLE TEXT, $COLUMN_DESCRIPTION TEXT)"
db?.execSQL(createTableQuery)
}28
override fun onUpgrade(db: SQLiteDatabase?, oldVersion: Int, newVersion:
Int) {
val dropTableQuery = "DROP TABLE IF EXISTS $TABLE_NAME"
db?.execSQL(dropTableQuery)
onCreate(db)
}
fun insertNote(note: NotesData) {
val db = writableDatabase
val values = ContentValues().apply {
put(COLUMN_TITLE, note.title)
put(COLUMN_DESCRIPTION, note.description)
}
db.insert(TABLE_NAME, null, values)
db.close()
}
fun getAllNotes(): List<NotesData> {
val noteList = mutableListOf<NotesData>()
```

```
val db = readableDatabase
val query = "SELECT * FROM $TABLE_NAME"
val cursor = db.rawQuery(query, null)
try {
if (cursor.moveToFirst()) {
do {
val id = cursor.getInt(cursor.getColumnIndexOrThrow(COLUMN_ID))
val title =
cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_TITLE))
val description =
cursor.getString(cursor.getColumnIndexOrThrow(COLUMN_DESCRIPTIO
N))
val note = NotesData(id, title, description)
noteList.add(note)
} while (cursor.moveToNext())
}
} finally {
cursor.close()
db.close()
}29
return noteList
}
}
```

Companion object:

➢

This contains static values for the database, table name, and column names, which are constants used across the class.

➢

DATABASE_NAME: The name of the database file (e.g., notesapp.db).

➢

DATABASE_VERSION: The version number of the database (used for managing upgrades).

➢

TABLE_NAME: The name of the table where notes are stored.

•

COLUMN_ID,

COLUMN_TITLE,

COLUMN_DESCRIPTION:

Column names for storing the ID, title, and description of each note.

onCreate(db: SQLiteDatabase?):

➢This method is called when the database is created for the first time.

➢ The SQL command "CREATE TABLE $TABLE_NAME..." creates a table

named allnotes with three columns:

o id (an integer that serves as the primary key).

o title (a text field for the note's title).

o description (a text field for the note's description).

➢ **Primary Key**: The id is the primary key, ensuring each note has a unique identifier.

onUpgrade(db: SQLiteDatabase?, oldVersion: Int, newVersion: Int):30

➢

This method is called when the database needs to be upgraded, usually when there is a change in the database schema, like adding new columns or changing data types.

➢

It drops the existing table using DROP TABLE IF EXISTS and calls onCreate() again to recreate the table with the new schema.

insertNote():

➢ This method inserts a new note into the database.

➢ **Parameters**: Takes a NotesData object as input, which contains the note's title and description.

➢ **Steps**:

o Opens the database in write mode (writableDatabase).

o Uses a ContentValues object to map the title and description to the appropriate columns.

o Inserts the note into the allnotes table using db.insert().

o Closes the database connection after the insertion.

getAllNotes():

➢ This method retrieves all notes from the database.

➢ **Steps**:

o Opens the database in read mode (readableDatabase).

o Executes a SQL query "SELECT * FROM $TABLE_NAME" to retrieve all records from the allnotes table.

o Loops through the result set (cursor) to extract the id, title, and description of each note.

o Creates NotesData objects for each note and adds them to a noteList.

5) What is Geofencing? How it is used for Monitoring the Location?

ONITORING A LOCATION IN ANDROID

Monitoring a location in Android involves continuously tracking the user's location and taking actions when specific conditions are met, such as when the user enters or exits a geographical area. This can be achieved through **Geofencing** or continuous **Location Updates** using the **Fused Location Provider**.

Two Main Approaches for Monitoring Location:

1. **Geofencing**: Allows monitoring specific geographical regions (circular areas) and triggers events when the user enters or exits those regions.
2. **Continuous Location Updates**: Provides continuous updates of the user's location, allowing you to track movement in real-time.

What is Geofencing?Geofencing allows apps to define geographical areas, known as geofences, and
monitor when a device enters or exits these areas. It's ideal for use cases such as sending reminders when entering a store or triggering notifications when exiting a defined location.

Creating and Add a Geofence

1. **Create the Geofence**: A geofence is defined by a location (latitude and longitude) and a radius. It can also include expiration time and transition types (enter, exit).
2. **GeofencingRequest**: This defines how geofences should behave and handles the creation of multiple geofences.
3. **PendingIntent**: A PendingIntent is used to define what should happen when a geofence is triggered (such as starting a service or broadcasting a message).

```
class GeofenceActivity : AppCompatActivity() {
private lateinit var geofencingClient: GeofencingClient
private lateinit var geofencePendingIntent: PendingIntent
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_geofence)
geofencingClient = LocationServices.getGeofencingClient(this)
val geofence = Geofence.Builder()
.setRequestId("GEOFENCE_ID")
```

.setCircularRegion(37.7749, -122.4194, 100f) // Coordinates for San Francisco, radius 100 meters

.setExpirationDuration(Geofence.NEVER_EXPIRE)

.setTransitionTypes(Geofence.GEOFENCE_TRANSITION_ENTER or Geofence.GEOFENCE_TRANSITION_EXIT)

.build()

val geofencingRequest = GeofencingRequest.Builder()

.setInitialTrigger(GeofencingRequest.INITIAL_TRIGGER_ENTER)

.addGeofence(geofence)

.build()

geofencePendingIntent = PendingIntent.getBroadcast(

this, 0, Intent(this, GeofenceBroadcastReceiver::class.java),

PendingIntent.FLAG_UPDATE_CURRENT

)

Create a Broadcast Receiver for Geofence Transitions

Create a BroadcastReceiver to handle events when the user enters or exits the defined geofence.

class GeofenceBroadcastReceiver : BroadcastReceiver() {

override fun onReceive(context: Context, intent: Intent) {

val geofencingEvent = GeofencingEvent.fromIntent(intent)

if (geofencingEvent.hasError()) {

// Handle error

return

```
}
val geofenceTransition = geofencingEvent.geofenceTransition
if (geofenceTransition == Geofence.GEOFENCE_TRANSITION_ENTER ||
geofenceTransition == Geofence.GEOFENCE_TRANSITION_EXIT) {
}
}
}
```

5) Briefly Explain Consuming Web Services Using Http in Android.

CONSUMING WEB SERVICES USING HTTP IN ANDROID

In Android, consuming web services via HTTP is a common requirement, allowing apps to interact with remote servers, fetch data, or submit information. There are multiple ways to implement HTTP requests in Android,

but two popular approaches include:

➢

Retrofit: A type-safe HTTP client for Android, which simplifies the process of making network requests and handling responses.

➢

HttpURLConnection: A built-in class in Android for managing HTTP requests, though it is more verbose compared to libraries like Retrofit.

Retrofit for HTTP Requests

Retrofit is highly preferred for modern Android development due to its ease of
use and powerful features like automatic JSON parsing. Here's how you can use Retrofit to consume web services.First, add the required dependencies for Retrofit and a converter (like Gson for
JSON parsing) to your app’s build.gradle file.

```
dependencies {
implementation 'com.squareup.retrofit2:retrofit:2.9.0'
implementation 'com.squareup.retrofit2:converter-gson:2.9.0'
}
```

Define API Endpoints

You define API endpoints using an interface. For instance, to fetch a list of posts
from a sample API.

```
import retrofit2.Call
import retrofit2.http.GET
data class Post(val userId: Int, val id: Int, val title: String, val body: String)
interface ApiService {
@GET("posts")
fun getPosts(): Call<List<Post>>
}
```

Setup Retrofit Instance

Create a singleton Retrofit instance with a base URL and a converter factory

```
import retrofit2.Retrofit
import retrofit2.converter.gson.GsonConverterFactory
object RetrofitInstance {
private val retrofit by lazy {
Retrofit.Builder()
.baseUrl("https://jsonplaceholder.typicode.com/")
.addConverterFactory(GsonConverterFactory.create())
.build()
}
val api: ApiService by lazy {
retrofit.create(ApiService::class.java)
}
}
```

Make an HTTP Request

Now, make the HTTP request in your Activity or Fragment

```
private fun fetchPosts() {
val call = RetrofitInstance.api.getPosts()
call.enqueue(object : Callback<List<Post>> {
override fun onResponse(call: Call<List<Post>>, response:
Response<List<Post>>) {
if (response.isSuccessful) {
response.body()?.let { posts ->
for (post in posts) {
```

```
println("Post: ${post.title}")
}
}
}
}
override fun onFailure(call: Call<List<Post>>, t: Throwable) {
println("Error: ${t.message}")
}
})
}
```

HttpURLConnection for HTTP Requests

If you prefer not to use external libraries, Android's HttpURLConnection is a lower-level solution to make HTTP requests. This is more manual but built into
the framework.

```
fun fetchUsingHttpURLConnection() {
val url = URL("https://jsonplaceholder.typicode.com/posts")
val connection = url.openConnection() as HttpURLConnection
try {
connection.requestMethod = "GET"
connection.connect()
val responseCode = connection.responseCode
if (responseCode == HttpURLConnection.HTTP_OK) {
```

val inputStream = connection.inputStream

val content = inputStream.bufferedReader().use { it.readText() }

println("Response: $content")

} else {

println("Failed to fetch data. Response code: $responseCode")

}

} catch (e: Exception) {

e.printStackTrace()

} finally {

connection.disconnect()

}

*}***Handling Permissions**

If you're making network requests in an Android app, ensure you add the required permission in your AndroidManifest.xml file.

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

For apps targeting Android 9 (API level 28) or higher, make sure to configure **cleartext traffic** if you're using HTTP instead of HTTPS

<application

android:usesCleartextTraffic="true"

...>

</application>

7) How do you handle SMS Messages in Android?

MS MESSAGING

SMS messaging is one of the main killer applications on a mobile phone today

— for some users as necessary as the phone itself. Any mobile phone you buy

today should have at least SMS messaging capabilities, and nearly all users of any age know how to send and receive such messages.

1. Setting up Permissions

For any SMS functionality, your app needs to request the appropriate permissions. In Android 13, you must request **SEND_SMS**, **RECEIVE_SMS**,

and optionally **READ_SMS** permissions.

Add the following permissions to your AndroidManifest.xml:

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

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

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

Since Android 6.0 (API level 23), apps must also request these permissions at runtime. Here's an example of how to handle runtime permission requests for SMS in Kotlin:

if (ContextCompat.checkSelfPermission(this, Manifest.permission.SEND_SMS) != PackageManager.PERMISSION_GRANTED) {

ActivityCompat.requestPermissions(this,

arrayOf(Manifest.permission.SEND_SMS), REQUEST_SMS_PERMISSION)

*}***2. Sending SMS using SmsManager**

You can send SMS messages using the SmsManager class. It provides simple methods to send text messages.

val phoneNumber = "1234567890"

val message = "Hello, this is a test message!"

val smsManager = SmsManager.getDefault()

smsManager.sendTextMessage(phoneNumber, null, message, null, null)

In this example:

•

phoneNumber: The recipient's phone number.

•

message: The text you want to send.

•

The other parameters (null values) are for service center address, delivery intent, and sent intent, which are optional.

3. Receiving SMS Messages

To receive an SMS, you'll need to create a BroadcastReceiver that listens for SMS broadcasts from the system. Android automatically sends a broadcast when a new SMS is received.

Step-by-Step Guide:

•

Create a BroadcastReceiver: This class will capture the broadcast that the system sends when an SMS is received.

•

Declare the receiver in the manifest: The receiver must be declared in the manifest with the action android.provider.Telephony.SMS_RECEIVED.

Example: SMS Receiver

SmsReceiver.kt*class SmsReceiver : BroadcastReceiver() {*

override fun onReceive(context: Context?, intent: Intent?) {

if (intent != null && Telephony.Sms.Intents.SMS_RECEIVED_ACTION == intent.action) {

val bundle = intent.extras

if (bundle != null) {

val pdus = bundle.get("pdus") as Array<>*

for (pdu in pdus) {

val format = bundle.getString("format")

val smsMessage = SmsMessage.createFromPdu(pdu as ByteArray, format)

// Extract sender and message content

val sender = smsMessage.originatingAddress

val messageBody = smsMessage.messageBody

// Example: Display the message in a Toast

Toast.makeText(context, "Message from $sender: $messageBody",

Toast.LENGTH_LONG).show()

}

}

}

}**AndroidManifest.xml** Add the receiver and permission declaration in your manifest:

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

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

<application ... >

<receiver android:name=".SmsReceiver" android:enabled="true" android:exported="true">

<intent-filter android:priority="999">

<action android:name="android.provider.Telephony.SMS_RECEIVED" />

</intent-filter>

</receiver>

</application>

Here:

- Telephony.Sms.Intents.SMS_RECEIVED_ACTION is the action indicating that an SMS has been received.
- The **pdus** array contains Protocol Data Units (PDUs), which represent

the received SMS message.

4. Requesting Runtime Permissions

Because of Android's security model, you need to ask for SMS permissions at runtime (Android 6.0 and above).

MainActivity.kt

```
class MainActivity : AppCompatActivity() {
private val REQUEST_SMS_PERMISSION = 101
override fun onCreate(savedInstanceState: Bundle?) {
super.onCreate(savedInstanceState)
setContentView(R.layout.activity_main)
// Request permission for sending SMS
if (ContextCompat.checkSelfPermission(this,
Manifest.permission.RECEIVE_SMS) !=
PackageManager.PERMISSION_GRANTED) {
ActivityCompat.requestPermissions(this,
arrayOf(Manifest.permission.RECEIVE_SMS),
REQUEST_SMS_PERMISSION)
}
// Button to send an SMS
val sendButton: Button = findViewById(R.id.send_sms_button)
sendButton.setOnClickListener {
sendSms("1234567890", "Hello from Android 13!")
}
```

```
}
private fun sendSms(phoneNumber: String, message: String) {
try {
val smsManager: SmsManager = SmsManager.getDefault()
smsManager.sendTextMessage(phoneNumber, null, message, null, null)
Toast.makeText(this, "SMS sent successfully",
Toast.LENGTH_SHORT).show()
} catch (e: Exception) {
Toast.makeText(this, "SMS sending failed: ${e.message}",
Toast.LENGTH_LONG).show()
}
}
override fun onRequestPermissionsResult(requestCode: Int, permissions:
Array<out String>, grantResults: IntArray) {
super.onRequestPermissionsResult(requestCode, permissions, grantResults)
if (requestCode == REQUEST_SMS_PERMISSION &&
grantResults.isNotEmpty() && grantResults[0] ==
PackageManager.PERMISSION_GRANTED) {
Toast.makeText(this, "SMS Permission Granted",
Toast.LENGTH_SHORT).show()
} else {
Toast.makeText(this, "SMS Permission Denied",
Toast.LENGTH_SHORT).show()
```

}

}

}

Key Points:

- **SmsManager**: Use it for sending SMS messages programmatically.
- **BroadcastReceiver**: Use it to listen for incoming SMS messages.
- **Permissions**: Always request runtime permissions and handle permission denial gracefully.

8) Write the steps for Creating and Implementing Maps in Android.

DISPLAYING MAPS IN ANDROID

Displaying maps in Android is primarily achieved using the **Google Maps API**, which allows developers to embed Google Maps into their applications. This service provides interactive and customizable map views that support functionalities such as zoom, markers, user location, and more.

Android offers two main libraries for displaying maps:

- **Google Maps SDK for Android**: Allows embedding maps powered by

Google Maps into Android apps.

➢

OSMDroid: An open-source alternative to Google Maps, useful for offline mapping or when developers prefer not to use Google's services.

Google Maps API Overview

The **Google Maps SDK for Android** provides a rich set of APIs to add maps and various features, such as markers, polylines, and polygons. It supports interactive features like zooming, scrolling, and adding overlays for better visual representation.**Steps to Implement Google Maps in Android**

1. **Set Up Google Maps API**: To start with Google Maps, you need to register your app in the Google Cloud Console and obtain an API key.

➢Enable the **Google Maps Android API** service.

➢Generate the **API key** that will be used in your Android project.

2. **Modify the Manifest**: Include the required permissions and API key in the AndroidManifest.xml file. The permissions ensure the app can access the device’s location and internet to display the maps.

<meta-data android:name="com.google.android.geo.API_KEY" android:value="YOUR_API_KEY_HERE" />

3. **Add Google Play Services Dependency**: Add the required dependencies for Google Maps in your build.gradle file.

implementation 'com.google.android.gms:play-services-maps:18.1.0'

4. **Create a Map Fragment**: A MapFragment or SupportMapFragment is used to embed the map in the layout. You define this fragment in your

activity’s layout file.

<fragment

android:id="@+id/map"

android:name="com.google.android.gms.maps.SupportMapFragment"

android:layout_width="match_parent"

android:layout_height="match_parent" />

5. **Initialize the Map in Activity**: In your activity or fragment, you need to initialize the map and manage its lifecycle. You can set up various features such as adding markers, zoom controls, and gestures.

import android.os.Bundle

import androidx.appcompat.app.AppCompatActivity

import com.google.android.gms.maps.CameraUpdateFactoryclass MapsActivity : AppCompatActivity(), OnMapReadyCallback {

private lateinit var map: GoogleMap

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity_maps)

val mapFragment = supportFragmentManager.findFragmentById(R.id.map) as

SupportMapFragment

mapFragment.getMapAsync(this)

}

override fun onMapReady(googleMap: GoogleMap) {

map = googleMap

val location = LatLng(37.7749, -122.4194) // Coordinates for San Francisco

map.addMarker(MarkerOptions().position(location).title("Marker in San Francisco"))

map.moveCamera(CameraUpdateFactory.newLatLngZoom(location, 12.0f))

}

*}***Important Components**

1. **SupportMapFragment**: This is a specialized fragment to display a map using Google Maps.
2. **GoogleMap**: The object that controls the map and allows customization (e.g., setting markers, adding overlays).
3. **MarkerOptions**: Used to add markers on the map at specific latitude/longitude coordinates.
4. **CameraUpdateFactory**: Manages the camera (view) on the map to zoom in or out and centre on certain locations.

9) Explain the lifecycle of an Android Service and how it differs from an activity lifecycle.

SERVICES

In Android development, **services** are components that perform background

operations, often running independently from an activity. They are typically used for long-running tasks, such as playing music, downloading files, or performing network requests. Services do not provide a user interface. A service can run independently meaning that after your app starts the service, it

can run even when your app is no longer in focus.

A Service provides great flexibility by having three different types. They are Foreground, Background, and Bound.

11) **Foreground services**: These services perform tasks that are noticeable to the user, like playing music or handling ongoing notifications.

2) **Background services**: These run without user interaction and without a visible user interface, like syncing data or running in the background for maintenance tasks.

3) **Bound services**: These allow activities (or other components) to bind to the service and interact with it, often for tasks like getting real-time updates or sending commands.

When to Use Service?

We will likely use a service when you want to do something that does not involve UI and needs it to run whether your app is running or not. For example,

if you want to play audio a service would be a choice because the audio will still play when the user opens another app.

COMMUNICATION BETWEEN A SERVICE AND AN ACTIVITY

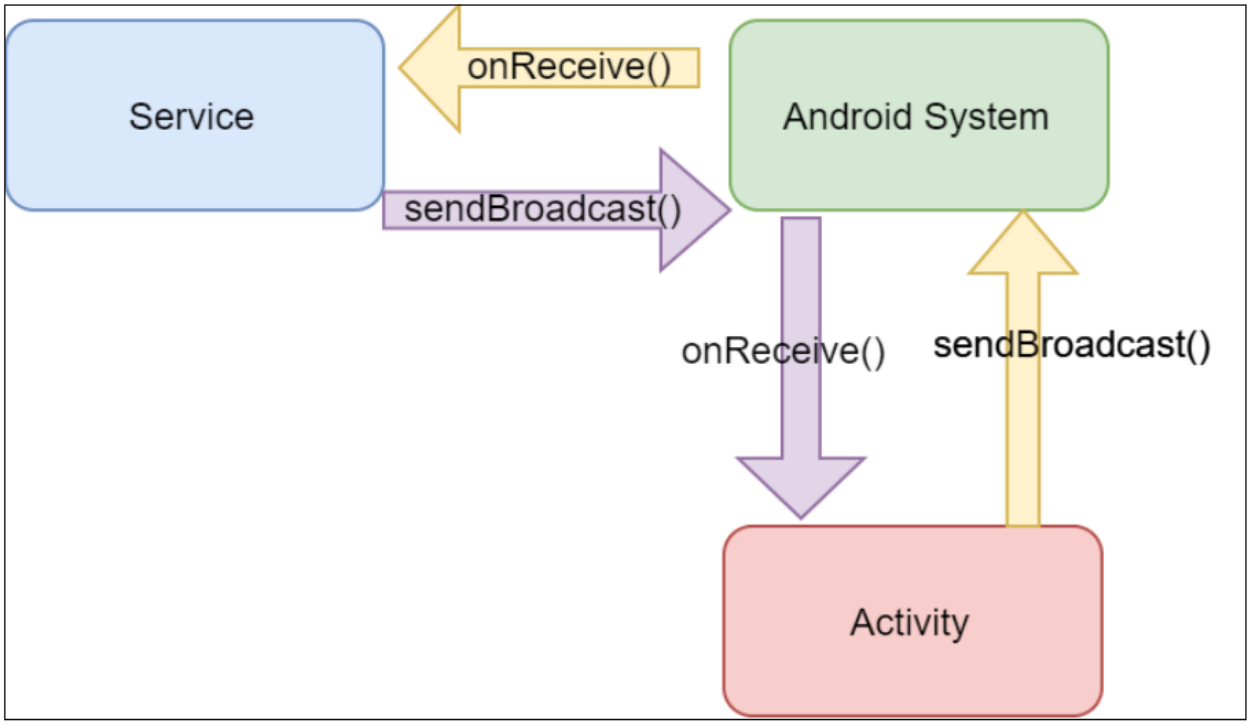
For an activity to communicate with a service, Android provides different mechanisms:

1) **Intents**: You can use explicit intents to start a service. Intents carry data to be processed by the service.

2) **Broadcast Receivers**: Services can send broadcasts that activities or other components listen to and respond to.

3) **Messenger**: A Messenger allows communication using Handler and Message objects. This is useful for passing simple messages back and forth between an activity and a service.

24) **AIDL (Android Interface Definition Language)**: This is used for more complex services, especially when multiple applications need to communicate with a service.



There are two simple ways for you to use an Activity and Service together. One
way is to create a bound service and bind a component in your activity to the service acting as a communication channel. Another way is to use the Android
Broadcast System, which sends out a broadcast intent system-wide.

10) How can an activity bind to a service in Android? Explain the binding process and its advantages.

BINDING ACTIVITIES TO SERVICES

A **bound service** allows activities to directly interact with the service. This binding is important when the activity requires frequent communication with the service, such as for fetching updates in real time.

Real-time Example: Music Player App

Imagine a music player application where a service is responsible for playing music in the background. The user interacts with the app's UI (the activity), but

the music continues playing even when the user navigates away from the activity. In this case, the activity can bind to the service to control the music (play, pause, stop) or retrieve the current playback status.

Step 1: Create the Service-> Define a bound service that plays music

class MusicService : Service() {

private val binder = LocalBinder()

inner class LocalBinder : Binder() {

fun getService(): MusicService = this@MusicService

}

override fun onBind(intent: Intent?): IBinder {

return binder

}

fun playMusic() {

// Logic to play music

}

fun pauseMusic() {

// Logic to pause music

}

fun stopMusic() {

// Logic to stop music

}

*}*5

Step 2: Bind the Activity to the Service->The activity can bind and communicate with the service.

class MusicPlayerActivity : AppCompatActivity() {

private var musicService: MusicService? = null

private var isBound = false

```
private val connection = object : ServiceConnection {
override fun onServiceConnected(name: ComponentName?, service: IBinder?) {
val binder = service as MusicService.LocalBinder
musicService = binder.getService()
isBound = true
}
override fun onServiceDisconnected(name: ComponentName?) {
isBound = false
}
}
override fun onStart() {
super.onStart()
Intent(this, MusicService::class.java).also { intent ->
bindService(intent, connection, Context.BIND_AUTO_CREATE)
}
}
override fun onStop() {
super.onStop()
if (isBound) {
unbindService(connection)
isBound = false6
}
```

}

fun onPlayButtonClicked() {

musicService?.playMusic()

}

fun onPauseButtonClicked() {

musicService?.pauseMusic()

}

fun onStopButtonClicked() {

musicService?.stopMusic()

}

}

➢ The MusicService is a **bound service** that handles music playback.

➢ The MusicPlayerActivity binds to the service during its lifecycle (onStart and onStop).

➢ When the user presses play, pause, or stop buttons, the activity sends commands to the bound service to control the music.

11) Describe the process of deploying APK files for testing and production in Android development.

DEPLOYING APK FILES

Release your app to users

You can release your Android apps several ways. Typically, you release apps

through an app marketplace such as Google Play. You can also release apps on
your own website or by sending an app directly to a user.

Release through an app marketplace

If you want to distribute your apps to the broadest possible audience, release them through an app marketplace.

Google Play is the premier marketplace for Android apps and is particularly useful if you want to distribute your apps to a large global audience. However, you can distribute your apps through any app marketplace, and you can use multiple marketplaces.

Release your apps on Google Play

Google Play is a robust publishing platform that helps you publicize, sell, and distribute your Android apps to users around the world. When you release your apps through Google Play, you have access to a suite of developer tools that let you analyze your sales, identify market trends, and control who your apps are being distributed to.

Google Play also gives you access to several revenue-enhancing features such
as in-app billing and app licensing. The rich array of tools and features, coupled
with numerous end-user community features, makes Google Play the premier marketplace for selling and buying Android apps.

Releasing your app on Google Play is a simple process that involves three basic
steps:

➢

Prepare promotional materials

To fully leverage the marketing and publicity capabilities of Google Play, you need to create promotional materials for your app such as screenshots, videos, graphics, and promotional text.

➢

Configure options and uploading assets

Google Play lets you target your app to a worldwide pool of users and devices. By configuring various Google Play settings, you can choose the countries you want to reach, the listing languages you want to use, and the price
you want to charge in each country.

You can also configure listing details such as the app type, category, and content rating. When you are done configuring options, you can upload your promotional materials and your app as a draft app.

➢

Publish the release version of your app

If you are satisfied that your publishing settings are correctly configured and your uploaded app is ready to be released to the public, click **Publish**. Once

it has passed Google Play review, your app will be live and available for download around the world.

12) Write a brief overview of the Derby app in iOS and the steps to build it.

UILDING THE DERBY APP IN IOS

1. Setting Up the Project

➢

Open **Xcode** and create a new project.

➢

Select **App** under the iOS tab and click **Next**.

➢

Name the project "DerbyApp" and choose **Swift** as the language.

➢

Set the interface to **Storyboard** and leave the other settings as they are.

➢

Click **Next**, choose a location for your project, and click **Create**.

2. Designing the User Interface (UI)

In **Main.storyboard**, you will create the interface for the Derby game.

a. Add UI Elements

➢

Open Main.storyboard and select the **ViewController**.

➢

Drag **Image Views** from the Object Library (on the right) onto the storyboard. Each Image View will represent a horse. Add three Image Views for three horses.

➢

Set images for the horses (you can use placeholder images for now).

➢

Add a **UIButton** labeled "Start Race" that will start the race when tapped.

➢

Add a **UILabel** at the top to display which horse wins the race.20

b. Arrange the UI

➢

Position the three horses near the left edge of the screen.

➢

Align them vertically with some space between them.

➢

Place the "Start Race" button below the horses.

➢

Position the result label at the top.

c. Connect UI Elements to Code

➢

Open the **Assistant Editor** (click the two overlapping circles at the top right) so that you can see both the ViewController.swift file and the

storyboard side by side.

➢

Ctrl-drag from each of the three Image Views to the ViewController.swift file to create outlets. Name them horse1, horse2, and horse3.

➢

Ctrl-drag from the "Start Race" button to create an action method. Name it startRaceTapped.

➢

Ctrl-drag from the label to create an outlet called resultLabel.

3. Writing the Game Logic in Swift

Now, let’s implement the logic for the Derby race. The horses will move across the screen when the race starts, and the game will declare a winner when the first horse reaches the finish line.

Open ViewController.swift and update it as follows:

```
import UIKit
class ViewController: UIViewController {
@IBOutlet weak var horse1: UIImageView!
@IBOutlet weak var horse2: UIImageView!
@IBOutlet weak var horse3: UIImageView!
```



```
// Outlet for the result label
@IBOutlet weak var resultLabel: UILabel!
```

```
// A timer to animate the horses
var raceTimer: Timer?
// Track if the race is running
var raceInProgress = false
override func viewDidLoad() {
super.viewDidLoad()
// Initialize the result label to be empty at start
resultLabel.text = ""
}
// Start race button action
@IBAction func startRaceTapped(_ sender: UIButton) {
if raceInProgress {
return // Ignore if a race is already running
}
raceInProgress = true
resultLabel.text = "" // Clear the result label before the race
// Reset horse positions
resetHorsePositions()22
// Start the race timer to move horses
raceTimer = Timer.scheduledTimer(timeInterval: 0.05, target: self, selector:
#selector(moveHorses), userInfo: nil, repeats: true)
}
// Reset the horses to the starting line
```

```
func resetHorsePositions() {
horse1.frame.origin.x = 20
horse2.frame.origin.x = 20
horse3.frame.origin.x = 20
}
// Move horses forward
@objc func moveHorses() {
let finishLine = view.frame.width - 100
// Move each horse by a random amount
horse1.frame.origin.x += CGFloat.random(in: 2...10)
horse2.frame.origin.x += CGFloat.random(in: 2...10)
horse3.frame.origin.x += CGFloat.random(in: 2...10)
// Check if any horse has crossed the finish line
if horse1.frame.origin.x >= finishLine {
declareWinner(horse: "Horse 1")
} else if horse2.frame.origin.x >= finishLine {23
declareWinner(horse: "Horse 2")
} else if horse3.frame.origin.x >= finishLine {
declareWinner(horse: "Horse 3")
}
}
// Stop the race and declare a winner
func declareWinner(horse: String) {
```

raceTimer?.invalidate() // Stop the race timer

raceInProgress = false

// Update the result label with the winner

resultLabel.text = "\(horse) Wins!"

}

}

Explanation of Code:

➢

Outlets and Actions: The outlets connect the UI elements (horses and result label) to the code. The startRaceTapped function is triggered when the user taps the "Start Race" button.

➢

Race Timer: The timer (raceTimer) is used to repeatedly move the horses across the screen. It updates every 0.05 seconds and moves each horse by a random number of pixels.

➢

Finish Line: We define the finish line as view.frame.width - 100 (100 pixels from the right edge of the screen).

➢

Random Movement: Each horse moves forward by a random amount between 2 and 10 pixels, simulating the race.24

➢

Winner Declaration: When a horse crosses the finish line, the race

stops, and the result label displays the winner.

4. Running the App

➢

Press Cmd + R or click the **Run** button in Xcode to build and run the app in the iOS Simulator.

➢

Tap the "Start Race" button to begin the race, and watch as the horses move across the screen.

➢

The result label will display the winner once one of the horses crosses the finish line.