Android Developer Interview Questions

deepwaterooo

April 7, 2020

Contents

1 Important and frequently asked Android developer interview questions

- https://www.educba.com/android-developer-interview-questions/
- https://www.educba.com/android-interview-questions/

1.1 Android

1.1.1 Explain Platform Architecture of android? And what is Google Android SDK?

- Android is an open source, LINUX based software pack. It mainly comprises of
  - Linux Kernel
  - Hardware Abstraction Layer (HAL)
  - Native libraries, Android Runtime
  - Java API
  - System Apps

- Talking from bottom to top
- **Linux Kernel**:

  - The base layer of an application that directly interfaces with the device hardware.
  - the Linux kernel provides drivers for connecting to hardware.
  - The driver's list contains – Audio, Binder, Display, Keypad, Bluetooth, Camera, USB, Wi-Fi and power management.

- **Hardware Abstraction Layer (HAL)**:
  - The Hardware Abstraction Layer (HAL) provides a standard that interacts with Kernel drivers to access these hardware features to the users.

- **Android Runtime**:
  - ART (Android Runtime) is there to help when multiple virtual machines (VM's) are run on low memory devices with the execution of DEX files.
  - The Dalvik Virtual Machine is in the same level as the Native Libraries and allows every Android app to run its own processes.

- **Native libraries**:
  - Native C or C++ is needed to build the native code.
  - *a set of open-source libraries, including the SQLite database, libc, and the WebKit browser engine. ???*

- **Java API**:
  - Android provides Java API's to use functionalities of native libraries to the apps.
  - Java API forms the building blocks on a need to create Android apps by providing the components and services.
  - This level provides higher-level services in the form of Java classes to applications on the device. The key services to know are the Activity Manager, Content Providers, Resource Manager, Notifications Manager, and the View System.

- **System Apps**:
  - Android comes with a set of core apps for emails, SMS messaging, calendars, internet browsing, contacts and more.

- **Google SDK** is *a development toolkit* that is used by developers to write programs for mobile devices. It provides a graphical user interface simulating Android driven environment and enabling the developer to test and debug their codes.

1.1.2 Can you describe the core building blocks of an Android application?

- This top-level question is a great way to warm up the developer and get a feel for how well they understand the basics of building an Android app from scratch. The basic components are as follows:
  - **Activity**: An activity is a subclass of the **ContextThemeWrapper** class. Since almost all activities interact directly with the user, it is often helpful to think of an activity as the screen for a particular action, such as logging in or taking a picture.
  - **View**: The view is everything you can see on the screen of the app—think of the individual UI elements like buttons, labels, and text fields.
  - **Intent**: The main purpose of intent is to invoke individual components. Common uses include starting the service, launching activities, displaying a list of contacts, dialing a phone number, or displaying a web page.

- **Service**: A service is a background process that can either be local or remote.
  * **Local services** may be accessed from within the application
  * while **remote services** are intended to be used by other applications running on the same device.
- **Content Provider**: Content providers share data between applications.
- **Fragment**: Fragments are best thought of as parts of an activity—you can display more than one fragment on the screen at the same time.
- **Android Manifest.xml**: The AndroidManifest.xml file provides essential information about your app required for it to run on the Android operating system. All Android apps have this file in their root directory.

1.1.3 Briefly explain the components/requirements for any Android development projects?

- This is the most popular Android developer Interview Questions asked in an interview. The below-listed components are needful for successful completion:
  - **Build**: contains the build output.
  - **Src**: holds the code and resource file.
  - **Res**: holds the bitmap images, UI, XML layouts.
  - **Assets**: holds the file which can be combined into a .apk file.
  - **Manifest**: holds the XML file.

1.1.4 Explain in brief the files and folder which are created when an Android project is created.

- The new project should have following files in the package in eclipse.
  - **src** This file contains java source files for the newly created project. The code for the application is to be written in this file. It should be made available under the name of a project.
  - **Assets** This is a folder which will contain all information regarding HTML files, text files, and databases.
  - **gen** This folder must have the R.java file. It is a file generated by the compiler and it references the resources that are found in the project. This file should not be modified as it is generated by the compiler.
  - **Android library** This folder contains an android.jar file which has all libraries needed for creating an Android application.
  - **bin** It contains the .apk file that is created by ADT during the code build process. This file is the application binary file. When a developer runs his code this file has everything required to run a code.
  - **res** This is a folder that contains all resource files used by the application. It has sub-folders like drawable, menu, layout, and values etc.

1.1.5 What are the important items in Android and explain the importance of XML based layouts?

- The must required items in an Android project when a project is created are as below:
  - Androidmanifest.xml
  - Build.xml

  - bin
  - src
  - res
  - assets

- Here we have two XML files which help in providing a consistent layout. It helps in giving the developer a standard graphical definition format. Usually, all layout details are placed in these XML files and the other items are placed in source files.

1.1.6 What is DDMS? Describe some of its capabilities.

- DDMS is short for Dalvik Debug Monitor Server. It ships natively with Android and contains a number of useful debugging features including:
  - port-forwarding services 端口转发服务
  - screen capture 屏幕截图
  - thread and heap information 线程和堆信息
  - network traffic tracking 网络跟踪
  - incoming call/SMS spoofing 来电和短信监听
  - simulating network state, speed, and latency 模拟网络状态，速度和延迟
  - location data spoofing 位置数据监听

port-forwarding services	端口转发服务
screen capture	屏幕截图
thread and heap information	线程和堆信息
network traffic tracking	网络跟踪
incoming call/SMS spoofing	来电和短信监听
simulating network state, speed, and latency	模拟网络状态，速度和延迟
location data spoofing	位置数据监听

1.1.7 Why did you become an Android developer? What are some of the advantages of the Android platform?

- You want a developer who really knows how to play to the strengths of your chosen platform. Some key advantages of Android are listed below for your convenience.
  - **Open Source**: No licenses, no distribution or development fees.
  - **DVM (Dalvik Virtual Machine)**: DVM is a highly optimized virtual machine for mobile devices.
  - **Platform Diversity**: Since Android is open-source, it has been adopted by a wide range of manufacturers of mobile devices.
  - **Experience with Java**: Java is the language of choice for Android app development. Those who already have years of experience in Java will feel right at home developing for Android.

1.1.8 What is difference between Serializable and Parcelable? Which is best approach in Android?

- **Serializable** is a standard Java interface that's easy to integrate into your app, as it doesn't require any methods. You simply mark a class Serializable by implementing the interface, and Java will automatically serialize it in certain situations. Despite being easy to implement, Serializable uses the Java reflection API, which makes it a slow process that creates lots of temporary objects.

- **Parcelable** is an Android specific interface where you implement the serialization yourself. It was created to be far more efficient than Serializable, and to get around some problems with the default Java serialization scheme. **Parcelable** is optimized for Android, so it's faster than Serializable. It's also fully customizable, so you can be explicit about the serialization process, which results in less garbage objects.
- While the developer may acknowledge that implementing Parcelable does require more work, the performance benefits mean that they should advise using Parcelable over Serialization, wherever possible.

1.1.9 Coin as many Android OS version that you remember?

VERSION	NAME
Android 8.0	Oreo
Android 7.0 – 7.1.2	Nougat
Android 6 – 6.0.1	Marshmallow
Android 5 – 5.1.1	Lollipop
Android 4.4 – 4.4.4	KitKat
Android 4.1 – 4.3	Jelly Bean
Android 4.0-4.0.4	Ice Cream Sandwich

1.2 Activity, Fragment, AsynTask, Lifecyles

1.2.1 Explain the process to launch an activity on an application.

- To launch an activity developer needs to explicitly define intent. It specifies the activity that we wish to start. The following code will help you understand that activity which is sent in the second parameter in the new activity class. The first parameter is the Intent constructor in the current activity context.

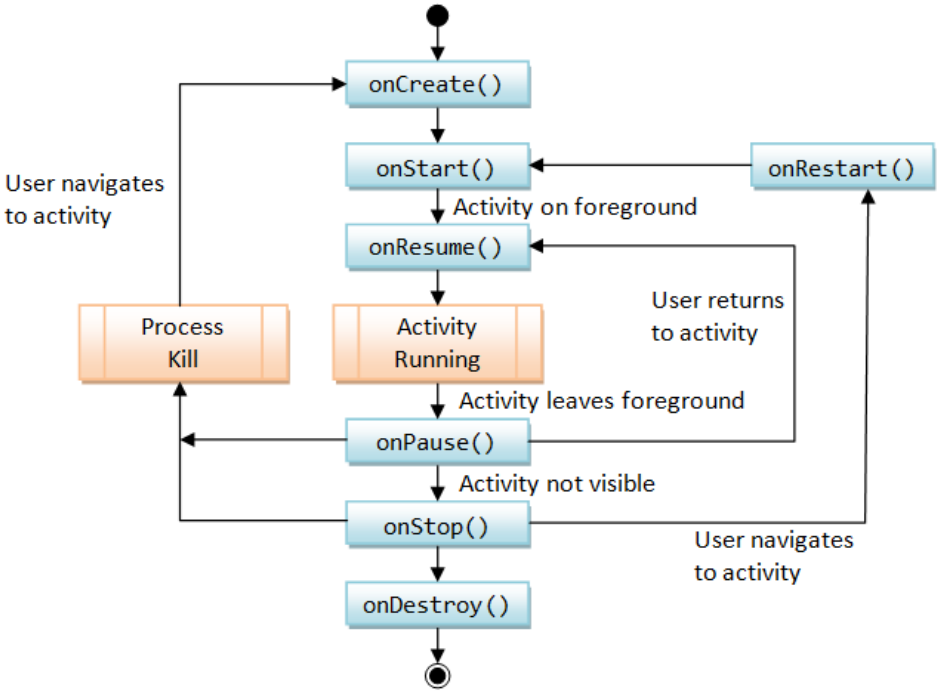
```
Intent intent1= new Intent(this, SecondActivity.class);
startActivity(intent1);
```

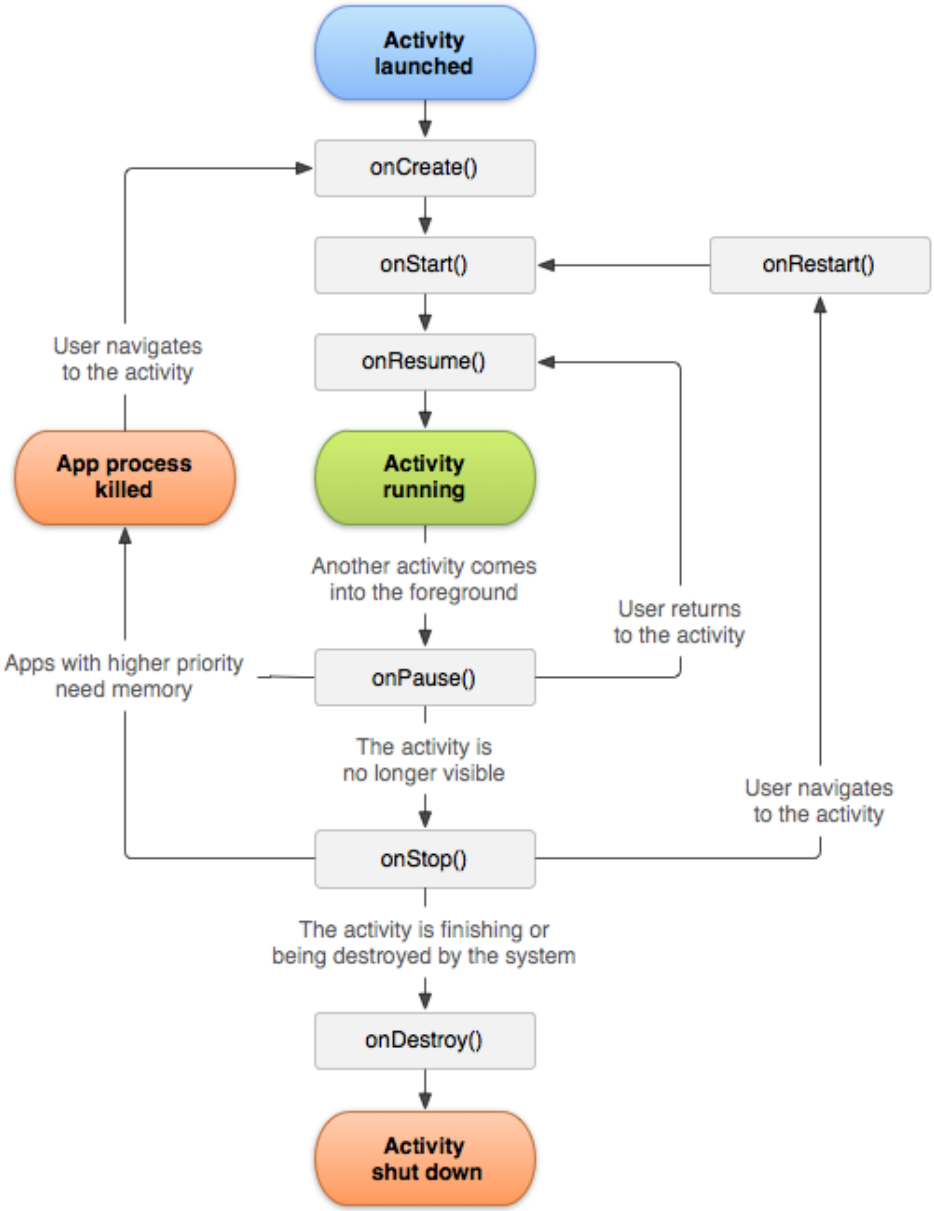
- If the user wishes to start activity from a particular fragment then below can be tried:

```
Intent intent1= new Intent(getActivity(), SecondActivity.class);
startActivity(intent1);
```

1.2.2 Can you explain the Android activity lifecycle?

- After a user navigates within the app, then the activity instances transit through different stages in their lifecycle.
- These activity classes provide a number of actions called as "callbacks" that gives information of the changed states the system creates, resumes or stops while resuming the activity.
- The activity life cycle has 4 states –
  - **Active or running** – If the activity is in the foreground of the screen it is called as active.
  - **Paused** – If the activity has lost focus but is still visible (like in the case of dialog comes top), then it is reoffered as paused.
  - **Stopped** – If an activity is completely obscured by another activity, it's called as stopped. It still retains all states and the information of member components.
  - **Finish** – If an activity is paused or stopped, the system can drop the activity from memory by either asking it to finish or simply killing the process.


onCreate()
onStart()
onRestart()
User navigates to activity
Activity on foreground
onResume()
User returns to activity
Process Kill
Activity Running
Activity leaves foreground
onPause()
Activity not visible
onStop()
User navigates to activity
onDestroy()



1.2.3 What are the states of an activity?

- There are four states of an activity. They are:
  - **Active**: When the activity is **active in the foreground**.
  - **Paused**: When activity is **in the background and still visible**.
  - **Stopped**: When activity is **not visible**.
  - **Destroyed**: When activity is **killed or terminated**.

1.2.4 What are the seven lifecycle methods of Android activity and what is their purpose?

- The seven lifecycle methods of Android activity are

  - onCreate()
  - onStart()
  - onRestart()
  - onResume()
  - onPause()
  - onStop()
  - onDestroy()

- Their purpose is to help structure your code around how you want an activity to perform throughout its lifecycle on the device.
- For example, **onCreate()** is where you would perform your static setup, from creating views to binding data to lists. It is always immediately followed by **onStart()**, where the app will be made visible to the user.

1.2.5 Write a quick script for launching a new activity within your application.

- The goal of this question is to quickly test their knowledge of explicit intent in launching an activity.
- An explicit intent explicitly defines the activity the developer wishes to start. A possible solution has been produced below.

```
1 Intent myIntent = new Intent(this, MyNewActivity.class);
2 startActivity(myIntent);
```

1.2.6 onDestroy()

- **Question**: The last callback in the lifecycle of an activity is onDestroy(). The system calls this method on your activity as the final signal that your activity instance is being completely removed from the system memory. Usually, the system will call onPause() and onStop() before calling onDestroy(). Describe a scenario, though, where onPause() and onStop() would not be invoked.
- onPause() and onStop() will not be invoked if **finish()** is called from within the **onCreate()** method. This might occur, for example, if you detect an error during onCreate() and call finish() as a result. In such a case, though, any cleanup you expected to be done in onPause() and onStop() will not be executed.
- Although onDestroy() is the last callback in the lifecycle of an activity, it is worth mentioning that this callback may not always be called and should not be relied upon to destroy resources. It is better have the resources created in **onStart()** and **onResume()**, and have them destroyed in **onStop()** and **onPause()**, respectively.

1.2.7 What's the difference between onCreate() and onStart()?

- The **onCreate()** method **is called once during the Activity lifecycle**, either when the application starts, or when the Activity has been destroyed and then recreated, for example during a configuration change.
- The **onStart()** method is called whenever the Activity becomes visible to the user, typically after **onCreate()** or **onRestart()**.

1.2.8 Is it possible to create an activity in Android without a user interface?

- Yes, an activity can be created without any user interface. These activities are treated as **abstract activities**.

1.2.9 When should you use a Fragment, rather than an Activity?

- This is still a much-debated topic, but the code used to create an Activity is fundamentally more involved than the code used to create a Fragment.
- The old Activity has to be destroyed, paused or stopped, and a new Activity has to be created.
- The best practice is to only use Activities when you need to swap the entire screen, and use fragments everywhere else.
- Any of the following use cases, where you'll almost always use a Fragment, rather than an Activity:
  - When you're working with UI components or behavior that you're going to use across multiple Activities.
  - When you're using one of the navigational methods that are closely linked to fragments, such as swipe views.
  - When your users would benefit from seeing two different layouts side-by-side.
  - When you have data that needs to persist across Activity restarts (i.e you need to use retained fragments).

1.2.10 You're replacing one Fragment with another —how do you ensure that the user can return to the previous Fragment, by pressing the Back button?

- This question provides an insight into the app developer's understanding of **the lifecycle of dynamic fragments, as well as Fragment transactions, and the back stack**.
- If the "Back" button is going to return the user to the previous Fragment, then you'll need to save each Fragment transaction to the back stack, by calling **addToBackStack()** before you **commit()** that transaction.
- The developer definitely shouldn't suggest creating a "Back" button specifically to handle navigating between fragments, but bonus points if they mention that you should never try to commit a FragmentTransaction after calling **onSaveInstanceState()**, as this can result in an exception.

1.2.11 How do you supply construction arguments into a Fragment?

- Construction arguments for a Fragment are passed via Bundle using the **Fragment.setArguments(B** method. The passed-in Bundle can then be retrieved through the **Fragment.getArguments()** method in the appropriate Fragment lifecycle method.
- It is a common mistake to pass in data through a custom constructor. Non-default constructors on a Fragment are not advisable because the Fragment may be destroyed and recreated due to a configuration change (e.g. orientation change). Using **setArguments()/getArguments()** ensures that when the Fragment needs to be recreated, the Bundle will be appropriately serialized/deserialized so that construction data is restored.
- https://stackoverflow.com/questions/9245408/best-practice-for-instantiating-a-new-ar
- If Android decides to recreate your Fragment later, it's going to call the no-argument constructor of your fragment. So overloading the constructor is not a solution.

- With that being said, the way to pass stuff to your Fragment so that they are available after a Fragment is recreated by Android is to pass a bundle to the setArguments method.
- So, for example, if we wanted to pass an integer to the fragment we would use something like:

```
public static MyFragment newInstance(int someInt) {
    MyFragment myFragment = new MyFragment();
    Bundle args = new Bundle();
    args.putInt("someInt", someInt);
    myFragment.setArguments(args);
    return myFragment;
}
```

- And later in the Fragment onCreate() you can access that integer by using:

```
getArguments().getInt("someInt", 0);
```

- This Bundle will be available even if the Fragment is somehow recreated by Android.
- Also note: setArguments() can only be called before the Fragment is attached to the Activity.

1.2.12 What is the difference between a fragment and an activity? Explain the relationship between the two.

- An activity is typically a single, focused operation that a user can perform (such as dial a number, take a picture, send an email, view a map, etc.).
  - Yet at the same time, there is nothing that precludes a developer from creating an activity that is arbitrarily complex.
- **Activity** implementations can optionally make use of the Fragment class
  - for purposes such as producing more modular code, building more sophisticated user interfaces for larger screens, helping scale applications between small and large screens, and so on.
  - Multiple fragments can be combined within a single activity and, conversely, the same fragment can often be reused across multiple activities.
  - This structure is largely intended to foster code reuse and facilitate economies of scale.
- A **fragment** is essentially a modular section of an activity, with its own lifecycle and input events, and which can be added or removed at will.
  - It is important to remember, though, that a fragment's lifecycle is directly affected by its host activity's lifecycle; i.e., when the activity is paused, so are all fragments in it, and when the activity is destroyed, so are all of its fragments.

1.2.13 Which method is called only once in a fragment life cycle?

- onAttached()

1.2.14 What is the relationship between the life cycle of an AsyncTask and an Activity? What problems can this result in? How can these problems be avoided?

- An AsyncTask is not tied to the life cycle of the Activity that contains it.
  - If the Activity is destroyed and a new instance of the Activity is created, the AsyncTask won't be destroyed.

  - So, for example, if you start an AsyncTask inside an Activity and the user rotates the device, the Activity will be destroyed (and a new Activity instance will be created)
  - But the AsyncTask will not die but instead goes on living until it completes.

- When the AsyncTask does complete, rather than updating the UI of the new Activity, it updates the former instance of the Activity (i.e., the one in which it was created but that is not displayed anymore!).
  - This can lead to an Exception (of the type **java.lang.IllegalArgumentException**: View not attached to window manager if you use, for instance, findViewById to retrieve a view inside the Activity).

- *Since the AsyncTask maintains a reference to the previous instance of the Activity, that Activity won't be garbage collected, resulting in a memory leak.*

- There's also the *potential???* for this to result in **a memory leak** since the AsyncTask maintains a reference to the Activty, which prevents the Activity from being garbage collected as long as the AsyncTask remains alive.

- For these reasons, using AsyncTasks for long-running background tasks is generally a bad idea.
  - Rather, for long-running background tasks, a different mechanism (such as a service) should be employed.

1.2.15 How would you create a multi-threaded Android app without using the Thread class?

- If you only need to override the **run()** method and no other Thread methods, then you should implement Runnable.

- In particular, be on the lookout for an Android developer demonstrating an understanding that you should only extend from a class when you need to modify some of its functionality.

1.2.16 What is a ThreadPool? And is it more effective than using several separate Threads?

- ThreadPool consists of a task queue and a group of worker threads, which allows it to run multiple parallel instances of a task.

- Here, you're assessing the app developer's understanding of how multithreading has the potential to improve an app's performance, but also how it can negatively impact performance when used incorrectly.

- Using ThreadPool is more efficient than having multiple operations waiting to run on a single thread, but it also helps you avoid the considerable overhead of creating and destroying a thread every time you require a worker thread.

1.2.17 What are "launch modes"? What are the two mechanisms by which they can be defined? What specific types of launch modes are supported?

- A "launch mode" is the way in which a new instance of an activity is to be associated with the current task.

- Launch modes may be defined using one of two mechanisms:
  - Manifest file. When declaring an activity in a manifest file, you can specify how the activity should associate with tasks when it starts. Supported values include:

    * **standard** (default). Multiple instances of the activity class can be instantiated and multiple instances can be added to the same task or different tasks. This is the common mode for most of the activities.
    * **singleTop**. The difference from standard is, if an instance of the activity already exists at the top of the current task and the system routes the intent to this activity, no new instance will be created because it will fire off an onNewIntent() method instead of creating a new object.
    * **singleTask**. A new task will always be created and a new instance will be pushed to the task as the root. However, if any activity instance exists in any tasks, the system routes the intent to that activity instance through the onNewIntent() method call. In this mode, activity instances can be pushed to the same task. This mode is useful for activities that act as the entry points.
    * **singleInstance**. Same as singleTask, except that the no activities instance can be pushed into the same task of the singleInstance's. Accordingly, the activity with launch mode is always in a single activity instance task. This is a very specialized mode and should only be used in applications that are implemented entirely as one activity.
  - Intent flags. Calls to startActivity() can include a flag in the Intent that declares if and how the new activity should be associated with the current task. Supported values include:
    * **FLAG_ACTIVITY_NEW_TASK**. Same as **singleTask** value in Manifest file (see above).
    * **FLAG_ACTIVITY_SINGLE_TOP**. Same as **singleTop** value in Manifest file (see above).
    * **FLAG_ACTIVITY_CLEAR_TOP**. If the activity being started is already running in the current task, then instead of launching a new instance of that activity, all of the other activities on top of it are destroyed and this intent is delivered to the resumed instance of the activity (now on top), through onNewIntent(). There is no corresponding value in the Manifest file that produces this behavior.

1.3 Intent vs ContentProvider

1.3.1 What is Intent and brief about it types as well?

- The intent is messaging objects. If a developer is trying to pass the data from one screen to another screen they will be using the Intent. Talking of the types there are of 2 types:
  - **Implicit**: These calls the system components.
  - **Explicit**: These invoke the activity class.

1.3.2 What's the difference between an implicit and an explicit intent?

- An **explicit intent** is where you tell the system which Activity or system component it should use to respond to this intent.
- **Implicit intents** allow you to declare the action you want to perform; the Android system will then check which components are registered to handle that action.
- Here, you're looking for an understanding of when you should use each type of intent,
  - as the vast majority of the time you'll use explicit intents to start components in your own application,
  - while implicit intents are most commonly used to communicate with components from other third party applications.

1.3.3 Describe three common use cases for using an Intent.

- Common use cases for using an Intent include:
  - To **start an activity**: You can start a new instance of an Activity by passing an Intent to **startActivity()** method.
  - To **start a service**: You can start a service to perform a one-time operation (such as download a file) by passing an Intent to **startService()**.
  - To **deliver a broadcast**: You can deliver a broadcast to other apps by passing an Intent to **sendBroadcast()**, **sendOrderedBroadcast()**, or **sendStickyBroadcast()**.

1.3.4 What is a ContentProvider and what is it typically used for?

- A **ContentProvider** manages access to a structured set of data.
- It encapsulates the data and provide mechanisms for defining data security.
- **ContentProvider** is the standard interface that connects data in one process with code running in another process.

1.3.5 How would you access data in a ContentProvider?

- Start by making sure your Android application has the necessary read access permissions.
- Then, get access to the ContentResolver object by calling **getContentResolver()** on the Context object, and retrieving the data by constructing a query using **ContentResolver.query()**.
- The **ContentResolver.query()** method returns a Cursor, so you can retrieve data from each column using Cursor methods.
- Accessing data is one of the tasks that's most likely to block the main thread, so it is very importance to **perform data queries on a separate thread**.

1.3.6 What is an Intent? Can it be used to provide data to a ContentProvider? Why or why not?

- The **Intent object** is a common mechanism for starting new activity and transferring data from one activity to another. However, you cannot start a ContentProvider using an Intent.
- When you want to access data in a **ContentProvider**, you must instead use the **ContentResolver object** in your application's Context to communicate with the provider as a client.
  - The **ContentResolver object** communicates with the provider object, an instance of a class that implements ContentProvider.
  - The provider object receives data requests from clients, performs the requested action, and returns the results.

1.4 Service

1.4.1 What is service in Android and what are their types?

- A **service** is an application component that can perform long-running operations in the background, and it does not provide a user interface. There are 3 types of services available
  - **Scheduled**: A service is scheduled when an API such as **JobScheduler** launches the service.

- **Started**: A service is started when an application component (means activity) calls **startService()**. After service started it can run in the background indefinitely, even if the components that started it are destroyed. It is stopped by **stopService()** method. The service can stop itself by calling the **stopSelf()** method.
- **Bound**: A service is bound when an application component binds to it by calling the **bindService()**. A bound service offers a client-server interface that allows components to interact with the service, send requests, receive a request. The client can unbind the service by calling the **unbindService()** method. The service cannot be stopped until all the clients unbind the service.

1.4.2 What is the difference between Service and IntentService? How is each used?

- **Service** is the base class for Android services that can be extended to create any service. A class that directly extends Service runs on the main thread so it will block the UI (if there is one) and should therefore either be used only for short tasks or should make use of other threads for longer tasks.
- **IntentService** is a subclass of Service that handles asynchronous requests (expressed as "Intents") on demand. Clients send requests through startService(Intent) calls. The service is started as needed, handles each Intent in turn using a worker thread, and stops itself when it runs out of work. Writing an IntentService can be quite simple; just extend the IntentService class and override the onHandleIntent(Intent intent) method where you can manage all incoming requests.

1.4.3 What is a broadcast receiver?

- The broadcast receiver communicates with the operation system messages such as "check whether an internet connection is available," what the battery label should be, etc.

1.4.4 Suppose that you are starting a service in an Activity as follows:

```
1 Intent service = new Intent(context, MyService.class);
2 startService(service);
```

- where MyService accesses a remote server via an Internet connection.
- If the Activity is showing an animation that indicates some kind of progress, what issue might you encounter and how could you address it?
- **Answer**:
- Responses from a remote service via the Internet can often take some time, either due to networking latencies, or load on the remote server, or the amount of time it takes for the remote service to process and respond to the request.
- As a result, if such a delay occurs, the animation in the activity (and even worse, the entire UI thread) could be blocked and could appear to the user to be "frozen" while the client waits for a response from the service. This is because the service is started on the main application thread (or UI thread) in the Activity.
- The problem can (and should) be avoided by relegating any such remote requests to a background thread or, when feasible, using an an asynchronous response mechanism.
- Note well: Accessing the network from the UI thread throws a runtime exception in newer Android versions which causes the app to crash.

1.4.5 What do you mean by AIDL? What are the data types supported in AIDL?

- AIDL stands for Android Interface Definition Language. It acts as an interface between client and service and enables and facilitates the communication between them. It handles interface requirements between both of them and handles communication through interprocess communication or IPC. This involves breaking the objects into smaller parts so that Android can understand those objects. This happens because a process cannot access memory of other processes that are running. The different data types supported by AIDL are:
  - **Strings**
  - **List**
  - **Map**
  - **charSequence**
  - **Java data types**: all Java data types like int, long, char, Boolean.

1.4.6 What are the different ways to define the service's IBinder interface and how the client receive it and make a connection?

- The different ways to define service's IBinder interface and pass it to the client (just like activities) are:
  - Extending the Binder class
  - Using a Messenger
  - Android Interface Definition Language (AIDL)
- For extending the Binder class – here if the services are private to the applications and run in the same process as the client then interface be created by extending the Binder class and returning an instance of it from onBind(). The client receives the Binder and can use it to directly access public method available in either the Binder implementation or the Service.

1.4.7 What are the steps involved in creating a bound service through Android Interface Definition Language (AIDL)?

- **Define** an AIDL interface in an .aidl file.
- **Save** this file in the src/ directory of the application hosting the Activity and any other application that needs to bind to this service —the latter is particularly important, and is often overlooked.
- **Build** your application. Android SDK tools will then generate an IBinder interface file in your gen directory.
- **Implement** this interface, by extending the generated Binder interface and implementing the methods inherited from the .aidl file.
- **Extend** Service and override onBind() to return your implementation of the Stub class.

1.5 Sensor

1.5.1 There are four Java classes related to the use of sensors on the Android platform. List them and explain the purpose of each.

- The four Java classes related to the use of sensors like the accelerometer or gyroscope (加速度计或陀螺仪) on the Android platform are:
  - **Sensor**: This class creates an instance of a specific sensor,

    * providing methods to identify which capabilities are available for a specific sensor.
  - **SensorManager**: Provides methods for
    * registering sensor event listeners,
    * the management of data acquisition, and sensor calibration.
    * It also provides methods for accessing and listing sensors.
  - **SensorEvent**:
    * This class provides information on a sensor event by creating a sensor event object.
    * It provides raw sensor data, including information regarding accuracy.
  - **SensorEventListener**: Interface that defines callback methods that will receive sensor event notifications.

1.5.2 How would you check for the presence of a Compass sensor on the system using the hasSystemFeature() method?

- While it may be tempting to call this method on SensorManager or Sensor, as they both come as part of the Android Sensor Framework, neither of those classes provide the hasSystemFeature() method.
- These classes are intended for direct access and acquisition of raw sensor data.
- When it comes to evaluating a system's capabilities, the PackageManager class can be used to retrieve information on application packages available on a given device. One possible solution to this problem is reproduced below.

```
PackageManager myCompass = getPackageManger();
If (!myCompass.hasSystemFeature(PackageManager.FEATURE_SENSOR_COMPASS)) {
    // This device lacks a compass, disable the compass feature
}
```

1.5.3 Which of the code snippets below is the correct way to check if a Compass sensor is present on the system? Explain your answer.

- Answer 1:

```
PackageManager m = getPackageManager();
if (!m.hasSystemFeature(PackageManager.FEATURE_SENSOR_COMPASS)) {
    // This device does not have a compass, turn off the compass feature
}
```

- Answer 2:

```
SensorManager m = getSensorManager();
if (!m.hasSystemFeature(SensorManager.FEATURE_SENSOR_COMPASS)) {
    // This device does not have a compass, turn off the compass feature
}
```

- Answer 3:

```
Sensor s = getSensor();
if (!s.hasSystemFeature(Sensor.FEATURE_SENSOR_COMPASS)) {
    // This device does not have a compass, turn off the compass feature
}
```

- The correct answer is Answer 1, the version that uses PackageManager.

- SensorManager and Sensor are part of Android Sensor Framework and are used for direct access and acquisition of raw sensor data. These classes do not provide any method like **hasSystemFeature()** which is used for evaluation of system capabilities.
- Android defines **feature IDs**, in the form of **ENUMs**, for any hardware or software feature that may be available on a device. For instance, the feature ID for the compass sensor is **FEATURE_SENSOR_COMPASS**.
- If your application cannot work without a specific feature being available on the system, you can prevent users from installing your app with a **<uses-feature>** element in your app's manifest file to specify a non-negotiable dependency.
- However, if you just want to disable specific elements of your application when a feature is missing, you can use the PackageManager class. **PackageManager** is used for retrieving various kinds of information related to the application packages that are currently installed on the device.

1.6 ANR & Crashes

1.6.1 What is ANR? What are the precautions to be taken to avoid ANR in an application?

- This is the basic Android interview questions asked in an interview. ANR is a dialog which Android shows when an application is not responding. It stands for Application Not Responding. Usually, this state is encountered when an application is performing many tasks on the main thread and it has been unresponsive for a long period of time.
- Following things can be taken into mind to avoid ANR:
  - Be careful that there are no infinite loops encountered when complex calculations are involved.
  - When a server is not responding for a long time and can result in ANR. In order to avoid this developer should define HTTP timeout for all web service and API calls.
  - A developer should use IntentService when there are many background tasks. They should be taken off the main UI thread.
  - All database and long-running network operations should be run on a different thread.

1.6.2 What is ANR, and why does it happen?

- "ANR" in Android is "Application Not Responding". It means when the user is interacting with the activity, and the activity is in the onResume() method, a dialog appears displaying "application not responding."
- It happens because we start a heavy and long running task like downloading data in the main UI thread. The solution of the problem is to start your heavy tasks in the backbround using Async Task class.

1.6.3 What are some measures you can take to avoid ANR?

- The dreaded ANR (Application Not Responding) message appears to the user when an Android application remains unresponsive for a long period of time. ANR is typically caused when the app performs too much on the main thread. To avoid ANR, an app should perform lengthy database or networking operations in separate threads. For background task-intensive apps, you can alleviate pressure from the UI thread by using the IntentService. In general, it helps to always define time-outs for all your web service calls and to remain ever vigilant for infinite loops in complex calculations.

1.6.4 What is an Application Not Responding (ANR) error, and how can you prevent them from occurring in your app?

- This question checks whether the developer is aware of the golden rule of threading on Android: never perform lengthy or intensive operations on the main thread.
- An ANR dialog appears when your UI has been unresponsive for more than 5 seconds, usually because you've blocked the main thread. To avoid encountering ANR errors, you should move as much work off the main thread as possible.

1.6.5 Under what condition could the code sample below crash your application? How would you modify the code to avoid this potential problem? Explain your answer.

```
Intent sendIntent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType(HTTP.PLAIN_TEXT_TYPE); // "text/plain" MIME type
startActivity(sendIntent);
```

1.7 Views

1.7.1 Write a code to generate a button dynamically.

```
protected void onCreate(Bundle newInstanceState) {
    super.onCreate(newInstanceState);
    Button button = new Button(this);
    button.setText("Button");
    setContentView(button);
});
```

1.7.2 How to handle multiple resolution screens in Android?

- Below five properties help in handling multiple screen resolutions in Android:
  - **Screen size** can be divided into *four generalized categories* like *small*, *normal*, *large* and *extra large*
  - **Screen density** can also be categorized into *low*, *medium*, *high* and *extra high*.
  - When a user **rotates the screen** the *screen orientation device should also get changed*.
  - The **resolution** defines the *physical pixels on a screen*.
  - **Independent pixel** provides the developer a density-independent way to define the various layouts as per the requirement. The layouts can be customized and used accordingly.

1.7.3 When might you use a FrameLayout?

- Here, you're looking for an understanding that you should always use the simplest layout possible for what you want to achieve, as FrameLayouts are designed to contain a single item, making them an efficient choice when you need to display a single View.
- If you add multiple Views to a FrameLayout then it'll stack them one above the other, so FrameLayouts are also useful if you need overlapping Views, for example if you're implementing an overlay or a HUD element.

1.7.4 Write a code for a Toast that will display the message "Hello, this is a Toast".

```
Toast.makeText(getApplicationContext(), "Hello, this is a Toast", Toast.LENGTH_LONG).show
```

1.7.5 Normally, in the process of carrying out a screen reorientation, the Android platform tears down the foreground activity and recreates it, restoring each of the view values in the activity's layout.

- In an app you're working on, you notice that a view's value is not being restored after screen reorientation. What could be a likely cause of the problem that you should verify, at a minimum, about that particular view?
- You should verify that it has a valid id. In order for the Android system to restore the state of the views in your activity, each view must have a unique ID, supplied by the **android:id** attribute.

1.7.6 Briefly describe some ways that you can optimize View usage.

- There are a number of methods, but the ones that tend to have the most impact are:
  - **Checking for excessive overdraw**: install your app on an Android device, and then enable the "Debug GPU Overview" option.
  - **Flattening your view hierarchy**: inspect your view hierarchy using Android Studio's 'Hierarchy Viewer' tool.
  - Measuring how long it takes each View to complete the measure, layout, and draw phases. You can also use Hierarchy Viewer to identify any parts of the rendering pipeline that you need to optimize.

1.7.7 What is an Adapter?

- Here, you're checking that the Android eveloper understands that you need an additional component to connect an AdapterView (such as ListView or GridView), to an external data source. An Adapter acts as this bridge, and is also responsible for converting each data entry into a View that can then be added to the AdapterView.

1.7.8 Outline the process of creating custom Views

- This is a complex topic, so you're only looking for a high-level overview of the steps involved. However, the developer should make it clear that you should always subclass the View that most closely resembles the custom component you want to create —very rarely would you extend the View class.
- After extending your class, you need to complete the following steps:
  - Create a res/values/attrs.xml file and declare the attributes you want to use with your custom View.
  - In your View class, add a constructor method, instantiate the Paint object, and retrieve your custom attributes.
  - Override either onSizeChanged() or onMeasure().
  - Draw your View by overriding onDraw().

1.7.9 What are the major difference between ListView and RecyclerView?

- There are many differences between ListView and RecyclerView, but the Android developer should be aware of the following in particular:
  - The ViewHolder pattern is entirely optional in ListView, but it's baked into RecyclerView.
  - ListView only supports vertical scrolling, but RecyclerView isn't limited to vertically scrolling lists.

1.7.10 What is a Handler typically used for?

- You use Handler to communicate between threads, most commonly to pass an action from a background thread to Android's main thread.
- This question allows you to check that the developer understands another fundamental concept of multithreading in Android: you cannot update the UI from any thread other that the main thread.

1.8 Other

1.8.1 What is SQLite? How does it differ from client-server database management systems?

- SQLite is the open-source relational database of choice for Android applications. The SQLite engine is serverless, transactional, and self-contained. Instead of the typical client-server relationship of most database management systems, the SQLite engine is integrally linked with the application. The library can also be called dynamically, and makes use of simple function calls that reduce latency in database access.

1.8.2 What is a BuildType in Gradle? And what can you use it for?

- Build types define properties that Gradle uses when building and packaging your Android app.
- This question allows you to check that the developer can differentiate between product flavors, build variants, and build types, as these are very similar concepts that are a common source of confusion:
  - A **build type** defines how a module is built, for example whether ProGuard is run.
  - A **product flavor** defines what is built, such as which resources are included in the build.
  - **Gradle** creates a build variant for every possible combination of your project's product flavors and build types.

1.9 the candidates must be well rehearsed in the below-listed details –

- Try to get more familiar with the Android Framework internals.
- No missing out fear.
- Start reading and practicing a lot more code.
- Try considering learning lot more languages.
- Try to contribute to the fullest to the open-source community.
- IDE must work for you.
- Architecture knowledge is must to have for better app design.
- Android best practices journals and magazines are available in the market (means web), try to read them on a regular basis.

1.9.1 Why do you find yourself fit for the position of an android developer? Or why should we hire you.?

- One of the prominent questions that are most likely to be asked. Be prepared to talk about yourself and why you should be the best candidate to be hired. This is one way the interviewer wants to know you to evaluate. One good approach will be to talk about the interesting one has in the field of Android development. Brief about yourself and how you have achieved your career growth so far, brief your current position, skills, and passion and then finish by touching the goal of the future. A bonus will be to identify the position you are applying, and future envision.

1.9.2 Highlights and brief about some of your professional strength?

- Be accurate and relevant is the key to this answer. Relate your experience with real scenarios and what you learned from this. These Android developer Interview questions are also intended to analyze the candidate's interest and learning attitude. First and foremost, thing – behavior that one share in the office environment should be quoted by the candidate. Apart from these the below mentioned can be of great importance –
  - Learning attitude
  - Creative thinking
  - Solution approach
  - Team player
- Note – The candidate must relate by citing real-life scenario and how this behavior has helped him/her in achieving the technical efficiency and boosted professionally. No irrelevant strength that will add no value to the job.
- Let us move to the next Android developer Interview Questions.