|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | Android | <chare> | 1 | <pext> | <starting>Android</heading> Android is an open-sourced operating system that is used on mobile devices,  such as mobiles and tablets.  The Android application executes  within its own process and its own instance of Dalvik Virtual Machine(DVM)  or Android RunTime(ART). </ending> | </end> |
| <hitle> | Activity | <chare> | 2 | <pext> | <starting>Activity</heading> Activity is a single screen that represents GUI(Graphical User Interface)  with which users can interact in order to do something  like dial the phone, view email, etc. </ending> | </end> |
| <hitle> | Service | <chare> | 3 | <pext> | <starting>Service</heading> Service is an application component that facilitates an application  to run in the background in order to perform long-running operations  without user interaction.  A service can run continuously in the background  even if the application is closed  or even after the user switches to another application. 1. Foreground Service: Operations that are immediately recognizable to users  2. Background Service:  Operations that are not immediately recognizable to users 3. Bound Service:  Services that provide a client-server interface for component interactions </ending> | </end> |
| <hitle> | Activity vs Service | <chare> | 2 | <pext> | <starting>Activity vs Service</heading> 1. Activity is designed to run in the foreground. 1-1. Service is mainly designed to run in the background.  Foreground services are also available. 2. Activity is used when the user interface is necessary. 2-1. Service is used when the user interface is not necessary. 3. Activity is dependent.  3-1. Service act independently. </ending> | </end> |
| <hitle> | Bundle | <chare> | 3 | <pext> | <starting>Bundle</heading> Bundles are used to pass the required data between various Android activities. These are like HashMap that can take trivial data types. </ending> | </end> |
| <hitle> | Adapter | <chare> | 1 | <pext> | <starting>Adapter</heading> An adapter in Android acts as a bridge between an AdapterView  and the underlying data for that view.  The adapter holds the data and sends the data to the adapter view,  the view can take the data from the adapter view  and shows the data on different views like a spinner, list view, grid view, etc. </ending> | </end> |
| <hitle> | DDMS | <chare> | 2 | <pext> | <starting>DDMS</heading> 1. Port forwarding services. 2. Thread and heap information. 3. Logcat. 4. Screen capture on the device. 5. Network traffic tracking. 6. Incoming call and SMS spoofing. 7. Location data spoofing. </ending> | </end> |
| <hitle> | Activity Lifecycle | <chare> | 3 | <pext> | <starting>Activity Lifecycle</heading> 1. OnCreate(): It is called when activity is created.  Using this, the views are created and data is collected from bundles. 2. OnStart(): It is called if the activity is becoming visible to the user.  It may be succeeded by onResume() if the activity comes to the foreground,  or onStop() if it becomes hidden. 3. OnResume(): It is called when the activity will start an interaction  with the user. 4. OnPause(): This is called when the activity is moving to the background  but hasn’t been killed yet. 5. OnStop(): This is called when an activity is no longer visible to the user. 6. OnDestroy(): This is called when the activity is finished or destroyed. 7. OnRestart(): This is called after the activity has been stopped,  prior to it being started again.  Differentiate: onCreate() is the first method that’s invoked  when an activity is launched for the first time.  onStart() is invoked after onCreate() has completed it’s task.  onResume() is called after onStart() has completed.  When an activity leaves its foreground  (probably for a smaller duration such as standby/sleep)   onPause() is invoked followed by onStop() (when the activity is not visible. eg. some other application is launched). onDestroy() is called when the activity or application is killed.  Essentially the lifecycle methods are divided into three layers of duration : 1. onCreate() and onDestroy() are present  during the entire duration of the activity 2. onStart() and onStop() are present while the activity is visible 3. onResume() and onPause() are present while the activity is in foreground </ending> | </end> |
| <hitle> | Sensor | <chare> | 1 | <pext> | <starting>Sensor</heading> Android-based devices have a collection of built-in sensors in them,  which measure certain parameters like motion, orientation,  and many more through their high accuracy.  The sensors can be both hardware and software based on nature.  There are three prominent categories of sensors in Android devices.  They are: 1. Position Sensor:  It is used for measuring the physical position of the Android device.  This has orientation sensors and magnetometers. 2. Motion Sensors: These sensors consist of gravity, rotational activity,   and acceleration sensors  which measure the rotation of the device or the acceleration, etc. 3. Environmental Sensor: It includes sensors that measure temperature,  humidity, pressure, and other environmental factors. </ending> | </end> |
| <hitle> | dialog boxes | <chare> | 1 | <pext> | <starting>dialog boxes</heading> 1. AlertDialog: The AlertDialog supports 0-3 buttons,  along with a list of selectable items such as checkboxes and radio buttons. It is used when you want to ask the user  about taking a decision between yes or no in response  to any particular action taken by the user,  by remaining in the same activity and without changing the screen. 2. DatePickerDialog: It is used for selecting the date by the user. 3. TimePickerDialog: Used for selecting the time by the user. 4. ProgressDialog: It is an extension of the AlertDialog and is used to display a progress bar.  It also supports the addition of buttons. This class was deprecated in API level 26 because it prevents the user from  interacting with the application. Instead of this class, we can use a progress  indicator such as ProgressBar,  which can be embedded in the user interface of your application. </ending> | </end> |
| <hitle> | AndroidManifest.xml | <chare> | 1 | <pext> | <starting>AndroidManifest.xml</heading> 1. The AndroidManifest.xml file contains information  regarding the application that the Android system must know  before the codes can be executed. 2. This file is essential in every Android application. 3. It is declared in the root directory. 4. This file performs several tasks such as: 4-1. Providing a unique name to the java package. 4-2. Describing various components of the application  such as activity, services, and many more. 4-3. Defining the classes which will implement these components. </ending> | </end> |
| <hitle> | intent | <chare> | 1 | <pext> | <starting>intent</heading> An intent is a messaging object that is used  to request an action from other components of an application.  It can also be used to launch an activity, send SMS, send an email,  display a web page, etc. It shows notification messages to the user. It alerts the user of a particular state that occurred.  There are two types of intents in Android: 1. Implicit Intent- Used to invoke the system components. 2. Explicit Intent- Used to invoke the activity class. </ending> | </end> |
| <hitle> | Toast | <chare> | 1 | <pext> | <starting>Toast</heading> Toast is a message that pops up on the screen.  It is used to display the message regarding the status of the operation  initiated by the user and covers only the expanse of space  required for the message while the user’s recent activity  remains visible and interactive. Toast notification automatically fades in and out  and it does not accept interaction events. </ending> | </end> |
| <hitle> | context | <chare> | 1 | <pext> | <starting>context</heading> The context in Android is the context of the current state  of the application or object.  The context comes with services like giving access to databases  and preferences, resolving resources, and more. There are two types of context. They are: 1. Activity context 1-1. This activity context is attached to the lifecycle of an activity. 1-2. The activity context can be used when you are passing the context  in the scope of an activity or you need the context  whose lifecycle is attached to the context of the activity. 2. Application context: 2-1. This application context is attached to the lifecycle of an application. 2-2. The application context should be used where you need a context  whose lifecycle is separate from the current context  or when you are passing a context beyond the scope of activity. </ending> | </end> |
| <hitle> | Intent- Implicit vs Explicit | <chare> | 1 | <pext> | <starting>Intent - Implicit vs Explicit</heading> Explicit Intent: An Explicit Intent is where you inform the system  about which activity should handle this intent.  Here target component is defined directly in the intent. Implicit Intent: An Implicit Intent permits you to declare the action you want to carry out.  Further, the Android system will check  which components are registered to handle that specific action  based on intent data.  Here target component is not defined in the intent. </ending> | </end> |
| <hitle> | launch mode | <chare> | 1 | <pext> | <starting>launch mode</heading> 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: 1. 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 are: 1-1. 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. 1-2. 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. 1-3. 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. 1-4. 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. Another mechanism to defione Launch modes is 2. 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 are: 2-1. FLAG\_ACTIVITY\_NEW\_TASK.  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. 2-2. FLAG\_ACTIVITY\_SINGLE\_TOP.  Same as singleTop value in Manifest file. 2-3. 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,  through onNewIntent().  There is no corresponding value in the Manifest file  that produces this behavior. </ending> | </end> |
| <hitle> | Container | <chare> | 1 | <pext> | <starting>Container</heading> Containers carry objects and widgets together,  based on which specific items are required  and in what particular arrangement is needed.  Containers may hold labels, buttons, fields, or even child containers, etc.  For example, if you want a form with fields on the left and labels on the right,  you will need a container.  If you want the OK and Cancel buttons to be below the rest of the form,  next to one another, and flush to the right side of the screen,  you will need a container.  If you have several widgets, you will need a container to have a root element  to place the widgets inside. Android provides a collection of view classes  that serve as containers for views.  These container classes are called layouts,  which are defined in the form of XML files  that cannot be changed by our code during execution.  The layout managers provided by Android SDK are LinearLayout,  RelativeLayout, FrameLayout, AbsoluteLayout, GridLayout, and TableLayout. </ending> | </end> |
| <hitle> | broadcast receiver | <chare> | 1 | <pext> | <starting>broadcast receiver</heading> A broadcast receiver is a mechanism used for listening to system-level events like listening for incoming calls, SMS, etc. by the host application. It is implemented as a subclass of BroadcastReceiver class and each message is broadcasted as an intent object. The Broadcast Receiver is defined inside the receiver tags with the necessary actions defined inside the intent filter as shown below. ``` <receiver android:name=".ConnectionReceiver" >  <intent-filter>  <action android:name="android.net.conn.CONNECTIVITY\_CHANGE" />  </intent-filter> </receiver> ``` </ending> | </end> |
| <hitle> | Serializable vs Parcelable | <chare> | 1 | <pext> | <starting>Serializable vs Parcelable</heading> While developing applications  usually it needs to transfer data from one activity to another.  This data needs to be added into a corresponding intent object.  Some additional actions are required to make the data suitable for transfer.  For doing that the object should be either serializable or parcelable. 1. Serializable: 1-1. Serializable is a standard Java interface.  In this approach, you simply mark a class Serializable  by implementing the interface and java will automatically serialize it. 1-2. Reflection is used during the process  and many additional objects are created.  This leads to plenty of garbage collection and poor performance. 2. Parcelable: 2-1. Parcelable is an Android-specific interface.  In this approach, you implement the serialization yourself. 2-2. Reflection is not used during this process and hence no garbage is created. 2-3. Parcelable is far more efficient than Serializable  since it gets around some problems with the default Java serialization scheme. Also, it is faster because it is optimized for usage  on the development of Android, and shows better results. </ending> | </end> |
| <hitle> | SQLite database in android | <chare> | 1 | <pext> | <starting>SQLite database in android</heading> SQLite is the open-source relational database used in Android.  The SQLite engine is serverless, transactional, and also self-contained.  Instead of the client-server relationship  of most database management systems,  the SQLite engine is integrally linked with the application.  The library can be called dynamically   and it can make use of simple function calls  that reduce latency in database access. </ending> | </end> |
| <hitle> | Service vs Thread | <chare> | 1 | <pext> | <starting>Service vs Thread</heading> 1. Service is an application component that facilitates an application to run in the background in order to perform long-running operations without user interaction. / A Thread is a concurrent unit of execution. 2. Service exposes few functionalities to other applications by calling Context.bindService(). / For Thread, Google has brought in handlers and loopers into threads. 3. When an application is killed, service is not killed. / When an application is killed, the thread is killed. </ending> | </end> |
| <hitle> | Content provider | <chare> | 1 | <pext> | <starting>Content provider</heading> Content provider is one of the primary building blocks of Android applications,  which manages access to a central repository of data.  It acts as a standard interface that connects data in one process  with code running in another process.  So it can be used to share the data between different applications. They are responsible for encapsulating the data and providing mechanisms  for defining data security.  It is implemented as a subclass of ContentProviderclass  and must implement a set of APIs that will enable other applications  to perform transactions. </ending> | </end> |
| <hitle> | compileSdkVersion vs targetSdkVersion | <chare> | 1 | <pext> | <starting>compileSdkVersion vs targetSdkVersion</heading> 1. compileSdkVersion: 1-1. The compileSdkVersion is the version of API  the application is compiled against.  You can use Android API features involved in that version of the API  (as well as all previous versions). 1-2. For example, if you try and use API 15 features  but set compileSdkVersion to 14, you will get a compilation error.  If you set compileSdkVersion to 15 you can still run the app   on an API 14 device as long as your app’s execution paths   do not attempt to invoke any APIs specific to API 15. 2. targetSdkVersion: 2-1. The targetSdkVersion indicates  that you have tested your app on the version you specify.  This is like a certification or sign-off you are giving the Android OS  as a hint to how it should handle your application in terms of OS features. 2-2. For example, setting the targetSdkVersion value to “11” or higher permits  the system to apply a new default theme (Holo) to the application when running on Android 3.0 or higher. It also disables screen compatibility mode when running on larger screens (because support for API level 11 implicitly supports larger screens). </ending> | </end> |
| <hitle> | Sensor class | <chare> | 1 | <pext> | <starting>Sensor class</heading> 1. Sensor class: This class helps you to create an instance of a specific sensor.  It provides methods that let you determine a sensor’s capabilities. 2. SensorManager class:  This class is used to create an instance of the sensor service.  It provides methods to access and list sensors,  to register and unregister sensor listeners, etc. 3. SensorEvent class: This Java class is used to create a sensor event object.  It provides information about the sensor event including raw sensor data,  the accuracy of data, type of sensor, timestamp of event, etc. 4. SensorEventListener interface:  This interface is used to create two callback methods  that receive sensor event notifications when sensor value changes  or when sensor accuracy changes.  Those two methods are onAccuracyChanged which is called  when sensor accuracy is changed and  onSensorChanged which is called when sensor values are changed. </ending> | </end> |
| <hitle> | JobSchedular | <chare> | 1 | <pext> | <starting>JobSchedular</heading> The JobSchedular API is used for scheduling different types of jobs  against the framework that will be executed in your app’s own process.  This allows your application to perform the given task  while being considerate of the device’s battery at the cost of timing control. The JobScheduler supports batch scheduling of jobs.  The Android system can combine jobs for reducing battery consumption. JobManager automatically handles the network unreliability  so it makes handling uploads easier. Some of situations where you would use this job scheduler are: 1. There are tasks that should be done when the device is connected  to a power supply. 2. There are tasks that require a Wi-Fi connection or network access. 3. There are tasks that should run on a regular basis as batch  where the timing is not critical. </ending> | </end> |
| <hitle> | Android application Architecture | <chare> | 1 | <pext> | <starting>Android application Architecture</heading> 1. Services: Used to perform background functionalities. 2. Intent: Used to perform the interconnection between activities  and the data passing mechanism. 3. Resource Externalization: strings and graphics. 4. Notification: light, sound, icon, notification, dialog box and toast. 5. Content Providers: It will share the data between applications. </ending> | </end> |
| <hitle> | storage in Android | <chare> | 1 | <pext> | <starting>storage in Android</heading> 1. Shared Preferences 2.Internal Storage 3. External Storage 4. SQLite Databases 5. Network Connection </ending> | </end> |
| <hitle> | fragment | <chare> | 1 | <pext> | <starting>fragment</heading> In Android, the fragment is the part of Activity that represents a portion  of the User Interface(UI) on the screen.  It is the modular section of the android activity that is very helpful  in creating UI designs that are flexible in nature and auto-adjustable  based on the device screen size.  There are mainly 3 types of fragments: Single Fragment List Fragment Fragment Transaction </ending> | </end> |
| <hitle> | fragment lifecycle | <chare> | 1 | <pext> | <starting>fragment lifecycle</heading> 1. onAttach() The very first method to be called  when the fragment has been associated with the activity.  This method executes only once during the lifetime of a fragment.  2. onCreate() This method initializes the fragment  by adding all the required attributes and components. 3. onCreateView() System calls this method  to create the user interface of the fragment.  The root of the fragment’s layout  is returned as the View component by this method to draw the UI. 4. onActivityCreated() It indicates that the activity has been created  in which the fragment exists.  View hierarchy of the fragment also instantiated before this function call.  5. onStart() The system invokes this method  to make the fragment visible on the user’s device. 6. onResume() This method is called to make the visible fragment interactive. 7. onPause() It indicates that the user is leaving the fragment.  System calls this method to commit the changes made to the fragment.  8. onStop() Method to terminate the functioning and visibility of fragments  from the user’s screen.  9. onDestroyView() System calls this method to clean up all kinds of resources  as well as view hierarchy associated with the fragment. 10. onDestroy() It is called to perform the final clean-up  of the fragment’s state and its lifecycle. 11. onDetach() The system executes this method  to disassociate the fragment from its host activity. </ending> | </end> |
| <hitle> | Android SDK | <chare> | 1 | <pext> | <starting>Android SDK</heading> The Google Android SDK is a toolset which is used by developers  to write apps on Android-enabled devices.  It contains a graphical interface  that emulates an Android-driven handheld environment  and allows them to test and debug their codes. </ending> | </end> |
| <hitle> | NDK | <chare> | 1 | <pext> | <starting>NDK</heading> NDK stands for Native Development Kit.  By using NDK, you can develop a part of an app using native language  such as C/C++ to boost the performance. The NDK is a tool that allows you to program in C/C++ for Android devices.  It provides platform libraries one can use to manage native activities  and access physical device components,  such as sensors and touch input.  NDK can be used: 1. to squeeze extra performance out of a device to achieve low latency  or run computationally intensive applications,  such as games or physics simulations. 2. to reuse your own or other developers’ C or C++ libraries. </ending> | </end> |
| <hitle> | View Group | <chare> | 1 | <pext> | <starting>View Group</heading> View Group is a collection of views and other child views.  It is an invisible part and the base class for layouts. </ending> | </end> |
| <hitle> | application Widgets in android | <chare> | 1 | <pext> | <starting>application Widgets in android</heading> Application widgets are miniature application views  that can be embedded in other applications and receive periodic updates. </ending> | </end> |
| <hitle> | singleton | <chare> | 1 | <pext> | <starting>singleton</heading> A singleton class is a class which can create only an object  that can be shared by all other classes. </ending> | </end> |
| <hitle> | sleep mode | <chare> | 1 | <pext> | <starting>sleep mode</heading> In sleep mode, CPU is slept and doesn't accept any commands  from android device except Radio interface layer and alarm. </ending> | </end> |
| <hitle> | drawable folder | <chare> | 1 | <pext> | <starting>drawable folder</heading> In Android, a drawable folder is compiled a visual resource  that can use as a background, banners, icons, splash screen, etc. </ending> | </end> |
| <hitle> | exceptions in Android | <chare> | 1 | <pext> | <starting>exceptions in Android</heading> 1. Inflate Exception 2. Surface.OutOfResourceException 3. SurfaceHolder.BadSurfaceTypeException 4. WindowManager.BadTokenException </ending> | </end> |
| <hitle> | Activity vs AppCompatActivity | <chare> | 1 | <pext> | <starting>Activity vs AppCompatActivity</heading> AppCompatActivity provides native ActionBar support  that is consistent across the application.  Also, it provides backward compatibility for other material design components  till SDK version 7(ActionBar was natively available since SDK 11).  Extending an Activity doesn’t provide any of these.  Note: Since SDK 21 every activity by default, extends AppCompatActivity. </ending> | </end> |
| <hitle> | Activity, AppCompatActivity, FragmentActivity and ActionBarActivity | <chare> | 1 | <pext> | <starting>Activity, AppCompatActivity, FragmentActivity and ActionBarActivity</heading> Activity is the base class.  FragmentActivity extends Activity.  AppCompatActivity extends FragmentActivity.  ActionBarActivity extends AppCompatActivity.  FragmentActivity is used for fragments.  Since the build version 22.1.0 of the support library,  ActionBarActivity is deprecated.  It was the base class of appcompat-v7.  At present, AppCompatActivity is the base class of the support library.  It has come up with many new features like ToolBar, tinted widgets,  material design color pallets etc. </ending> | </end> |
| <hitle> | Android Support Library | <chare> | 1 | <pext> | <starting>Android Support Library</heading> The android platform supports a wide variety of the versions  and devices to choose from.  With the release of every new version,  new Android APIs are added and evolved.  To make these new Android APIs available to users on older devices  the Android Support Library was designed.  Android Support Library provides developers with newer APIs  that are compatible on older framework releases. 1. Compatibility Libraries: These focus on back porting features  so that older frameworks can take advantage of newer releases.  The major libraries include v4 and v7-appcompat. v4 includes classes like DrawerLayout and ViewPager  while appcompat-v7 provides classes for support ActionBar and ToolBar. 2. Component Libraries: These include libraries of certain modules  that don’t depend on other support library dependencies. They can be easily added or removed.  Examples include v7-recyclerview and v7-cardview. 3. Miscellaneous libraries:  The Android support libraries consists of few other libraries such as v8  which provides support for RenderScript, annotations  for supporting annotations like @NonNull. </ending> | </end> |
| <hitle> | onPause() only scenario | <chare> | 1 | <pext> | <starting>onPause() only scenario</heading> Create and launch a new activity which obscures the current activity partially. This can be done by defining the layout\_width and layout\_height  to partially cover the screen.  This would keep the first activity visible but not in the foreground.  Example: define the layout\_width and layout\_height as 200dp each. </ending> | </end> |
| <hitle> | activity- screen rotate | <chare> | 1 | <pext> | <starting>activity- screen rotate</heading> When the screen is rotated, the current instance of the activity is destroyed  a new instance of the Activity is created in the new orientation.  The onRestart() method is invoked first when a screen is rotated.  The other lifecycle methods get invoked in the similar flow  as they were when the activity was first created. Lifecycle methods flow: onPause() onSaveInstanceState() onStop() onDestroy() onCreate() onStart() onRestoreInstanceState() onResume()  How to prevent the data from reloading and resetting when the screen is rotated? The most basic approach is to add an element attribute tag `android:configChanges` inside the activity tag in the AndroidManifest.xml as shown below. <activity android:name=".MainActivity"  android:configChanges="orientation|screenSize">  <intent-filter>  <action android:name="android.intent.action.MAIN" />  <category android:name="android.intent.category.LAUNCHER" />  </intent-filter> </activity>  In general, the configChanges for any activity are defined as android:configChanges="orientation|screenSize|keyboardHidden" The `keyboardHidden` configuration is to prevent the keyboard from resetting if it's pulled out.  `android:configChanges` is not the recommended way by Google. Though it's the simplest way to use, it comes with its own share of drawbacks. First, the common perception that android:configChanges = "orientation" will magically retain the data is a complete misinterpretation. The orientation changes can occur from a number of other events such as changing the default language can trigger a configuration change and destroy and recreate the activity. Second, the activity can restart itself if it's in the background and Android decides to free up its heap memory by killing it. When the application returns to the foreground it'll restart it's data to the original state and the user may not like that. A better alternative of `android:configChanges` is; Saving the current state of the activity when it's being destroyed and restoring the valuable data when it's restarted can be done by overriding the methods `onSaveInstanceState()` and `onRestoreInstanceState()` of the activity class. </ending> | </end> |
| <hitle> | activity lifecycle flow launch, background, foreground, rotate, portrait, landscape | <chare> | 1 | <pext> | <starting>activity lifecycle flow launch, background, foreground, rotate</heading> 1. When the screen is rotated, the current instance of the activity is destroyed  a new instance of the Activity is created in the new orientation.  The onRestart() method is invoked first when a screen is rotated.  The other lifecycle methods get invoked in the similar flow  as they were when the activity was first created. Lifecycle methods flow: onPause() onSaveInstanceState() onStop() onDestroy() onCreate() onStart() onRestoreInstanceState() onResume() 2. When the activity is launched (or show in foreground) onCreate() onStart() onResume() 3. When the app goes into background (shutdown or home button is pressed) onPause() onStop() onDestroy() 4. When the screen changes orientation from portrait to landscape? onStop() OnCreate() onRestoreInstanceState() </ending> | </end> |
| <hitle> | onSaveInstanceState() vs onRestoreInstanceState() | <chare> | 1 | <pext> | <starting>onSaveInstanceState() vs onRestoreInstanceState()</heading> In general the onSaveInstanceState() is invoked  after onPause() and before the onStop().  But the API documentation explicitly states  that the onSaveInstanceState( ) method will be called before onStop()  but makes no guarantees it will be called before or after onPause().  The onRestoreInstanceState() is called after onStart() is invoked.  The onRestoreInstanceState() method is invoked  only when the activity was killed before.  If the activity is NOT killed the onSaveInstanceState() is NOT called.  When the activity is being destroyed, the onSaveInstanceState() gets invoked. The onSaveInstanceState contains a Bundle parameter.  The data to be saved is stored in the bundle object in the form of a HashMap. The bundle object is like a custom HashMap object.  The data is retrieved in the onRestoreInstanceState() method using the keys. </ending> | </end> |
| <hitle> | screen orientation fixed | <chare> | 1 | <pext> | <starting>screen orientation fixed</heading> The screen orientation can be fixed by adding the attribute `android:screenOrientation="portrait"` or `android:screenOrientation="landscape"` in the activity tag.  To keep the screen always on for a particular screen add the `android:keepScreenOn="true"` in the root tag of the activity layout. </ending> | </end> |
| <hitle> | activity restart | <chare> | 1 | <pext> | <starting>activity restart</heading> public class MainActivity extends AppCompatActivity { public void restartActivity() {  MainActivity.this.recreate();  } } </ending> | </end> |
| <hitle> | intent usages | <chare> | 1 | <pext> | <starting>intent usages</heading> Android Intents are used to 1. start an activity - startActivity(intent) 2. start a service - startService(intent) 3. deliver a broadcast - sendBroadcast(intent) </ending> | </end> |
| <hitle> | intent setFlags(), addFlags() | <chare> | 1 | <pext> | <starting>intent setFlags(), addFlags()</heading> When we're using setFlags,  we're replacing the old flags with a new set of Flags.  When we use addFlags, we're appending more flags. </ending> | </end> |
| <hitle> | phone call intent | <chare> | 1 | <pext> | <starting>phone call intent</heading> 1. To enable calling from the application we need to add  a permission in the manifest tag of AndroidManifest.xml <uses-permission android:name="android.permission.CALL\_PHONE" /> In the MainActivity the following code invokes an action call  to the given number represented as a string.  The string is parsed as a URI. String phone\_number = "XXXXXXX" // replace it with the number Intent intent=new Intent(Intent.ACTION\_CALL,Uri.parse("tel:"+phone number); startActivity(intent); 2. To open a URL we need to add a permission. <uses-permission android:name="android.permission.INTERNET" /> The intent to view a URL is defined below. Intent intent = new Intent(Intent.ACTION\_VIEW, Uri.parse("https://www.journaldev.com/")); startActivity(intent); </ending> | </end> |
| <hitle> | setFlags() vs addFlags() | <chare> | 1 | <pext> | <starting>setFlags() vs addFlags()</heading> When we're using setFlags,  we're replacing the old flags with a new set of Flags.  When we use addFlags, we're appending more flags. </ending> | </end> |
| <hitle> | intent- clear back stack activity | <chare> | 1 | <pext> | <starting>intent- clear back stack activity</heading> 1. The first approach is to use a `FLAG\_ACTIVITY\_CLEAR\_TOP` flag. Intent intent= new Intent(ActivityA.this, ActivityB.class); intent.setFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_TOP); startActivity(intent); finish(); 2.The second way is by using `FLAG\_ACTIVITY\_CLEAR\_TASK`  and `FLAG\_ACTIVITY\_NEW\_TASK` in conjunction. Intent intent= new Intent(ActivityA.this, ActivityB.class); intent.setFlags(Intent.FLAG\_ACTIVITY\_NEW\_TASK | Intent.FLAG\_ACTIVITY\_CLEAR\_TASK); startActivity(intent); </ending> | </end> |
| <hitle> | FLAG\_ACTIVITY\_CLEAR\_TASK vs FLAG\_ACTIVITY\_CLEAR\_TOP | <chare> | 1 | <pext> | <starting>FLAG\_ACTIVITY\_CLEAR\_TASK vs FLAG\_ACTIVITY\_CLEAR\_TOP</heading> `FLAG\_ACTIVITY\_CLEAR\_TASK` is used to clear all the activities from the task  including any existing instances of the class invoked.  The Activity launched by intent  becomes the new root of the otherwise empty task list.  This flag has to be used in conjunction with `FLAG\_ ACTIVITY\_NEW\_TASK`.  `FLAG\_ACTIVITY\_CLEAR\_TOP` on the other hand,  if set and if an old instance of this Activity exists in the task list then barring  that all the other activities are removed  and that old activity becomes the root of the task list.  Else if there's no instance of that activity  then a new instance of it is made the root of the task list.  Using `FLAG\_ACTIVITY\_NEW\_TASK` in conjunction is a good practice,  though not necessary. </ending> | </end> |
| <hitle> | FLAG\_ACTIVITY\_NEW\_TASK | <chare> | 1 | <pext> | <starting>FLAG\_ACTIVITY\_NEW\_TASK</heading> When we're trying to launch an activity from outside the activity's context,  a FLAG\\_ACTIVITY\\_NEW\\_TASK is compulsory  else a runtime exception would be thrown.  Example scenarios are: launching from a service,  invoking an activity from a notification click.  If the activity instance is already on the task list when the flag is set,  it will invoke the onNewIntent() method of that Activity.  All the implementation stuff goes in that method. </ending> | </end> |
| <hitle> | android:launchMode | <chare> | 1 | <pext> | <starting>android:launchMode</heading> - \*\*standard\*\* : It's the default launch mode for an activity  wherein every new instance of the activity called  will be put on top of the stack as a separate entity.  Hence calling startActivity() for a particular class 10 times  will create 10 activities in the task list. - \*\*singleTop\*\*: It differs from the standard launch mode  in the fact that when the Activity instance that's invoked is already  present on the top of the stack, instead of creating a new Activity,  that instance will be called.  In cases where the same Activity instance is not on the top of the stack  or if it doesn't exist in the stack at all then a new instance of the activity  will be added to the stack.  Hence we need to handle the upcoming intent in both  the `onCreate()` and `onNewIntent()` methods to cover all cases. - \*\*singleTask\*\*: This is different from singleTop  in the case that if the Activity instance is present in the stack,  the onNewIntent() would be invoked and  that instance would be moved to the top of the stack.  All the activities placed above the singleTask instance  would be destroyed in this case.  When the activity instance does not exist in the stack,  the new instance would be placed on the top of the stack  similar to the standard mode. - \*\*singleInstance\*\* : An activity with this launchMode defined  would place only a singleton activity instance in the Task.  The other activities of the application will be placed in a separate Task. </ending> | </end> |
| <hitle> | taskAffinity | <chare> | 1 | <pext> | <starting>taskAffinity</heading> A taskAffinity is an attribute tag defined in the activity tag  in the AndroidManifest.xml for launchMode singleInstance.  Activities with similar taskAffinity values are grouped together in one task. </ending> | </end> |
| <hitle> | LinearLayout, RelativeLayout, AbsoluteLayout | <chare> | 1 | <pext> | <starting>LinearLayout, RelativeLayout, AbsoluteLayout</heading> LinearLayout arranges its children  in a single row or single column one after the other.  RelativeLayout arranges it's children in positions relative to each other  or relative to parent depending upon the LayoutParams defined for each view. AbsoluteLayout needs the exact positions  of the x and y coordinates of the view to position it. Though this is deprecated now. </ending> | </end> |
| <hitle> | FrameLayout vs TableLayout | <chare> | 1 | <pext> | <starting>FrameLayout vs TableLayout</heading> A FrameLayout stack up child views above each other  with the last view added on the top.  Though we can control the position of the children inside the FrameLayout  using the layout\_gravity attribute.  When the width and height of the FrameLayout are set to wrap\_content,  the size of the FrameLayout equals the size of the largest child (plus padding). A TableLayout consists of TableRows.  The children are arranged in the form of rows and columns. </ending> | </end> |
| <hitle> | ConstraintLayout | <chare> | 1 | <pext> | <starting>ConstraintLayout</heading> A ConstraintLayout is a ViewGroup which allows you to position  and size widgets in a flexible way. By ConstraintLayout we can do various types of constraints like: 1. Relative positioning 2. Margins 3. Centering positioning 4. Circular positioning 5. Visibility behavior 5. Dimension constraints 7. Chains 8. Virtual Helpers objects 9. Optimizer You cannot have a circular dependency in constraints. </ending> | </end> |
| <hitle> | commit() vs apply() in Shared Preferences | <chare> | 1 | <pext> | <starting>commit() vs apply() in Shared Preferences</heading> Data is stored in SharedPreferences in the form of a key-value pair(HashMap). commit() was introduced in API 1 whereas apply() came up with API 9. commit() writes the data synchronously  and returns a boolean value of success or failure  depending on the result immediately. apply() is asynchronous and it won't return any boolean response.  Also, if there is an apply() outstanding and we perform another commit(),  then the commit() will be blocked until the apply() is not completed. commit() is instantaneous and performs disk writes.  If we're on the main UI thread apply() should be used since it's asynchronous. </ending> | </end> |
| <hitle> | StateListDrawable | <chare> | 1 | <pext> | <starting>StateListDrawable</heading> A StateListDrawable is a drawable object defined in the XML  that allows us to show a different color/background  for a view for different states.  Essentially it's used for Buttons to show a different look for each state (pressed, focused, selected, none). </ending> | </end> |
| <hitle> | kill Activity | <chare> | 1 | <pext> | <starting>kill Activity</heading> Let's say, We'll declare and assign a class instance of the FirstActivity to itself. public class FirstActivity extends AppCompatActivity { public static FirstActivity firstActivity;  @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);  firstActivity=this;  } } We'll call finish() on the instance of the FirstActivity to kill the activity from any other activity. FirstActivity.firstActivity.finish() </ending> | </end> |
| <hitle> | AsyncTask vs Thread | <chare> | 1 | <pext> | <starting>AsyncTask vs Threade</heading> 1. A Thread is generally used for long tasks to be run in the background.  We need a Handler class to use a Thread.  1-1. An AsyncTask is an intelligent Thread subclass.  It's recommended to use AsyncTask when the caller class is the UI Thread  as there is no need to manipulate the handlers.  2. AsyncTask is generally used for small tasks that can communicate back  with the main UI thread using the two methods  onPreExecute() and onPostExecute() it has.  2-1. A Handler class is preferred when we need to perform a background task repeatedly after every x seconds/minutes. </ending> | </end> |
| <hitle> | doInBackground | <chare> | 1 | <pext> | <starting>doInBackground</heading> The returned value of the doInBackground  goes to the onPostExecute() method.  We can update the main UI thread from here.  To get the returned value in the onCreate() method  we need to use the following code snippet. ``` MyTask myTask= new MyTask(); String result=myTask.execute().get(); ``` This approach is not recommended as it blocks the main UI thread  until the value is not returned.  The ideal scenario to use it is  when the other views of the UI thread need the value from the AsyncTask for processing. </ending> | </end> |
| <hitle> | service | <chare> | 1 | <pext> | <starting>service</heading> A service is a component in android that's used for performing tasks in the background such as playing Music, location updating etc. Unlike activities, a service does not have a UI. Also, a service can keep running in the background even if the activity is destroyed. </ending> | </end> |
| <hitle> | service- start/stop | <chare> | 1 | <pext> | <starting>service- start/stop</heading> A service is started by ``` startService(new Intent(this, MyService.class)); ``` We need to register the service first in the AndroidManifest.xml file. ``` <service android:name="MyService"/> ``` To stop a service we execute `stopService()`. To stop the service from itself we call `stopSelf()`. </ending> | </end> |
| <hitle> | Bound vs Unbound Service | <chare> | 1 | <pext> | <starting>Bound vs Unbound Service</heading> Services are largely divided into two categories : \*\*Bound Services\*\* and \*\*Unbound/Started Services\*\* 1. \*\*Bound Services\*\*: An Android component may bind itself  to a Service using `bindservice()`.  A bound service would run as long as the other application components  are bound to it.  As soon as the components call `unbindService()`, the service destroys itself. 2. \*\*Unbound Services\*\*: A service is started when a component (like activity)  calls startService() method and it runs in the background indefinitely  even if the original component is destroyed. </ending> | </end> |
| <hitle> | lifecycle of service | <chare> | 1 | <pext> | <starting>lifecycle of service</heading> - `onStartCommand()` : This method is called when startService() is invoked. Once this method executes, the service is started  and can run in the background indefinitely.  This method is not needed if the service is defined as a bounded service.  The service will run indefinitely in the background when this method is defined.  We'll have a stop the service ourselves - `onBind()` This method needs to be overridden when the service is defined  as a bounded service.  This method gets called when bindService() is invoked.  In this method, we must provide an interface  that clients use to communicate with the service, by returning an IBinder. We should always implement this method,  but if you don’t want to allow binding, then you should return null. - `onCreate()` : This method is called while the service is first created.  Here all the service initialization is done - `onDestroy()` : The system calls this method  when the service is no longer used and is being destroyed.  All the resources, receivers, listeners clean up are done here </ending> | </end> |
| <hitle> | Broadcast Receivers vs Services | <chare> | 1 | <pext> | <starting>Broadcast Receivers vs Services</heading> A service is used for long running tasks in the background such as  playing a music or tracking and updating the user's background location.  A Broadcast Receiver is a component that once registered within an application  executes the onReceive() method when some system event gets triggered.  The events the receiver listens to  are defined in the AndroidManifest.xml in the intent filters.  0Types of system events that a Broadcast Receiver listens to are: changes in the network, boot completed, battery low, push notifications received etc.  We can even send our own custom broadcasts using `sendBroadcast(intent)`. </ending> | </end> |
| <hitle> | RecyclerView | <chare> | 1 | <pext> | <starting>RecyclerView</heading> RecyclerView is a ViewGroup added to the Android Studio  as a successor of the GridView and ListView.  It is an improvement on both of them.  It has been created to make possible construction of any lists  with XML layouts as an item that can be customized vastly  while improving the efficiency of ListViews and GridViews.  This improvement is achieved by recycling the views  which are out of the visibility of the user.  For example, if a user scrolled down to a position  where items 4 and 5 are visible;  items 1, 2, and 3 would be cleared from the memory  to reduce memory consumption.  </ending> | </end> |
| <hitle> | RecyclerView vs ListView | <chare> | 1 | <pext> | <starting>RecyclerView vs ListView</heading> - A RecyclerView recycles and reuses cells when scrolling. This is a default behaviour.  It's possible to implement the same in a ListView too  but we need to implement a ViewHolder there. - A RecyclerView decouples list from its container  so we can put list items easily at run time in the different containers  (linearLayout, gridLayout) by setting LayoutManager. - Animations of RecyclerView items are decoupled  and delegated to `ItemAnimator` </ending> | </end> |
| <hitle> | AsyncTask in orientation change | <chare> | 1 | <pext> | <starting>AsyncTask in orientation change</heading> The lifecycle of an AsyncTask is not tied onto the Activity  since it's occurring on a background thread.  Hence an orientation change won't stop the AsyncTask.  But if the AsyncTask tries to update the UI thread  after the orientation is changed, it would give rise to `java.lang.IllegalArgumentException: View not attached to window manager`  since it will try to update the former instances of the activity that got reset. </ending> | </end> |
| <hitle> | construction arguments Fragment | <chare> | 1 | <pext> | <starting>construction arguments Fragment</heading> Construction arguments for a Fragment are passed via Bundle  using the Fragment#setArgument(Bundle) 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. </ending> | </end> |
| <hitle> | Service vs IntentService | <chare> | 1 | <pext> | <starting>Service vs IntentService</heading> 1. 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. 2. 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. </ending> | </end> |
| <hitle> | fragment vs activity | <chare> | 1 | <pext> | <starting>fragment vs activity</heading> 1. Activity is an application component that gives a user interface  where the user can interact.  1-1.The fragment is only part of an activity,  it basically contributes its UI to that activity. 2. Activity is not dependent on fragment. 2-1. Fragment is dependent on activity. It can’t exist independently. 3. we need to mention all activity it in the manifest.xml file. 3-1. Fragment is not required to mention in the manifest file 4. We can’t create multi-screen UI without using fragment in an activity. 4-1. After using multiple fragments in a single activity,  we can create a multi-screen UI. 5. Activity can exist without a Fragment. 5-1. Fragment cannot be used without an Activity. 6. Creating a project using only Activity then it’s difficult to manage.  6-1. While Using fragments in the project,  the project structure will be good and we can handle it easily. 7. Lifecycle methods are hosted by the OS. The activity has its own life cycle.  7-1. Lifecycle methods in fragments are hosted by hosting the activity. 8. Activity is not light weight.  8-1. The fragment is the lite weight. </ending> | </end> |
| <hitle> | AsyncTask, Activity lifecycle | <chare> | 1 | <pext> | <starting>AsyncTask, Activity lifecycle</heading> An AsyncTask is not tied to the life cycle of the Activity that contains it.  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. Then, 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) 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. </ending> | </end> |
| <hitle> | Gradle | <chare> | 1 | <pext> | <starting>Gradle</heading> Gradle is a build system (open source) that is used to automate building,  testing, deployment, etc.  “Build.gradle” are scripts where one can automate the tasks.  For example, the simple task to copy some files from one directory  to another can be performed by Gradle build script  before the actual build process happens.  Usage: Every Android project needs a Gradle for generating an apk  from the .java and .xml files in the project.  Simply put, a Gradle takes all the source files (java and XML)  and applies appropriate tools, e.g., converts the java files into dex files  and compresses all of them into a single file known as apk that is actually used. </ending> | </end> |
| <hitle> | Intent filters vs Intent | <chare> | 1 | <pext> | <starting>Intent filters vs Intent</heading> An Intent is an object passed to  Context.startActivity(), Context.startService() or  Activity.startActivityForResult() etc.  to launch an activity or get an existing activity to do something new.  On the other hand, an Intent filter describes the capability of the component (like activities, services, and broadcast receivers).  </ending> | </end> |
| <hitle> | setContentView() | <chare> | 1 | <pext> | <starting>setContentView()</heading> Activity life cycle onCreate() method is called only once.  And this is the big reason we need to call the setContentView() in onCreate(). And it will be inefficient to call this function in onResume(), onStart(),  and somewhere else because those methods are called more than once.  </ending> | </end> |
| <hitle> | View | <chare> | 1 | <pext> | <starting>View</heading> The view is a class that represents the basic building block for UI components. A View occupies a rectangular area on the screen  and is responsible for drawing and event handling.  It is a superclass for all the UI components.  The most common UI components are: TextView EditText ImageView Button ProgressBar CheckBox, etc. </ending> | </end> |
| <hitle> | View vs ViewGroup | <chare> | 1 | <pext> | <starting>View vs ViewGroup</heading> 1. View is a simple rectangle box that responds to the user’s actions. 1-1. ViewGroup is the invisible container. It holds View and ViewGroup 2. View is the SuperClass of All component like TextView, EditText, ListView, etc. 2-1. ViewGroup is a collection of Views(TextView, EditText, ListView, etc..),  somewhat like a container. 3. A View object is a component of the user interface (UI)  like a button or a text box, and it’s also called a widget. 3-1. A ViewGroup object is a layout,  that is, a container of other ViewGroup objects (layouts)  and View objects (widgets) 4. Examples are EditText, Button, CheckBox, etc. 4-1. For example, LinearLayout is the ViewGroup that contains Button(View),  and other Layouts also. 5. View refers to the android.view.View class. 5-1 ViewGroup refers to the android.view.ViewGroup class 6. android.view.View which is the base class of all UI classes. 6-1. ViewGroup is the base class for Layouts. </ending> | </end> |
| <hitle> | coroutine | <chare> | 1 | <pext> | <starting>coroutine</heading> 1. A coroutine is an instance of suspendable computation.  It is conceptually similar to a thread, in the sense that  it takes a block of code to run that works concurrently  with the rest of the code.  However, a coroutine is not bound to any particular thread.  It may suspend its execution in one thread and resume in another one.  2. The Kotlin team defines coroutines as “lightweight threads”.  They are sort of tasks that the actual threads can execute.  Kotlin coroutines introduce a new style of concurrency  that can be used on Android to simplify async code.  The official documentation says that coroutines are lightweight threads.  By lightweight, it means that creating coroutines doesn’t allocate new threads. Instead, they use predefined thread pools and smart scheduling  for the purpose of which task to execute next and which tasks later. </ending> | </end> |
| <hitle> | Garbage Collector | <chare> | 1 | <pext> | <starting>Garbage Collector</heading> Garbage Collector in Android has no compacting.  This means the address of objects in the heap never changed  after their creation.  So garbage collection can be triggered when an allocation fails when an 1. OutOfMemoryError is about to be triggered, 2. When the size of the heap hits some soft limit, and 3. When a GC was explicitly requested. </ending> | </end> |
| <hitle> | Reduce APK size | <chare> | 1 | <pext> | <starting>Reduce APK size</heading> 1. Remove unused sources 2. Use of Vector Drawables 3. Reuse your code 4. Compress PNG and JPEG files 5. Use of Lint 6. Use images in WebP file format 7. Use of proguard 8. Use of ShrinkResources 9. Limit the usage of external libraries 10. Use the Android Size Analyzer tool 11. Generate App Bundles instead of APK 12. Use of Resconfigs </ending> | </end> |
| <hitle> | prevent ANR Application Not Responding | <chare> | 1 | <pext> | <starting>prevent ANR Application Not Responding</heading> ANR stands for Application Not Responding.  An ANR will occur if you’re running a process on the UI thread  which takes an extended time, usually around 5 seconds.  During this point, the GUI (Graphical User Interface) will lock up  which can end in anything the user presses won’t be actioned.  After the 5 seconds approx. has occurred, if the thread still hasn’t recovered  then an ANR dialogue box is shown informing the user  that the appliance isn’t responding and can give the user the choice  to either wait, in the hope that the app will eventually recover,  or to force close the app. Stop doing heavy tasks on the main thread.  Instead, use worker threads such as IntentService, AsyncTask Handler,  or another Thread simply.  Detecting where ANRs happen is straightforward if it’s a permanent block  (deadlock acquiring some locks for instance),  but harder if it’s just a short-lived delay.  First, re-evaluate your code and appearance for vulnerable spots  and long-running operations. </ending> | </end> |
| <hitle> | KTX | <chare> | 1 | <pext> | <starting>KTX</heading> KTX library is the only one among the foundation components  which was introduced for the first time with the release of the Jetpack. Android KTX is a collection of Kotlin extensions  that are designed to facilitate developers to remove boilerplate code  as well as to write concise code while developing android applications  with Kotlin language.  Here KTX in the name stands for Kotlin Extensions.  ++++++++++++++++++++++++++++++++++++++++++++++ Below is an example of a piece of code without using and after using the Android KTX library: Code snippet of SQLite without using KTX library: db.beginTransaction() try { // insert data db.setTransactionSuccessful() }  finally { db.endTransaction() } Above code after using KTX library: db.transaction {  // insert data } </ending> | </end> |
| <hitle> | Dagger | <chare> | 1 | <pext> | <starting>Dagger</heading> Dagger is a Dependency Injection framework  that will generate a lot of boilerplate code  for you to achieve the goal of Dependency injection in Android development. </ending> | </end> |
| <hitle> | Junit | <chare> | 1 | <pext> | <starting>JUnit</heading> JUnit is a “Unit Testing” framework for Java Applications  which is already included by default in android studio.  It is an automation framework for Unit as well as UI Testing.  It contains annotations such as @Test, @Before, @After, etc.  </ending> | </end> |
| <hitle> | AIDL | <chare> | 1 | <pext> | <starting>AIDL</heading> On Android what happens with the processors is one process  can’t normally access the memory of another process.  So in order to interact they need to decompose their objects  into primitives that the OS can understand and gather the objects  across that boundary.  The code to do that gathering is very complex to write,  so Android handles it with AIDL.  So generally AIDL is similar to the other IDLs and it allows  to define the programming interface  that both the client and service agree upon  in order to interact with each other using interprocess communication (IPC). </ending> | </end> |
| <hitle> | RecyclerView improve | <chare> | 1 | <pext> | <starting>RecyclerView improve</heading> 1. Set a specific width and height to ImageView in RecyclerView items 2. Avoid using NestedView 3. Use the setHasFixedsize method 4. Use the image loading library for loading images 5. Do less work in the OnBindViewHolder method 6. Use the NotifyItem method for your RecyclerView </ending> | </end> |
| <hitle> | Architecture Components | <chare> | 1 | <pext> | <starting>Architecture Components</heading> 1. Room 2. WorkManager 3. Lifecycle 4. ViewModel 5. LiveData 6. Navigation 7. Paging 8. Data Binding </ending> | </end> |
| <hitle> | activityCreator | <chare> | 1 | <pext> | <starting>activityCreator</heading> An activityCreator is the first step  towards the creation of a new Android project.  It is made up of a shell script that will be used  to create new file system structure necessary for writing codes  within the Android IDE. </ending> | </end> |
| <hitle> | Adb | <chare> | 1 | <pext> | <starting>Adb</heading> Adb is short for Android Debug Bridge.  It allows developers the power to execute remote shell commands.  Its basic function is to allow and control communication  towards and from the emulator port. </ending> | </end> |
| <hitle> | 4 essential states of activity | <chare> | 1 | <pext> | <starting>4 essential states of activity</heading> 1. Active – if the activity is at the foreground 2. Paused – if the activity is at the background and still visible 3. Stopped – if the activity is not visible and therefore is hidden  or obscured by another activity 4. Destroyed – when the activity process is killed or completed terminated </ending> | </end> |
| <hitle> | Escape characters android | <chare> | 1 | <pext> | <starting>Escape characters android</heading> Escape characters are preceded by double backslashes.  For example, a newline character is created using ‘\\n’ </ending> | </end> |
| <hitle> | Permissions android | <chare> | 1 | <pext> | <starting>Permissions android</heading> Permissions allow certain restrictions to be imposed primarily  to protect data and code. Without these, codes could be compromised,  resulting to defects in functionality. </ending> | </end> |
| <hitle> | intent filter | <chare> | 1 | <pext> | <starting>intent filter</heading> Because every component needs to indicate which intents they can respond to,  intent filters are used to filter out intents  that these components are willing to receive.  One or more intent filters are possible,  depending on the services and activities that is going to make use of it. </ending> | </end> |
| <hitle> | activity lifetime | <chare> | 1 | <pext> | <starting>activity lifetime</heading> 1. Entire lifetime – activity happens between onCreate and onDestroy 2. Visible lifetime – activity happens between onStart and onStop 3. Foreground lifetime – activity happens between onResume and onPause </ending> | </end> |
| <hitle> | activity process states | <chare> | 1 | <pext> | <starting>activity process states</heading> 1. foreground activity 2. visible activity 3. background activity 4. empty process </ending> | </end> |
| <hitle> | service by AIDL | <chare> | 1 | <pext> | <starting>service by AIDL </heading> 1. create the .aidl file, which defines the programming interface 2. implement the interface, which involves  extending the inner abstract Stub class as well as implanting its methods. 3. expose the interface, which involves implementing the service to the clients. </ending> | </end> |
| <hitle> | AIDL data types | <chare> | 1 | <pext> | <starting>AIDL data types</heading> -string -charSequence -List -Map -all native Java data types like int,long, char and Boolean </ending> | </end> |
| <hitle> | visible activity | <chare> | 1 | <pext> | <starting>visible activity</heading> A visible activity is one that sits behind a foreground dialog.  It is actually visible to the user, but not necessarily being in the foreground itself. </ending> | </end> |
| <hitle> | kill foreground activity | <chare> | 1 | <pext> | <starting>kill foreground activity</heading> The foreground activity, being the most important among the other states,  is only killed or terminated as a last resort,  especially if it is already consuming too much memory.  When a memory paging state has been reach by a foreground activity,  then it is killed so that the user interface can retain its responsiveness  to the user. </ending> | </end> |
| <hitle> | Sticky Intent | <chare> | 1 | <pext> | <starting>Sticky Intent</heading> A Sticky Intent is a broadcast from sendStickyBroadcast() method  such that the intent floats around even after the broadcast,  allowing others to collect data from it. </ending> | </end> |
| <hitle> | action android | <chare> | 1 | <pext> | <starting>action android</heading> In Android development, an action is what the intent sender wants to do  or expected to get as a response.  Most application functionality is based on the intended action. </ending> | </end> |
| <hitle> | nine-patch image | <chare> | 1 | <pext> | <starting>nine-patch image</heading> In general, a Nine-patch image allows resizing  that can be used as background or other image size requirements  for the target device.  The Nine-patch refers to the way you can resize the image:  4 corners that are unscaled, 4 edges that are scaled in 1 axis,  and the middle one that can be scaled into both axes. </ending> | </end> |
| <hitle> | states in activity | <chare> | 1 | <pext> | <starting>states in activity</heading> 1. Active state: The activity is in the foreground 2. Paused state: The activity is in the background and visible 3. Stopped state: The activity is in the background but not visible,  even hidden or obscuring other activities 4. Destroyed state: The activity is completely terminated or killed/removed </ending> | </end> |
| <hitle> | critical loops activity | <chare> | 1 | <pext> | <starting>critical loops activity</heading> Loop 1, Entire Lifetime: The activity happens between onCreate and onDestroy. Loop 2, Visible Lifetime: The activity happens between onStart and onStop Loop 3, Foreground Lifetime:  The activity happens between onResume and onPause </ending> | </end> |
| <hitle> | dependency injection | <chare> | 1 | <pext> | <starting>dependency injection</heading> Dependency injection is a technique  used to remove hard-coded dependencies between objects,  making it easier to change them later.  This is useful in situations where you want to be able to swap out  one implementation of a dependency for another,  without having to change the code that uses it. </ending> | </end> |
| <hitle> | PendingIntent | <chare> | 1 | <pext> | <starting>PendingIntent</heading> PendingIntent is a high-level API that lets you create and manage  asynchronous requests to other apps.  You can use this class for starting activities, delivering results,  and receiving results from other apps. PendingIntent starts at some point in the future but Intent starts immediately. Pending Intent is frequently used in: 1. NotificationiManager 2. AlarmManager 3. AppWidgetManager </ending> | </end> |
| <hitle> | Flavors in Android | <chare> | 1 | <pext> | <starting>Flavors in Android</heading> Customizing an app across different code bases  can be tedious and error-prone.  Therefore, using product flavors is ideal since you can create app variants with a single code base.  You should also mention that Gradle makes it easy to create build-type variants. </ending> | </end> |
| <hitle> | Unit tests | <chare> | 1 | <pext> | <starting>Unit tests</heading> Unit tests are tests that focus on a single component   or small group of components in an Android app.  These tests help to ensure that the code for these components  is working as expected. Android unit tests are typically written in Java. Unit tests help to ensure  that the individual components of an Android application  are functioning correctly.  By writing unit tests, developers can catch bugs early on and prevent them  from becoming bigger issues later on.  Additionally, unit tests can help to improve  the overall quality of an Android application  by helping to ensure that all of its components are working correctly. </ending> | </end> |
| <hitle> | Robolectric vs Mockito | <chare> | 1 | <pext> | <starting>Robolectric vs Mockito</heading> Mockito is a library that allows you to create mock objects for testing. Robolectric is a library that allows you to run Android unit tests on a JVM  instead of on a device or emulator. </ending> | </end> |
| <hitle> | unit testing tools | <chare> | 1 | <pext> | <starting>unit testing tools</heading> There are a few different tools that can be used for unit testing in Android,  but some of the most popular ones include JUnit, Robolectric, and Espresso. Each of these tools has its own strengths and weaknesses,  so it’s important to choose the one that’s right for your particular project. </ending> | </end> |
| <hitle> | mock object | <chare> | 1 | <pext> | <starting>mock object</heading> A mock object is a test object that simulates the behavior of a real object.  In Android development, mock objects can be used  to simulate the behavior of a real Android object,  such as an Activity or a Fragment.  This allows you to test the behavior of your code  without having to actually launch the Android object. </ending> | </end> |
| <hitle> | coverage with unit tests | <chare> | 1 | <pext> | <starting>coverage with unit tests</heading> Coverage is a measure of how much of your code  is being executed by your unit tests.  It is important to achieve high coverage  because it means that your tests are exercising a large portion of your code,  and therefore are more likely to find bugs.  It also means that your tests are more likely  to be effective in preventing regressions. 1. Statement coverage: This ensures  that each statement/line in the source code is executed and tested. 2. Decision coverage: This ensures  that every decision point that results in true or false is executed and run. 3. Path coverage: This ensures  that every possible route from a given point is run and tested. </ending> | </end> |
| <hitle> | Test Driven Development (TDD) | <chare> | 1 | <pext> | <starting>Test Driven Development (TDD)</heading> There is no one-size-fits-all answer to this question,  as the best way to implement TDD in Android  will vary depending on the specific project and team. how to implement TDD? – Planning and designing tests before writing any code – Writing tests that are small and focused – Automating as much of the testing process as possible – Using a continuous integration server to run tests automatically – Regularly reviewing and refactoring tests </ending> | </end> |
| <hitle> | unit testing cons | <chare> | 1 | <pext> | <starting>unit testing cons</heading> One of the main limitations of unit testing in Android  is the lack of a true emulator environment.  This means that unit tests can only be run on actual devices,  which can be time-consuming and expensive.  Additionally, the Android platform is constantly changing,  which can make it difficult to keep unit tests up-to-date. </ending> | </end> |
| <hitle> | AndroidTestCase | <chare> | 1 | <pext> | <starting>AndroidTestCase</heading> AndroidTestCase is a class that provides a framework  for testing Android applications.  It is a subclass of junit.framework.TestCase, and provides functionality  to set up and tear down an Android test environment,  as well as access to various Android system services. </ending> | </end> |
| <hitle> | ActivityInstrumentationTestCase2 | <chare> | 1 | <pext> | <starting>ActivityInstrumentationTestCase2</heading> ActivityInstrumentationTestCase2 is a class  that provides functional testing of a single activity.  You can use this class to test a single activity in isolation  from the rest of the system. </ending> | </end> |
| <hitle> | unit tests best practice | <chare> | 1 | <pext> | <starting>unit tests best practice</heading> Some best practices that I follow when writing unit tests in Android  are to keep my tests short and focused,  to write tests for all public methods in my classes,  and to make sure that my tests cover all possible scenarios.  Additionally, I always run my tests on a simulator or emulator  before running them on a real device  to ensure that they will work as expected. </ending> | </end> |
| <hitle> | Assert | <chare> | 1 | <pext> | <starting>Assert</heading> Assert methods are utility methods that support assert conditions in test cases. They belong to the Assert class in JUnit 4 and the Assertions class in JUnit 5. It is recommended to import the assert methods statically to the test class  for avoiding using the class as a prefix to the method.  </ending> | </end> |
| <hitle> | TestNG vs Junit | <chare> | 1 | <pext> | <starting>TestNG vs Junit</heading> 1. Open-source unit testing framework for writing test cases. 1-1. TestNG is similar to JUnit but with extended functionalities. 2. JUnit does not support advanced annotations. 2-1. TestNG supports advanced and special annotations too. 3. JUnit does not support parallel testing. 3-1. TestNG supports multiple threads to run parallelly. 4. JUnit, Group test is not supported. 4-1. TestNG, Group test is supported in TestNG. 5. JUnit, The naming convention for annotations  like Before, After, Expected are confusing. 5-1. TestNG, The naming convention for understanding annotations  like BeforeMethod, AfterMethod, ExpectedException  is easy to understand based on their functionality. 6. JUnit, Cannot rerun failed cases. 6-1. TestNG, Can rerun failed tests. </ending> | </end> |
| <hitle> | Android - Application Components | <chare> | 1 | <pext> | <starting>Android - Application Components</heading> Application components are the essential building blocks  of an Android application.  These components are loosely coupled by the application manifest file  AndroidManifest.xml that describes each component of the application  and how they interact. Application components are: 1. Activities They dictate the UI and handle the user interaction to the smart phone screen. 2. Services They handle background processing associated with an application. 3. Broadcast Receivers They handle communication between Android OS and applications. 4.Content Providers They handle data and database management issues. 5. Intents Intent is a messaging object that is used to request an action from other components of an application. </ending> | </end> |
| <hitle> | characteristics of activity | <chare> | 1 | <pext> | <starting>characteristics of activity</heading> 1. Activity is always associated with UI. 2. No two UI can have the same Activity or vice-versa. 3. Activity has its own life cycle. </ending> | </end> |
| <hitle> | purpose, structure of manifest file | <chare> | 1 | <pext> | <starting>purpose, structure of manifest file</heading> The purpose is to provide essential information  about your app to the Android system,  which the system must have before it can run any of app's code. </ending> | </end> |
| <hitle> | View Binding | <chare> | 1 | <pext> | <starting>View Binding</heading> View binding is a feature that allows you to more easily write  code that interacts with views.  Once view binding is enabled in a module, it generates a binding class  for each XML layout file present in that module. An instance of a binding class contains direct references to all views  that have an ID in the corresponding layout. </ending> | </end> |
| <hitle> | Application Framework | <chare> | 1 | <pext> | <starting>Application Framework</heading> 1. Activity Manager − Controls all aspects of the application lifecycle  and activity stack. 2. Content Providers − Allows applications to publish and share data  with other applications. 3. Resource Manager − Provides access to non-code embedded resources  such as strings, color settings and user interface layouts. 4. Notifications Manager − Allows applications to display alerts  and notifications to the user. 5. View System − An extensible set of views used to create  application user interfaces. </ending> | </end> |
| <hitle> | Android Strings File | <chare> | 1 | <pext> | <starting>Android Strings File</heading> The strings.xml file is located in the res/values folder and it contains  all the text that your application uses.  For example, the names of buttons, labels, default text,  and similar types of strings go into this file.  This file is responsible for their textual content. </ending> | </end> |
| <hitle> | Android Layout File | <chare> | 1 | <pext> | <starting>Android Layout File</heading> The activity\_main.xml is a layout file available in res/layout directory,  that is referenced by your application when building its interface.  You will modify this file very frequently to change the layout  of your application.  </ending> | </end> |
| <hitle> | Accessing Resources in Code | <chare> | 1 | <pext> | <starting>Accessing Resources in Code</heading> When your Android application is compiled, a R class gets generated,  which contains resource IDs for all the resources available  in your res/ directory.  You can use R class to access that resource using sub-directory  and resource name or directly resource ID. Example: To access res/drawable/myimage.png and set an ImageView? ImageView imageView = (ImageView) findViewById(R.id.myimageview); imageView.setImageResource(R.drawable.myimage); Example: When res/values/strings.xml is like this − <?xml version="1.0" encoding="utf-8"?> <resources>  <string name="hello">Hello, World!</string> </resources> Now you can set the text on a TextView object with ID msg using a resource ID ? TextView msgTextView = (TextView) findViewById(R.id.msg); msgTextView.setText(R.string.hello); Example When a layout res/layout/activity\_main.xml is like this− <?xml version="1.0" encoding="utf-8"?> <LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"  android:layout\_width="fill\_parent"   android:layout\_height="fill\_parent"   android:orientation="vertical" >  <TextView android:id="@+id/text"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="Hello, I am a TextView" />  <Button android:id="@+id/button"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="Hello, I am a Button" /> </LinearLayout> setContentView(R.layout.activity\_main) will load this layout for an Activity, in the onCreate() method? public void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main); } </ending> | </end> |
| <hitle> | Accessing Resources in XML | <chare> | 1 | <pext> | <starting>Accessing Resources in XML</heading> When we have a resource file like res/values/strings.xml file  that includes a color resource and a string resource? − <?xml version="1.0" encoding="utf-8"?> <resources>  <color name="opaque\_red">#f00</color>  <string name="hello">Hello!</string> </resources> Now you can use these resources in the layout file  to set the text color and text string. − <?xml version="1.0" encoding="utf-8"?> <EditText xmlns:android="http://schemas.android.com/apk/res/android"  android:layout\_width="fill\_parent"  android:layout\_height="fill\_parent"  android:textColor="@color/opaque\_red"  android:text="@string/hello" /> </ending> | </end> |
| <hitle> | activity load UI by xml layout | <chare> | 1 | <pext> | <starting>activity load UI by xml layout</heading> An activity class loads all the UI component using the XML file available  in res/layout folder of the project.  To load from res/layout/activity\_main.xml file: setContentView(R.layout.activity\_main); </ending> | </end> |
| <hitle> | activity MAIN action, LAUNCHER category | <chare> | 1 | <pext> | <starting>activity MAIN action, LAUNCHER category</heading> An application can have one or more activities without any restrictions.  Every activity you define for your application must be declared  in your AndroidManifest.xml file and the main activity for your app  must be declared in the manifest with an <intent-filter>  that includes the MAIN action and LAUNCHER category: <?xml version="1.0" encoding="utf-8"?> <manifest xmlns:android="http://schemas.android.com/apk/res/android"  package="com.example.tutorialspoint7.myapplication">  <application  android:allowBackup="true"  android:icon="@mipmap/ic\_launcher"  android:label="@string/app\_name"  android:supportsRtl="true"  android:theme="@style/AppTheme">  <activity android:name=".MainActivity">  <intent-filter>  <action android:name="android.intent.action.MAIN" />  <category android:name="android.intent.category.LAUNCHER" />  </intent-filter>  </activity>  </application> </manifest> </ending> | </end> |
| <hitle> | service state | <chare> | 1 | <pext> | <starting>service state</heading> 1. Started A service is started when an application component, such as an activity,  starts it by calling startService().  Once started, a service can run in the background indefinitely,  even if the component that started it is destroyed. 2. Bound A service is bound when an application component binds to it  by calling bindService().  A bound service offers a client-server interface that allows components  to interact with the service, send requests, get results,  and even do so across processes with interprocess communication (IPC). </ending> | </end> |
| <hitle> | view binding usuage | <chare> | 1 | <pext> | <starting>view binding usuage</heading> If view binding is enabled for a module, a binding class is generated  for each XML layout file that the module contains.  Each binding class contains references to the root view  and all views that have an ID.  The name of the binding class is generated by converting  the name of the XML file to Pascal case and adding the word "Binding"  to the end. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* <LinearLayout ... >  <TextView android:id="@+id/name" />  <ImageView android:cropToPadding="true" />  <Button android:id="@+id/button"  android:background="@drawable/rounded\_button" /> </LinearLayout> \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Let's say we have TestLayout.xml and the generated binding class is called  TestLayoutBinding. If a button inside the layout has android:id property,  the binding class has a field with the assigned id name to refer it.  If not, there's no reference to it in the binding class. Every binding class also includes a getRoot() method,  providing a direct reference for the root view of the corresponding layout file. The getRoot() method in the the binding class returns the root layout view  like LinearLayout, ConstraintLayout whatever we have in the root in xml. </ending> | </end> |
| <hitle> | view binding in activities | <chare> | 1 | <pext> | <starting>view binding in activities</heading> To set up an instance of the binding class for use with an activity,  We need to do something in the activity's onCreate() method: 1. Call the static inflate() method included in the generated binding class.  This creates an instance of the binding class for the activity to use. 2. Get a reference to the root view by either calling the getRoot() method  or using Kotlin property syntax. 3. Pass the root view to setContentView() to make it the active view  on the screen. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* private lateinit var binding: ResultProfileBinding override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  binding = ResultProfileBinding.inflate(layoutInflater)  val view = binding.root  setContentView(view) } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* We can now use the instance of the binding class to reference  any of the views: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* binding.name.text = viewModel.name binding.button.setOnClickListener { viewModel.userClicked() } </ending> | </end> |
| <hitle> | view binding in fragments | <chare> | 1 | <pext> | <starting>view binding in fragments</heading> To set up an instance of the binding class for use with a fragment,  We need to do something in the fragment's onCreateView() method: 1. Call the static inflate() method included in the generated binding class.  This creates an instance of the binding class for the fragment to use. 2. Get a reference to the root view by either calling the getRoot() method  or using Kotlin property syntax. 3. Return the root view from the onCreateView() method  to make it the active view on the screen. There's onething we have to know is, Fragments outlive their views. So, we need to make sure you clean up any references  to the binding class instance in the fragment's onDestroyView() method. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* private var \_binding: ResultProfileBinding? = null // This property is only valid between onCreateView and // onDestroyView. private val binding get() = \_binding!! override fun onCreateView(  inflater: LayoutInflater,  container: ViewGroup?,  savedInstanceState: Bundle? ): View? {  \_binding = ResultProfileBinding.inflate(inflater, container, false)  val view = binding.root  return view } override fun onDestroyView() {  super.onDestroyView()  \_binding = null } \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* We can now use the instance of the binding class to reference any of the views: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* binding.name.text = viewModel.name binding.button.setOnClickListener { viewModel.userClicked() } </ending> | </end> |
| <hitle> | view binding pros, view-binding vs fidnViewById() | <chare> | 1 | <pext> | <starting>view binding pros, view-binding vs fidnViewById()</heading> 1. Null safety: Since view binding creates direct references to views,  there's no risk of a null pointer exception due to an invalid view ID. Additionally, when a view is only present in some configurations of a layout,  the field containing its reference in the binding class is marked with @Nullable. 2. Type safety: The fields in each binding class have types  matching the views they reference in the XML file.  This means that there's no risk of a class cast exception. ----------------------------------------------------------------------- Using view binding, incompatibilities between the layout and the code  will result in the build failing at compile time rather than at runtime. </ending> | </end> |
| <hitle> | view-binding vs data-binding | <chare> | 1 | <pext> | <starting>view-binding vs data-binding</heading> 1. Faster compilation: View binding requires no annotation processing,  so compile times are faster. 2. Ease of use: View binding does not require specially-tagged XML layout files,  so it is faster to adopt in your apps.  Once you enable view binding in a module, it applies  to all of that module's layouts automatically. 3. View binding doesn't support layout variables or layout expressions,  so it can't be used to declare dynamic UI content straight  from XML layout files. 4. View binding doesn't support two-way data binding. </ending> | </end> |
| <hitle> | tools:viewBindingType for different configurations | <chare> | 1 | <pext> | <starting>tools:viewBindingType for different configurations</heading> When you declare views across multiple configurations,  occasionally it makes sense to use a different view type  depending on the particular layout. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* # in res/layout/example.xml <TextView android:id="@+id/user\_bio" /> # in res/layout-land/example.xml <EditText android:id="@+id/user\_bio" /> \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* View binding supports a tools:viewBindingType attribute,  allowing you to tell the compiler what type to use in the generated code. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* # in res/layout/example.xml (unchanged) <TextView android:id="@+id/user\_bio" /> # in res/layout-land/example.xml <EditText android:id="@+id/user\_bio" tools:viewBindingType="TextView" /> \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* For example, suppose you have two layouts  where one contains a BottomNavigationView and  another contains a NavigationRailView.  Both classes extend NavigationBarView,  which contains most of the implementation details.  If your code doesn't need to know exactly which subclass is present  in the current layout, you can use tools:viewBindingType to set  the generated type to NavigationBarView in both layouts: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* # in res/layout/navigation\_example.xml <BottomNavigationView android:id="@+id/navigation" tools:viewBindingType="NavigationBarView" /> # in res/layout-w720/navigation\_example.xml <NavigationRailView android:id="@+id/navigation" tools:viewBindingType="NavigationBarView" /> </ending> | </end> |
| <hitle> | view-binding validation | <chare> | 1 | <pext> | <starting>sqlite</heading> View binding is unable to validate the value of this attribute  when generating code.  To avoid compile time and runtime errors,  1. The value must be a class that inherits from android.view.View. 2. The value must be a superclass of the tag it is placed on. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* <TextView tools:viewBindingType="ImageView" />  <!-- ImageView is not related to TextView -->  <TextView tools:viewBindingType="Button" />  <!-- Button is not a superclass of TextView --> \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. The final type must resolve consistently across all configurations. </ending> | </end> |
| <hitle> | view binding environment | <chare> | 1 | <pext> | <starting>Migrate view binding</heading> 1. Set the viewBinding build option to true in the module-level build.gradle file android {  ...  buildFeatures {  viewBinding true  } } If the app does not use Parcelize features, remove the line  that enables Kotlin Android Extensions. plugins {  id 'kotlin-android-extensions' } If the app uses Parcelize features, we have to switch to using the standalone kotlin-parcelize Gradle plugin. plugins {  id 'kotlin-parcelize' } 2. Ane then, we need to update activity and fragment classes. 2-1. Remove all imports from kotlinx.android.synthetic. 2-2. Inflate an instance of the generated binding class  for the activity or fragment to use. 2-2-1. For activities, update the activity's onCreate() method  to inflate a view-binding instance.  2-2-2. For fragments, update the fragment's onCreateView() method to inflate a view-binding instance. 2-3. Change all view references to use the binding class instance  instead of synthetic properties: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* // Reference to "name" TextView using synthetic properties. name.text = viewModel.nameString // Reference to "name" TextView using the binding class instance. binding.name.text = viewModel.nameString </ending> | </end> |
| <hitle> | data-binding environment | <chare> | 1 | <pext> | <starting>data-binding environment</heading> To configure your app to use data binding,  enable the dataBinding build option in your build.gradle file in the app module. android {  ...  buildFeatures {  dataBinding true  } } </ending> | </end> |
| <hitle> | data-binding expression language | <chare> | 1 | <pext> | <starting>data-binding expression language</heading> 1. Common expression Mathematical + - / \* % String concatenation + Logical && || Binary & | ^ Unary + - ! ~ Shift >> >>> << Comparison == > < >= <= (Note that < needs to be escaped as &lt;) instanceof Grouping () Literals - character, String, numeric, null Cast Method calls Field access Array access [] Ternary operator ?: \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* android:text="@{String.valueOf(index + 1)}" android:visibility="@{age > 13 ? View.GONE : View.VISIBLE}" android:transitionName='@{"image\_" + id}' \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. Don't use this super new Explicit generic invocation 3.  </ending> | </end> |
| <hitle> | clean architectural principles | <chare> | 1 | <pext> | <starting>clean architectural principles</heading> 1. Separation of concerns: Separation of concerns is a design principle for separating  a computer program into distinct sections. The most important principle to follow is separation of concerns. 2. Drive UI from data models: You should drive your UI from data models, preferably persistent models.  3. Single source of truth When a new data type is defined in your app, you should assign  a Single Source of Truth (SSOT) to it.  4. Unidirectional Data Flow The single source of truth principle is often used  with the Unidirectional Data Flow (UDF) pattern.  In UDF, state flows in only one direction.  The events that modify the data flow in the opposite direction. </ending> | </end> |
| <hitle> | Separation of concerns clean-archi | <chare> | 1 | <pext> | <starting>Separation of concerns</heading> Separation of concerns is a design principle for separating  a computer program into distinct sections. The most important principle to follow is separation of concerns. It's a common mistake to write all your code in an Activity or a Fragment. These UI-based classes should only contain logic that handles UI  and operating system interactions.  By keeping these classes as lean as possible, you can avoid many problems  related to the component lifecycle,  and improve the testability of these classes. </ending> | </end> |
| <hitle> | Drive UI from data models clean-archi | <chare> | 1 | <pext> | <starting>Drive UI from data models</heading> You should drive your UI from data models, preferably persistent models.  Data models represent the data of an app.  They're independent from the UI elements and other components in your app. This means that they are not tied to the UI and app component lifecycle,  but will still be destroyed when the OS decides to remove the app's process  from memory.  Why persistent models are ideal? 1. Your users don't lose data if the Android OS destroys your app  to free up resources. 2. Your app continues to work in cases when a network connection is flaky  or not available.  If you base your app architecture on data model classes,  you make your app more testable and robust. </ending> | </end> |
| <hitle> | Single source of truth clean-archi | <chare> | 1 | <pext> | <starting>Single source of truth clean-archi</heading> When a new data type is defined in your app, you should assign  a Single Source of Truth (SSOT) to it.  The SSOT is the owner of that data, and only the SSOT can modify or mutate it. To achieve this, the SSOT exposes the data using an immutable type,  and to modify the data, the SSOT exposes functions or receive events  that other types can call.  Single source of truth pros: 1. It centralizes all the changes to a particular type of data in one place. 2. It protects the data so that other types cannot tamper with it. 3. It makes changes to the data more traceable. Thus, bugs are easier to spot.  In an offline-first application, the source of truth for application data  is typically a database.  In some other cases, the source of truth can be a ViewModel or even the UI. </ending> | </end> |
| <hitle> | Unidirectional Data Flow clean-archi | <chare> | 1 | <pext> | <starting>Unidirectional Data Flow clean-archi</heading> The single source of truth principle is often used  with the Unidirectional Data Flow (UDF) pattern.  In Unidirectional Data Flow, state flows in only one direction.  The events that modify the data flow in the opposite direction. In Android, state or data usually flow  from the higher-scoped types of the hierarchy to the lower-scoped ones. Events are usually triggered from the lower-scoped types  until they reach the SSOT for the corresponding data type.  For example, application data usually flows from data sources to the UI.  User events such as button presses flow from the UI to the SSOT  where the application data is modified and exposed in an immutable type.  Unidirectional Data Flow pros: This pattern better guarantees data consistency, is less prone to errors, is easier to debug and brings all the benefits of the SSOT pattern. </ending> | </end> |
| <hitle> | clean app architecture | <chare> | 1 | <pext> | <starting>clean app architecture</heading> Application should have 1. The UI layer that displays application data on the screen. 2. The domain layer that simplify and reuse the interactions between  the UI and data layers. 2. The data layer that contains the business logic of your app  and exposes application data.  \*\*\*UI layer\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* The role of the UI layer (or presentation layer) is to display the application data  on the screen. Whenever the data changes, either due to user interaction like  pressing a button, or external input like a network response,  the UI should update to reflect the changes.  The UI layer is made up of: 1. UI elements that render the data on the screen.  We can build these elements using Views or Jetpack Compose functions. 2. And also, State holders like ViewModel classes that hold data,  expose it to the UI, and handle logic.  \*\*\*Domain layer\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* The domain layer is an optional layer that sits between the UI and data layers. The domain layer is responsible for encapsulating complex business logic,  or simple business logic that is reused by multiple ViewModels.  This layer is optional because not all apps will have these requirements.  We should use it only when needed—for example, to handle complexity  or favor reusability. Classes in this layer are commonly called use cases or interactors.  Each use case should have responsibility over a single functionality.  For example, your app could have a GetTimeZoneUseCase class  if multiple ViewModels rely on time zones to display the proper message  on the screen.  \*\*\*Data Layer\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* The data layer of an app contains the business logic.  The business logic is what gives value to your app—it's made of rules  that determine how your app creates, stores, and changes data. The data layer is made of repositories that each can contain  zero to many data sources.  We should create a repository class for each different type of data  you handle in your app.  Repository classes are responsible for: 1. Exposing data to the rest of the app. 2. Centralizing changes to the data. 3. Resolving conflicts between multiple data sources. 4. Abstracting sources of data from the rest of the app. 5. Containing business logic. Each data source class should have the responsibility of working  with only one source of data, which can be a file, a network source,  or a local database.  Data source classes are the bridge between the application and the system  for data operations. </ending> | </end> |
| <hitle> | best practices clean-archi | <chare> | 1 | <pext> | <starting>best practices clean-archi</heading> 1. Don't store data in app components. We have to avoid designating your app's entry points—such as activities,  services, and broadcast receivers—as sources of data.  Instead, they should only coordinate with other components  to retrieve the subset of data that is relevant to that entry point.  Each app component is rather short-lived,  depending on the user's interaction with their device  and the overall current health of the system. 2. Reduce dependencies on Android classes. The app components should be the only classes  that rely on Android framework SDK APIs such as Context, or Toast. Abstracting other classes in your app away from them helps with testability  and reduces coupling within your app. 3. Create well-defined boundaries of responsibility  between various modules in your app. For example, don't spread the code that loads data from the network  across multiple classes or packages in your code base.  Similarly, don't define multiple unrelated responsibilities—such as data caching  and data binding—in the same class.  4. Expose as little as possible from each module. For example, don't be tempted to create a shortcut  that exposes an internal implementation detail from a module.  5. Focus on the unique core of your app so it stands out from other apps. Don't reinvent the wheel by writing the same boilerplate code again and again. Instead, We need to make the app unique,  and let the Jetpack libraries and other recommended libraries  handle the repetitive boilerplate. 6. Consider how to make each part of your app testable in isolation. For example, having a well-defined API for fetching data from the network  makes it easier to test the module that persists that data in a local database.  If instead, you mix the logic from these two modules in one place,  or distribute your networking code across your entire code base,  it becomes much more difficult to test effectively. 7. Types are responsible for their concurrency policy. If a type is performing long-running blocking work, it should be responsible  for moving that computation to the right thread.  That particular type knows the type of computation that it is doing and  in which thread it should be executed.  Types should be main-safe, meaning they're safe to call from the main thread  without blocking it. 8. Persist as much relevant and fresh data as possible. That way, users can enjoy your app's functionality  even when their device is in offline mode.  </ending> | </end> |
| <hitle> | clean architectural pros | <chare> | 1 | <pext> | <starting>clean architectural pros</heading> 1. It improves the maintainability, quality and robustness of the overall app. 2. It allows the app to scale. More people and more teams can contribute  to the same codebase with minimal code conflicts. 3. It helps with onboarding. As Architecture brings consistency to your project,  new members of the team can quickly get up to speed and be more efficient  in less amount of time. 4. It is easier to test. A good Architecture encourages simpler types  which are generally easier to test. 5. Bugs can be investigated methodically with well defined processes. </ending> | </end> |
| <hitle> | fragment pros | <chare> | 1 | <pext> | <starting>fragment pros</heading> 1. We can reuse a Fragment. Write the Fragment code once,  and reuse the Fragment in more than one Activity without having  to repeat code. 2. We can add or remove a Fragment dynamically.  Add, replace, or remove a Fragment from an Activity as needed. 3. We can integrate a mini-UI within the Activity. Integrate a Fragment  with an Activity UI or overlay the UI, so that the user can interact  with the Fragment UI without leaving the Activity. 4. We can retain data instances after a configuration change.  Since a Fragment has its own lifecycle, it can retain an instance of its data  after a configuration change (such as changing the device orientation). 5. We can represent sections of a layout for different screen sizes.  Encapsulating an interactive UI within a Fragment makes it easier  to display the interactive UI on different screen sizes. </ending> | </end> |