|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <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> | DP vs SP | <chare> | 1 | <pext> | Key Difference: is SP and DP define two different units used in android  for defining layout properties like margins, text size, etc.  SP stands for scale-independent pixels or scale per pixel,  and DP is density-independent pixels or density per pixel.  SP and DP are quiet the same,  except that SP has an additional scaling factor  based on the user’s preferred text size.  SP must be used for defining text sizes but not for layout sizes. | </end> |
| <hitle> | JVM Java Virtual Machine | <chare> | 1 | <pext> | The JVM is a virtual machine to run Java desktop, server,  and web applications.  The JVM has been shaped to support multiple host architectures  and run everywhere. But, it is too heavy for embedded devices.  Java or Kotlin file is processed by JVM  and JVM generates Java Byte coe - \*.jar file. | </end> |
| <hitle> | DVM Dalvik Virtual Machine | <chare> | 1 | <pext> | The DVM is a virtual machine to run Android applications.  The DVM executes Dalvik bytecode,  which is compiled from programs written in the Java language.  Java Byte code is processed by DVM  and DVM generates Dalvik Byte code - \*.dex.  And this dex file is processed by Android Processor. | </end> |
| <hitle> | Android OS Architecture | <chare> | 1 | <pext> | Android os architecture contains different number of components  to support any android device needs:  1. Applications  Applications is the top layer of android architecture.  The pre-installed applications like home, contacts, camera, gallery etc  and third party applications downloaded from the play store  like chat applications, games etc. will be installed on this layer only.  It runs within the Android run time with the help of the classes  and services provided by the application framework.  2. Application framework  Application Framework provides several important classes  which are used to create an Android application.  It provides a generic abstraction for hardware access and  also helps in managing the user interface with application resources.  Generally, it provides the services with the help of which  we can create a particular class and make that class helpful  for the Applications creation.  It includes different types of services activity manager,  notification manager, view system, package manager etc.  3. Application runtime  Android Runtime environment is one of the most important part of Android.  It contains components like core libraries  and the Dalvik virtual machine(DVM).  Mainly, it provides the base for the application framework  and powers our application with the help of the core libraries.  4. Platform libraries  The Platform Libraries includes various C/C++ core libraries  and Java based libraries such as Media,  Graphics, Surface Manager, OpenGL etc  to provide a support for android development.  5. Linux Kernel  Linux Kernel is heart of the android architecture.  It manages all the available drivers such as display drivers,  camera drivers, Bluetooth drivers, audio drivers, memory drivers, etc  which are required during the runtime.  The Linux Kernel will provide an abstraction layer  between the device hardware and the other components  of android architecture.  It is responsible for management of memory, power, devices etc. | </end> |
| <hitle> | Activity | <chare> | 1 | <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> | 1 | <pext> | Service is an application component and designed specifically  to allow applications to initiate and perform background tasks.  services are designed to perform tasks that take a long time to complete  like downloading a file over an internet connection  or streaming music to the user, but do not require a user interface. A service can run continuously in the background  even if the application is closed  or even after the user switches to another application. --------------Types of Service---------------  1. Started Services / Background Services  2. Bound Services  3. Foreground Services  ----------Started Services / Background Services-------------  Started Service also called background service  performs an operation that isn’t directly noticed by the user.  For example, if an app used a service to compact its storage,  that would usually be a background service.  -----------Bound Services-----------  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, receive results,  and even do so across processes with IPC (interprocess communication).  ------------ Foreground Services--------------  A foreground service is nothing but a service that executes tasks  that are noticeable to the user.  For notifying users, we need to show notifications in the status bar.  These notifications should have a priority of PRIORITY\_LOW or higher.  The notification cannot be removed unless the task is finished  and the service is stopped or removed by itself or the system  due to various factors.  This way, users will be aware that some work is being performed  in the background that may consume system resources.  A simpler example of a foreground service would be any audio player app.  ------------Why Foreground Services--------------  If the apps that are targeting Android are trying to use a normal service  to do background operations when the app is not live,  then the system will kill this service and we can’t complete our tasks.  For the smooth handling of things in the background,  we need to use a foreground service notifying the users.  ------------Life Cycle of Android Services-------------  1. Started Service (Unbounded Service):  By following this path, a service will initiate  when an application component calls the startService() method.  Once initiated, the service can run continuously in the background  even if the component is destroyed which was responsible  for the start of the service.  stopService() and stopSelf() are available to stop the execution of service.  2. Bounded Service:  It can be treated as a server in a client-server interface.  By following this path, android application components can send requests  to the service and can fetch results.  A service is termed as bounded when an application component  binds itself with a service by calling bindService() method.  To stop the execution of this service, all the components  must unbind themselves from the service by using unbindService() method.  -------------Service lifecycle callbacks--------------------  1. onStartCommand()  The Android service calls this method when a component like activity  requests to start a service using startService().  Once the service is started, it can be stopped explicitly using stopService()  or stopSelf() methods.  2. onBind()  This method is mandatory to implement in android service  and is invoked whenever an application component calls  the bindService() method in order to bind itself with a service.  UI is also provided to communicate with the service effectively  by returning an IBinder object.  If the binding of service is not required then the method must return null.  3. onUnbind()  The Android system invokes this method when all the clients  get disconnected from a particular service interface.  4. onRebind()  Once all clients are disconnected from the particular interface of service  and there is a need to connect the service with new clients,  the system calls this method.  5. onCreate()  Whenever a service is created either using onStartCommand()  or onBind(), the android system calls this method.  This method is necessary to perform a one-time-set-up.  6. onDestroy()  When a service is no longer in use, the system invokes this method.  ---------------Service lifecycle flow--------------------  1. startService()  onCreate()  onStartCommand()  onDestroy()  2. bindService()  onCreate()  onBind()  onUnbind() – onRebind()  onDestroy()  ----------------IBinder----------------------  To create a bound service, we must define the interface that specifies  how a client can communicate with the service.  This interface between the service and a client must be  an implementation of IBinder and is what the service must return  from the onBind() callback method.  After the client receives the IBinder,  it can begin interacting with the service through that interface.  ----------howto bind/bound activity to service-----------  1. We had to override an onBind() method in our service  We need to make this onBind() method return a Binder object  that can be used to communicate with the service.  2. To implement a binder, the first thing we need to do  is create a public inner class in our service that extends Binder.  Then, we can add a single method in your binder class  that returns an instance of the containing service.  This allows any binding activities to directly access the service  using a Service object.  3. Then, we just return an instance of that binder in the onBind() method.  We'll also set up a public method in our service to interact with activity.  4. We have to implement the ServiceConnection interface in the activity  that is doing the binding.  The ServiceConnection interface contains two callbacks that are used  to relay information about the connection with the bound service.  4-1. onServiceConnected() is called when the service becomes bound.  This method also contains an IBinder object which represents the binder  we returned in our interface.  4-2. onServiceDisconnected() is used to inform the binding activity  that the service has crashed or has been killed by the system  and that the service is no longer bound.  public class MyBoundService extends Service {  public class MyBoundServiceBinder extends Binder {  public MyBoundService getService() {  return MyBoundService.this;  }  }  @Override  public IBinder onBind(Intent intent) {  return new MyBoundServiceBinder();  }  public void showToast() {  Toast.makeText(this, "Some text!", Toast.LENGTH\_SHORT).show();  }  }  public class MainActivity extends Activity implements ServiceConnection {  @Override  protected void onStart() {  super.onStart();  Intent intent = new Intent(this, BoundService.class);  bindService(intent, this, Context.BIND\_AUTO\_CREATE);  }  @Override  protected void onStop() {  super.onStop();  unbindService(this);  }  @Override  public void onServiceConnected(ComponentName name, IBinder service) {  MyBoundServiceBinder binder = (MyBoundServiceBinder)service;  MyBoundService myService = binder.getService();  myService.showToast();  }  @Override  public void onServiceDisconnected(ComponentName name) {  }  }  -----------pass data between activity and servie exchange---------------  1. Create Static Variables and Methods:  This is the simplest method.  Make the variables static and then both the Activity and Service  can access and update the variables at the same time.  2. Create a Bound Service (Best for Two-way Communication):  A bound service is a server in a client-server interface.  It allows activities to bind to the service to send and receive the data.  It typically lives only while it servers the components (like activity)  and doesn’t run in the background indefinitely.  3. Use Intent to Send Data to Service:  This is the easiest way to send data to the Service.  While calling the startService() method, add data to the intent object.  4. Create a Singleton Instance of the Service:  We can also create a singleton instance of the service to share data.  5. Use Broadcast Receiver:  We can use broadcasts to send data to the Service.  -------------Alarm Manager--------------  Android AlarmManager allows to access system alarm.  By this, we can schedule the application to run at a specific time in the future.  It works whether your phone is running or not.  The Android AlarmManager holds a CPU wake lock  that provides guarantee not to sleep the phone until broadcast is handled.  -------------Service restart options, onStartCommand return value--------  The onStartCommand() callback method is required to return  an integer value to define what should happen with  regard to the service in the event that it is destroyed  by the Android runtime system.  Possible return values for these methods are:  1. START\_NOT\_STICKY indicates to the system that the service  should not be restarted in the event that it is destroyed  unless there are pending intents awaiting delivery.  It’s the best option to avoid running a service  in case if it is not necessary.  2. START\_STICKY indicates that the service should be restarted  as soon as possible after it has been destroyed  if the destruction occurred after the onStartCommand() method returned.  In the event that no pending intents are waiting to be delivered,  the onStartCommand() callback method is called with a NULL intent value.  The intent being processed at the time  that the service was destroyed is discarded.  This is suitable for the service which are not executing commands  but running independently and waiting for the job.  3. START\_REDELIVER\_INTENT indicates that,  if the service was destroyed after returning  from the onStartCommand() callback method,  the service should be restarted with the current intent redelivered  to the onStartCommand() method followed by any pending intents.  This is useful for services that are receiving commands of work to do,  and want to make sure they do eventually complete the work  for each command sent. | </end> |
| <hitle> | IntentService | <chare> | 1 | <pext> | IntentService in Android is a base class for Services  to handle asynchronous requests that are expressed in the form of Intents  when there is a demand to do so.  The requests are sent by the clients  through Context.startService(Intent) calls to start the service.  It handles each Intent one by one with the help of a worker thread  and stops automatically when the work is done.  All the requests are controlled using a single worker thread  so they may be sometimes time-consuming.  However, they do not interfere with the main loop of the application.  -------Howto IntentService----------------  1. Create a new class called ExampleIntentService  that extends IntentService and override the onHandleIntent() method  to specify the task that you want to perform in the background.  class ExampleIntentService : IntentService("ExampleIntentService") {  // This method is called on a worker thread  // with a request to process.  override fun onHandleIntent(intent: Intent?) {  // Perform a task in the background  Log.d("ExampleIntentService", "Task in progress")  try {  // Perform the task for 5 seconds  Thread.sleep(5000)  } catch (e: InterruptedException) {  // Print the stack trace  // if an interruption occurs  e.printStackTrace()  }  Log.d("ExampleIntentService", "Task completed")  }  }  2. In the activity create the instance of IntentService by Intent.  And then start it by startService()  startServiceButton.setOnClickListener {  // Create an Intent to start  // the ExampleIntentService  val intent = Intent(this@MainActivity, ExampleIntentService::class.java)  // Start the service  // using the intent  startService(intent)  }  3. In the AndroidManifest.xml file, we need to declare this service  within the <application> element.  <service  android:name=".ExampleIntentService"  android:exported="false" /> | </end> |
| <hitle> | JobIntentService | <chare> | 1 | <pext> | JobIntentService works in the same way as a Service  however it enqueues the work into the JobScheduler.  This means you can easily convert your Services and IntentServices  into a JobIntentService and keep the same functionality.  The JobIntentService also exposes a static method to start the service (JobIntentService.enqueueWork()).  For device targeting SDK 26 or later, This class’ works are dispatched  via JobScheduler class and for SDK 26 or below devices,  It uses Context.startService() (Same as in IntentService)  ----------HowTo JobIntent Service-----------  1. Create a subclass of JobIntentService  public class MyJobIntentService extends JobIntentService {  }  2. Now override the onHandleWork() methods  and expose enqueueWork() methods.  private static final int JOB\_ID = 1;  public static void enqueueWork(Context context, Intent intent) {  enqueueWork(context, MyJobIntentService.class, JOB\_ID, intent);  }  @Override  protected void onHandleWork(@NonNull Intent intent) {  /\*\*  Write code here..  Perform Long operation here such as Download/Upload of file.  The system or framework is already holding a wake lock for us at this point.  \*/  }  3. Configuration in AndroidManifest.xml file  <uses-permission android:name="android.permission.WAKE\_LOCK"/>  4. Call from Activity/ Fragment/Receiver  public void onStartJobIntentService() {  Intent mIntent = new Intent(this, MyJobIntentService.class);  MyJobIntentService.enqueueWork(this, mIntent);  } | </end> |
| <hitle> | Service vs IntentService | <chare> | 1 | <pext> | 1. When to use which one?  1-1. The Service can be used in tasks with no UI, but shouldn’t be too long.  If you need to perform long tasks, you must use threads within Service.  1-2. The IntentService can be used in long tasks usually  with no communication to Main Thread.  If communication is required, we can use Main Thread handler  or broadcast intents.  Another case of use is when callbacks are needed (Intent triggered tasks).  2. Runs On which thread?  2-1. The Service runs in background  but it runs on the Main Thread of the application.  2-2. The IntentService runs on a separate worker thread.  3. How to start?  3-1. The Service is triggered by calling method startService().  3-2. The IntentService is triggered using an Intent,  it spawns a new worker thread and the method onHandleIntent()  is called on this thread.  4. Triggered From which thread?  The Service and IntentService may be triggered from any thread,  activity or other application component.  5. Disadvantages of using these?  5-1. The Service may block the Main Thread of the application.  5-2. The IntentService cannot run tasks in parallel.  Hence all the consecutive intents will go into the message queue  for the worker thread and will execute sequentially. | </end> |
| <hitle> | Activity vs Service | <chare> | 1 | <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> | 1 | <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> | Bundle | <chare> | 1 | <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> | 1 | <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> | 1 | <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> | Android:launch mode | <chare> | 1 | <pext> | A “launch mode” is the way in which a new instance of an activity  is to be associated with the current task.  -----------launch mode Task, backstack-----------  A task holds a collection of activities that are arranged  in a stack called the backstack.  The first activity to be launched in a task is the root activity.  Pressing the back button on a root activity not only kills the activity,  but it kills the task, and possibly the app. -------------define launch mode------------  We can define launch mode in Manifestfile or Intent flags.  -----------define launch mode Manifest------------  1. Manifest file. When declaring an activity in a manifest file,  you can specify how the activity should associate with tasks when it starts. There are 4 of supported values are:  \*standard\*, \*singleTop\*, \*singleTask\*, \*singleInstance\*  -------------standard(default) launch mode------------  1-1. standard (default) is the default launch mode.  Every new instance of the activity will be put on top of the stack  as a separate entity.  \*Example\*  Let’s say you have a task with a root activity called Activity A,  and then launch a new activity called B,  B gets pushed to the top of the stack.  Now let’s say you launch another Activity B from the B we just created,  you will now have two instances of Activity B stacked on top of one another.  If a user presses the back button,  they will go back to another instance of Activity B.  -------------singleTop------------- 1-2. singleTop stops different instances of the same activities  from being stacked on top of one another.  And no new instance will be created  because it will fire off an onNewIntent() method  instead of creating a new object.  \*Example\*  Let’s say Activity B is a single top.  We can define launch modes in the manifest.  <activity android:name=".Activity\_B" android:launchMode="singleTop"/>  So now if you launch Activity B from Activity B,  instead of creating a new instance of B and stacking it on top of the old B,  the intent is passed into the current instance of B.  The only way to have two instances of the same activity in one task  is to launch a different activity from B, and then create B from that activity.  The main concept of the single top launch mode is that  you can not have two instances of the same activity  stacked on top of one another.  ----------------singleTop with FLAG\_ACTIVITY\_CLEAR\_TOP----------  If you don’t want 2 instances of the same single top activity in a task  while using single top launch mode,  you can pass the FLAG\_ACTIVITY\_CLEAR\_TOP constant in your intent.  \*Example\*  let’s say your activity stack is ABC (B is still single top).  If you launch B from C with FLAG\_ACTIVITY\_CLEAR\_TOP,  your stack will pop all the way down to B and the intent will be passed  to that instance of B.  Intent intent = new Intent(this, Activity\_B.class);  intent.addFlags(Intent.FLAG\_ACTIVITY\_CLEAR\_TOP);  startActivity(intent);  -------------singleTask----------- 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.  \*Example\*  if your stack is ABC, and you launch B (a single task) from C,  we pop all the way down to B, and pass the intent to that instance.  ------------- task affinity---------------  But what if we want B to be in its own task?  This is where task affinity comes into play.  Task affinity lets you define which task an activity belongs to.  By default, an activity has the same task affinity as its root activity.  With task affinity, we can now separate activities into different tasks.  <activity android:name=".Activity\_A"  android:taskAffinity="com.affinity.of.a">  <intent-filter>  <action android:name="android.intent.action.MAIN" />  <category android:name="android.intent.category.LAUNCHER" />  </intent-filter>  </activity>  <activity android:name=".Activity\_B"  android:launchMode="singleTask"  android:taskAffinity="com.affinity.of.b"  />  Here, Activity A and Activity B have different task affinities.  When A is created in Task 1, C is launched from A.  By default, each activity has the same affinity as its root,  so we are still in Task 1. Now B is launched from C.  B has a different task affinity, so it is now the root of Task 2,  which then moves to the foreground.  If we launch a new instance of Activity C from B, C belongs to Task 2  because that is the task affinity of the root activity (in this case, B).  But what if we now try to launch A from C in Task 2?  Since A‘s affinity is Task 1, we shift from Task 2 to Task 1,  pop all the way back to Activity A, and finally pass the intent to A.  -------------singleInstance------------  1-4. singleInstance. Like singleTask, an Activity with Single Instance  can be the only Activity across all tasks.  The difference between singleTask and singleInstance is that  an Activity with Single Instance can also be the only activity in a task.  <activity android:name=".Activity\_B"  android:launchMode="singleInstance"  android:taskAffinity="com.affinity.of.b"  />  \*Example\*  Activity B will have a launch mode of Single Instance.  Activity A in Task 1 launches Activity B.  This makes Activity B launch in a new task,  which is then put in the foreground.  Activity B then launches Activity C.  Since a Single Instance can be the only activity in a task,  C is launched on top of Activity A in Task 1,  and then Task 1 comes to the foreground.  Task affinity also plays a role in Single Instance.  If Activity B has no task affinity, a user can not navigate back to Task 1.  If B did have task affinity,  a user will be able to go back and forth between Task 1 and 2.  -----------define launch mode Intent flags------ 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:  FLAG\_ACTIVITY\_NEW\_TASK,  FLAG\_ACTIVITY\_SINGLE\_TOP,  FLAG\_ACTIVITY\_CLEAR\_TOP  ------------- FLAG\_ACTIVITY\_NEW\_TASK----------------------- 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.  -------------FLAG\_ACTIVITY\_SINGLE\_TOP-------------- 2-2. FLAG\_ACTIVITY\_SINGLE\_TOP.  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.  ------------- FLAG\_ACTIVITY\_CLEAR\_TOP-------------- 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. | </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 | <chare> | 1 | <pext> | Broadcasts are messages that the Android system and Android apps send  when events occur that might affect the functionality of other apps.  In general, broadcasts are messaging components used  for communicating across apps when events of interest occur.  For example, the Android system sends an event  when the system boots up, when power is connected or disconnected,  and when headphones are connected or disconnected.  Android apps can also broadcast events, for example  when new data is downloaded.  -----------types of broadcasts-----------  1. System broadcasts are delivered by the system.  2. Custom broadcasts are delivered by your app.  2-1. Normal broadcast by sendBroadcast().  2-2. Ordered broadcast by sendOrderedBroadcast().  2-3. Local broadcast by LocalBroadcastManager.sendBroadcast().  -------------System broadcasts----------  A system broadcast is a message that the Android system sends  when a system event occurs.  System broadcasts are wrapped in Intent objects.  The intent object's action field contains event details.  The intent can also contain more data about the event  in its extra field.  -----------System broadcast Example-----------  1. When the device boots, the system broadcasts a system Intent  with the action ACTION\_BOOT\_COMPLETED.  2. When the device is disconnected from external power,  the system sends a system Intent with the action field  ACTION\_POWER\_DISCONNECTED.  -----------System broadcasts target--------------  System broadcasts aren't targeted at specific recipients.  Interested apps must register a component to "listen" for these events.  This listening component is called a broadcast receiver.  ----------Custom broadcasts----------  Custom broadcasts are broadcasts that app what we make sends out.  We can use a custom broadcast when we want app to take an action  without launching an activity.  More than one broadcast receiver can be registered  to receive your broadcast.  ----------Custom broadcast example----------  For example, we can use a custom broadcast  when we want to let other apps know that data has been downloaded  to the device and is available for them to use.  -----------howto custom broadcast---------------  To create a custom broadcast, define a custom Intent action.  When we specify the action for the Intent,  we have to use unique package name to make sure that the intent  doesn't conflict with an intent that is broadcast from a different app  or from the Android system.  ---------------types of custom broadcast----------  1. Normal broadcast  For a normal broadcast, pass the intent to sendBroadcast().  2. Ordered broadcast  For an ordered broadcast, pass the intent to sendOrderedBroadcast().  3. Local broadcast  For a local broadcast, pass the intent to LocalBroadcastManager.sendBroadcast().  ------------Normal broadcasts------------  The sendBroadcast() method sends broadcasts  to all the registered receivers at the same time,  in an undefined order. This is called a normal broadcast.  A normal broadcast is the most efficient way to send a broadcast.  With normal broadcasts,  receivers can't propagate the results among themselves,  and they can't cancel the broadcast.  --------howto send Normal broadcasts----------  For example, the following method sends a normal broadcast to all interested broadcast receivers:  public void sendBroadcast() {  Intent intent = new Intent();  intent.setAction("com.example.myproject.ACTION\_SHOW\_TOAST");  // Set the optional additional information in extra field.  intent.putExtra("data","This is a normal broadcast");  sendBroadcast(intent);  }  ------------Ordered broadcasts-----------------  To send a broadcast to one receiver at a time,  we can use the sendOrderedBroadcast() method.  The android:priority attribute that's specified in the intent filter  determines the order in which the broadcast is sent.  If more than one receiver with same priority is present,  the sending order is random.  The Intent is propagated from one receiver to the next.  During its turn, a receiver can update the Intent,  or it can cancel the broadcast.  (If the receiver cancels the broadcast, the Intent can't be propagated further.)  ----------howto send Ordered broadcasts--------  For example, the following method sends an ordered broadcast to all interested broadcast receivers:  public void sendOrderedBroadcast() {  Intent intent = new Intent();  // Set a unique action string prefixed by your app package name.  intent.setAction("com.example.myproject.ACTION\_NOTIFY");  // Deliver the Intent.  sendOrderedBroadcast(intent);  }  ------------Local broadcasts------------------  If we don't need to send broadcasts to a different app,  we can use the LocalBroadcastManager.sendBroadcast() method,  which sends broadcasts to receivers within the app.  This method is efficient, because it doesn't involve  interprocess communication.  Also, using local broadcasts protects the app against some security issues.  --------howto send local broadcast-----------  1. To get an instance of LocalBroadcastManager,  call getInstance() and pass in the application context.  2. Call sendBroadcast() on the instance.  Pass in the intent that you want to broadcast.  LocalBroadcastManager.getInstance(this).sendBroadcast(customBroadcastIntent); | </end> |
| <hitle> | Non-ordered vs Ordered Broadcasts | <chare> | 2 | <pext> | 1. In non-ordered mode, broadcasts are sent  to all interested receivers “at the same time”.  This basically means that one receiver can not interfere in any way  with what other receivers will do neither can it prevent  other receivers from being executed.  One example of such broadcast is the ACTION\_BATTERY\_LOW one.  2. In ordered mode, broadcasts are sent to each receiver in order  and one receiver is able to abort the broadcast so that receivers  with a lower priority would not receive it (thus never execute).  The order is controlled by the android:priority attribute  for the intent-filter element in the manifest file that is related to your receiver.  An example of this type of broadcast is the ACTION\_NEW\_OUTGOING\_CALL one. | </end> |
| <hitle> | broadcast receiver | <chare> | 1 | <pext> | 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> ```  -----------types of broadcast receiver-----------  1. Static Broadcast Receivers:  These types of Receivers are declared in the manifest file  and works even if the app is closed.  2. Dynamic Broadcast Receivers:  These types of receivers work only if the app is active or minimized.  -----------two ways to register broadcast receiver---------------  Static registration, Dynamic registration  -----------Static registration-----------------  1. Static registration  Toregister broadcast receiver statically, we can add a <receiver> element  to AndroidManifest.xml file.  Within the <receiver> element:  Use the path to BroadcastReceiver subclass  as the android:name attribute.  To prevent other applications from sending broadcasts to your receiver,  set the optional android:exported attribute to false.  This is an important security guideline.  To specify the types of intents the component is listening for,  use a nested <intent-filter> element.  <receiver  android:name="com.example.myproject.AlarmReceiver"  android:exported="false">  <intent-filter>  <action android:name="com.example.myproject.intent.action.ACTION\_SHOW\_TOAST"/>  </intent-filter>  </receiver>  ------------- Dynamic registration and unregistration------------  We can also register a broadcast receiver dynamically,  which ties its operation to the lifecycle of your activity.  To register your receiver dynamically, call registerReceiver()  and pass in the BroadcastReceiver object and an intent filter.  IntentFilter intentFilter = new IntentFilter();  intentFilter.addAction(ACTION\_SHOW\_TOAST);  mReceiver = new AlarmReceiver();  registerReceiver(mReceiver, intentFilter);  You also need to unregister the receiver by calling unregisterReceiver()  and passing in BroadcastReceiver object  unregisterReceiver(mReceiver);  Where we call these methods depends on the desired lifecycle  of BroadcastReceiver object:  If the receiver is only needed when activity is visible,  then register the receiver in onResume().  And Unregister the receiver in onPause().  We can also use the onStart()/onStop() or onCreate()/onDestoy() method pairs,  if they are more appropriate for use case.  ---------------BraodcastManager---------------  BroadCast Manager is used when notifies users with a message, mail,  or any other format regarding a specific event.  -----------Local Broadcast Manager howto send/register----------------  LocalBroadcastManager is used to register and send a broadcast of intents  to local objects.  To avoid having to manage the security aspects,  we can use the LocalBroadcastManager class.  LocalBroadcastManager lets you send and receive broadcasts  within a single process and a single application,  which means you don't have to worry about cross-application security.  To send a broadcast using LocalBroadcastManager:  1. Get an instance of LocalBroadcastManager by calling getInstance()  and passing in the application context.  2. Call sendBroadcast() on the instance, passing in the intent  that you want to broadcast.  LocalBroadcastManager.getInstance(this).sendBroadcast(customBroadcastIntent);  To register your receiver to receive only local broadcasts:  1. Get an instance of LocalBroadcastManager by calling getInstance()  and passing in the application context.  2. Call registerReceiver(), passing in the receiver and an intent filter  as you would for a regular broadcast receiver.  You must register local receivers dynamically,  because static registration in the manifest is unavailable.  LocalBroadcastManager.getInstance(this).registerReceiver  (mReceiver, new IntentFilter(CustomReceiver.ACTION\_CUSTOM\_BROADCAST));  To unregister the broadcast receiver:  LocalBroadcastManager.getInstance(this).unregisterReceiver(mReceiver);  ------------send custom broadcast------------  To send custom broadcast, we can use sendBroadcast()  with an explicit Intent that identifies the receiver.  sendBroadcast(new Intent(this, MyReceiver.class).setAction("MyAction"));  -------------------Sticky broadcast--------------  Sticky broadcasts are a special type of broadcast for which  the sent intent object remains in the cache after the broadcast is complete.  The system may re-broadcast sticky intents to later registrations of receivers.  ----------------secure broadcast, receiver restrict---------------  1. Restrict your app to receive broadcast  1-1. Specify a permission parameter when registering a broadcast receiver  then only broadcasters who have requested the permission  can send an Intent to the receiver.  1-2. Set the android:exported attribute to “false” in the manifest.  This restrict to receive broadcasts from sources outside of the app.  1-3. Limit yourself to only local broadcasts with LocalBroadcastManager.  2. Control receiver of your broadcast  2-1. We can specify a permission when sending a broadcast  then only receivers who have requested that permission  can receive the broadcast.  2-2. In Android 4.0 and higher, we can specify a package  with setPackage(String) when sending a broadcast.  The system restricts the broadcast to the set of apps that match the package.  2-3. Send local broadcasts with LocalBroadcastManager.  ---------single broadcast receiver for multiple events/actions------------  1. Dynamically  1-1. BroadcastReceiver can handle multiple actions,  we can add them to the intent filter,  and then register it with that filter.  1-2. And inside your own receiver switch for actions.  private BroadcastReceiver myReceiver;  @Override  public void onReceive(Context context, Intent intent) {  IntentFilter myFilter = new IntentFilter();  myFilter.addAction("action1");  myFilter.addAction("action2");  registerReceiver(myReceiver, myFilter);  }  @Override  public void onReceive(Context context, Intent intent) {  String action = intent.getAction();  if(action != null) {  if(action.equals("action1") {  // CODE  } else if (action.equals("action2") {  // CODE  }  }  }  2. Statically  2-1. We can register one or more action for broadcast receiver  using intentfilter in the AndroidManifest file.  2-2. And then check the Intent's action in the onReceive of the receiver.  <receiver android:name=".myReceiver">  <intent-filter android:priority="1000000">  <action android:name="android.intent.action.ACTION\_HEADSET\_PLUG" />  </intent-filter>  <intent-filter android:priority="1000000">  <action android:name="android.intent.action.MEDIA\_BUTTON" />  </intent-filter>  </receiver>  @Override  public void onReceive(Context context, Intent intent) {  String action = intent.getAction();  if(action != null) {  if(action.equals("action1") {  // CODE  } else if (action.equals("action2") {  // CODE  }  }  }  -------------Broadcast Receiver pros------------  1. A Broadcast receiver wakes your application up,  the inline code works only when your application is running.  2. No UI but can start an Activity.  3. It has maximum limit of 10secs,  We should not do any asynchronous operations  which may take more time, also whe should not do  heavy database operations or networking operations  in broadcast receiver. | </end> |
| <hitle> | Serializable vs Parcelable | <chare> | 1 | <pext> | 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.  -----------Serializable-------------  Serializable is a Java interface that enables an object to be serialized,  meaning that it can be converted into a byte stream and stored in a file  or transmitted over a network.  When an object is serialized, it is converted into a byte stream  that is easily readable by the receiving end.  The receiving end can then convert the byte stream back into an object.  Serialization is a quick and efficient way of transferring data across activities.  ---------Howto Serializable---------  Creating a Serializable object is as simple  as implementing the java.io.Serializable interface in the class  and there are no methods to override.  ----------- Parcelable-------------  Parcelable is an Android-specific interface that enables an object  to be passed as a parameter from one activity to another.  This is a more efficient method compared to serialization,  as it doesn’t require the object to be converted into a byte stream.  When an object is passed using parcelable,  it is passed directly from one activity to another.  -----------Howto Parcelable ---------------  The class must implement the Parcelabel interface  and fill all its required methods, then, our class must have  a non-null static field called CREATOR with the type of Parcelable.Creator.  By using kotlin-parcelize plugin we can rely on Kotlin  to create our Parcelable object by adding @Parcelize annotation to our class  and implementing the Parcelable interface in it  without overriding any methods: | </end> |
| <hitle> | SQLite database in android | <chare> | 1 | <pext> | Android SQLite is a very lightweight database which comes with Android OS.  Android SQLite combines a clean SQL interface  with a very small memory footprint and decent speed.  For Android, SQLite is “baked into” the Android runtime,  so every Android application can create its own SQLite databases.  Once a database is created successfully its located in data/data//databases/  accessible from Android Device Monitor.  SQLite is a typical relational database, containing tables  which consists of rows and columns, indexes etc.  We can create our own tables to hold the data accordingly.  This structure is referred to as a schema.  Android has features available to handle changing database schemas,  which mostly depend on using the SQLiteOpenHelper class.  -----------SQLiteOpenHelper------------  SQLiteOpenHelper is designed to get rid of two very common problems.  1. When the application runs the first time –  At this point, we do not yet have a database.  So we will have to create the tables, indexes, starter data, and so on.  2. When the application is upgraded to a newer schema –  Our database will still be on the old schema  from the older edition of the app.  We will have option to alter the database schema  to match the needs of the rest of the app.  SQLiteOpenHelper wraps up these logic to create  and upgrade a database as per our specifications.  For that we’ll need to create a custom subclass of SQLiteOpenHelper  implementing at least the following three methods.  1. Constructor:  This takes the Context (e.g., an Activity), the name of the database,  an optional cursor factory, and an integer representing the version  of the database schema you are using (typically starting from 1 and increment later).  2. onCreate(SQLiteDatabase db) :  It’s called when there is no database and the app needs one.  It passes us a SQLiteDatabase object, pointing to a newly-created database,  that we can populate with tables and initial data.  3. onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion):  It’s called when the schema version we need  does not match the schema version of the database,  It passes us a SQLiteDatabase object and the old and new version numbers.  Hence we can figure out the best way to convert the database  from the old schema to the new one.  ------------Open/Close SQLite Connection-------  1. To open the database connection, call getWritableDatabase() method.  public DBManager open() throws SQLException {  dbHelper = new DatabaseHelper(context);  database = dbHelper.getWritableDatabase();  return this;  }  2. To close a database connection call close() method  public void close() {  dbHelper.close();  }  ---------- Insert Record SQLite table-----------  To insert a new record, call insert() method .  public void insert(String name, String desc) {  ContentValues contentValue = new ContentValues();  contentValue.put(DatabaseHelper.SUBJECT, name);  contentValue.put(DatabaseHelper.DESC, desc);  database.insert(DatabaseHelper.TABLE\_NAME, null, contentValue);  }  //Content Values creates an empty set of values using the given initial size.  ------------Update Record SQLite table---------  To update a record, call update() method.  public int update(long \_id, String name, String desc) {  ContentValues contentValues = new ContentValues();  contentValues.put(DatabaseHelper.SUBJECT, name);  contentValues.put(DatabaseHelper.DESC, desc);  int i = database.update(DatabaseHelper.TABLE\_NAME, contentValues, DatabaseHelper.\_ID + " = " + \_id, null);  return i;  }  ------SQLite Delete Record-------------  To delete call delete() method with the id of the record.  public void delete(long \_id) {  database.delete(DatabaseHelper.TABLE\_NAME, DatabaseHelper.\_ID + "=" + \_id, null);  }  ----------SQLite Cursor-------------  A Cursor represents the entire result set of the query.  Once the query is fetched a call to cursor.moveToFirst() is made.  Calling moveToFirst() allows us to test  whether the query returned an empty set,  and it moves the cursor to the first result when the set is not empty.  public Cursor fetch() {  String[] columns = new String[] { DatabaseHelper.\_ID, DatabaseHelper.SUBJECT, DatabaseHelper.DESC };  Cursor cursor = database.query(DatabaseHelper.TABLE\_NAME, columns, null, null, null, null, null);  if (cursor != null) {  cursor.moveToFirst();  }  return cursor;  }  -----------SQLite CursorAdapter------------  Another way to use a Cursor is to wrap it in a CursorAdapter.  Just as ArrayAdapter adapts arrays, CursorAdapter adapts Cursor objects,  making their data available to an AdapterView like a ListView.  ---------getReadabledatabase vs getWritabledatabase-------------  1.getReadableDatabase is the first to read and write to open the database,  but when the database disk is full, it will open failure, after failure,  the program will try to open the database as read-only,  if you can complete the required tasks,  the read-only database will be closed.  And it returns a read-write database object.  2.getWritableDatabase is also read-write to open the database,  but when the database disk is full, it will open the failure,  and then because the use of getwritabledatabase,  so open a read-only database has failed, then the program will be wrong.  ----------- SQLiteDatabase.Query vs RawQuery--------------  1. To issue a query to a database in Android we can use query() methods  located in SQLiteDatabas.  query() returns a cursor object that contains the result set for the query.  2. If the query() do not provide enough flexibility for a query  that an app needs to run, rawQuery() methods can be used instead.  Unlike the query(), the rawQuery() methods take two parameters-  a String parameter representing the query to run,  and a String[] to support query placeholder substitution.  --------------query from multiple tables--------------  1. We can use inner join().  SELECT  trackid,  name,  title  FROM  tracks  INNER JOIN albums ON albums.albumid = tracks.albumid;  In the tracks table, the AlbumId column is a foreign key.  And in the albums table, the AlbumId is the primary key.  2. We can use a union query.  SELECT FirstName, LastName, 'Employee' AS Type  FROM employees  UNION  SELECT FirstName, LastName, 'Customer'  FROM customers; | </end> |
| <hitle> | SQLite vs SQL | <chare> | 2 | <pext> | 1. SQLite:  It is a well built, embedded relational database management system  that is mostly used in mobile devices for data storage.  It is file based and does not support stored procedures.  2. SQL:  It is a structured query language.  It is server based and supports stored procedures. | </end> |
| <hitle> | SQLite transactions | <chare> | 2 | <pext> | A transaction is a unit of work that is performed against a database.  ACID defines the properties of one or more changes to the database.  ------------ Atomicity------------  1. Atomicity: ensures successful completion of all the transactions.  ------------ Consistency ------------  2. Consistency: ensures that the database changes state  on a successful transaction.  ------------ Isolation ------------  3. Isolation: allows transparent working of transactions,  independent of each other.  ------------ Stability ------------  4. Stability: ensures perseverance of the result  or effect of a committed transaction, in the event of a system failure. | </end> |
| <hitle> | SQLite pros | <chare> | 2 | <pext> | 1. To work, we do not need a separate service processor system.  2. SQLite comes with a zero configuration.  Thus, there is no need for configuration or administration.  3. One cross-platform disk can store an SQLite database.  4. SQLite is compact – less than 400 KiB  5. SQLite is standalone, which means no external dependencies.  6. And, it supports almost all OS types.  7. Also, you need to use ANSI-C to write it, and it provides an easy-to-use API | </end> |
| <hitle> | storage classes in SQLite | <chare> | 2 | <pext> | SQLite storage classes include  1. Null: NULL  2. Integer: represents an integer with a sign (1, 2, 3, etc.)  3. Real: IEEE 8-byte floating-point number  4. Text: a text string that is stored using the database encoding (UTF-8, UTF-16BE)  5. BLOB (Binary Large Object): Accurate storage of a block of data when we enter. | </end> |
| <hitle> | VARCHAR in SQLite | <chare> | 2 | <pext> | Generally, SQLite does not have a specific length for VARCHAR.  For example, you can declare VARCHAR (10), and SQLite will store  500 million characters there. It will keep all 500 characters intact. | </end> |
| <hitle> | whento use SQLite | <chare> | 2 | <pext> | SQLite can be used when:  1. Embedded applications: do not require an extension,  for example, mobile apps or games  2. Disk Access replacement: an application that requires  direct writing or reading files to disk.  3. Testing: when testing the logic of business applications  On the other hand, you need not use SQLite for:  1. Multi-user applications: In cases, where multiple clients must have  access and use the same database  2. Applications that require large volume records:  It allows you to use only one write operation at a time | </end> |
| <hitle> | recover SQLite database | <chare> | 2 | <pext> | To restore information, you can use a backup of the database file.  In case, you didn’t create a backup earlier, recovery is impossible.  SQLite uses SQLite SECURE DELETE,  which overwrites all deleted content with zeros. | </end> |
| <hitle> | EECN in SQLite | <chare> | 2 | <pext> | Any ECCN does not describe the source code of the main source  of the public domain SQLite.  Therefore, specify ECCN as EAR99.  But, if you add a new code or associate SQLite with the application,  it can change the EECN number. | </end> |
| <hitle> | SQLite indexes | <chare> | 2 | <pext> | SQLite indexes are special lookup tables used by the database search engine  to speed up the discovery of data.  In simple words, this is a pointer to the data in the table. | </end> |
| <hitle> | SQLite aggregate functions | <chare> | 2 | <pext> | The SQLite aggregate functions are the functions  where values of multiple rows and columns are grouped  as input and form a single value as output.  1. SQLite MIN function: It is used to select a minimum value for a column.  SELECT MIN(aggregate\_expression)  FROM tables  [WHERE conditions];  2. SQLite MAX function: It is used to select a maximum value for a column.  3. SQLite SUM function: It is used to select the total for a numeric column.  4. SQLite AVG function: It is used to select the average value for a table column.  5. SQLite COUNT function: It is used to count the number of rows in a database table.  6. SQLite UPPER function: It is used to convert a string into upper-case letters.  7. SQLite LOWER function: It is used to convert a string into lower-case letters.  8. SQLite LENGTH function: It is used get the length of a string. | </end> |
| <hitle> | SQLite\_Schema error code | <chare> | 2 | <pext> | SQLite\_Schema is an error code that is returned  when there is a change in the database.  The SQLite \_schema error code is indicated  when a prepared SQL statement is no longer valid  and cannot be executed.  These types of errors occur when using sqlite3 prepare ()  and sqlite3 step () interfaces to run SQL. | </end> |
| <hitle> | SQLite commands | <chare> | 2 | <pext> | DDL - Data Definition Language - commands  CREATE  ALTER  DROP  DML - Data Manipulation Language - commands  INSERT  UPDATE  DELETE  DQL - Data Query Language - commands  SELECT | </end> |
| <hitle> | create a database | <chare> | 3 | <pext> | In SQLite, sqlite3 command is used to create database.  Sqlite3 database\_name.db | </end> |
| <hitle> | create a table | <chare> | 3 | <pext> | CREATE TABLE statement is used to create a table in SQLite database.  You have to define the columns and data types of each column  while creating the table.  CREATE TABLE database\_name.table\_name(  column1 datatype PRIMARY KEY(one or more columns),  column2 datatype,  column3 datatype,  columnN datatype,  ); | </end> |
| <hitle> | drop a table | <chare> | 3 | <pext> | DROP TABLE command is used to delete  or permanently drop a table from SQLite database.  DROP TABLE table\_name; | </end> |
| <hitle> | AUTOINCREMENT field | <chare> | 3 | <pext> | For autoincrement, you have to declare a column of the table  to be INTEGER PRIMARY KEY, then whenever you insert a NULL  into that column of the table, the NULL is automatically converted  into an integer which is one greater than the largest value  of that column over all other rows in the table, or 1 if the table is empty. | </end> |
| <hitle> | insert data | <chare> | 3 | <pext> | INSERT INTO statement is used to insert data in a table in SQLite database.  There are two ways to insert data in table:  INSERT INTO TABLE\_NAME [(column1, column2, column3,...columnN)]  VALUES (value1, value2, value3,...valueN);  INSERT INTO TABLE\_NAME VALUES (value1,value2,value3,...valueN); | </end> |
| <hitle> | retrieve data from table | <chare> | 3 | <pext> | The SELECT command is used to retrieve data from SQLite table.  If you want to retrieve all columns from the table use SELECT \*  otherwise use the specific column's name separated by commas.  SELECT \* FROM table\_name;  Or  SELECT column1, column2, columnN FROM table\_name; | </end> |
| <hitle> | UPADTE query | <chare> | 3 | <pext> | The UPDATE query is used to modify the existing records in the SQLite table.  You have to use the WHERE clause to modify a specific row  otherwise all rows will be updated.  UPDATE table\_name  SET column1 = value1, column2 = value2...., columnN = valueN  WHERE [condition]; | </end> |
| <hitle> | delete the existing records | <chare> | 3 | <pext> | In SQLite, DELETE command is used to delete the existing records from a table.  You should use the WHERE clause to choose the specific row  otherwise all rows will be deleted.  DELETE FROM table\_name  WHERE [conditions....................]; | </end> |
| <hitle> | WHERE clause in CRUD statements | <chare> | 3 | <pext> | WHERE clause is used to refer a specific row  where the CRUD operation is executed.  Without using WHERE clause all the rows will be affected. | </end> |
| <hitle> | AND/OR operators | <chare> | 3 | <pext> | AND & OR operators are used with WHERE clause to combine  two or more than two conditions together.  SELECT column1, column2, columnN  FROM table\_name  WHERE [condition1] AND [condition2]...OR [conditionN]; | </end> |
| <hitle> | LIKE operator with WHERE | <chare> | 3 | <pext> | The LIKE operator is used to match text values against a pattern  using wildcards.  It uses two wildcards % and \_ with string for matching with input.  SELECT FROM table\_name  WHERE column LIKE 'XXXX%'  SELECT FROM table\_name  WHERE column LIKE 'XXXX\_' | </end> |
| <hitle> | LIMIT clause with SELECT query | <chare> | 3 | <pext> | LIMIT clause is used with SELECT statement  when we want a limited number of fetched records.  SELECT column1, column2, columnN  FROM table\_name  LIMIT [no of rows] | </end> |
| <hitle> | ORDER BY clause with SELECT | <chare> | 3 | <pext> | The ORDER BY clause is used to sort the fetched data  in a specific order either ascending or descending.  SELECT column-list  FROM table\_name  [WHERE condition]  [ORDER BY column1, column2, .. columnN] [ASC | DESC]; | </end> |
| <hitle> | GROUP BY clause | <chare> | 3 | <pext> | SQLite GROUP BY clause is used to collect the same elements into a group.  It is used with SELECT statement.  SELECT column-list  FROM table\_name  WHERE [ conditions ]  GROUP BY column1, column2....columnN  ORDER BY column1, column2....columnN | </end> |
| <hitle> | DISTINCT clause | <chare> | 3 | <pext> | The DISTINCT clause is always used with SELECT statement.  It is used to retrieve only unique records and restrict the duplicate entries.  It is used when the table has multiple duplicate records.  SELECT DISTINCT column1, column2,.....columnN  FROM table\_name  WHERE [condition] | </end> |
| <hitle> | UNION operator | <chare> | 3 | <pext> | SQLite UNION Operator is used to combine the result set of  two or more tables using SELECT statement.  Both the tables must have same number of fields in result table.  SELECT expression1, expression2, ... expression\_n  FROM tables  [WHERE conditions]  UNION  SELECT expression1, expression2, ... expression\_n  FROM tables  [WHERE conditions]; | </end> |
| <hitle> | UNION and UNION ALL operator | <chare> | 3 | <pext> | The UNION ALL operator is used to combine the result of  two or more tables using SELECT statement.  The unique difference between UNION  and UNION ALL operator is that UNION operator ignores the duplicate entries  while combining the results while UNION ALL doesn't ignore duplicate values.  SELECT expression1, expression2, ... expression\_n  FROM tables  [WHERE conditions]  UNION ALL  SELECT expression1, expression2, ... expression\_n  FROM tables  [WHERE conditions]; | </end> |
| <hitle> | types of JOINS | <chare> | 3 | <pext> | SQLite JOIN clause is used to combine two or more tables in a database.  It combines the table by using the common values of the both table.  SQLite INNER JOIN  SQLite OUTER JOIN  SQLite CROSS JOIN | </end> |
| <hitle> | INNER JOIN | <chare> | 3 | <pext> | SQLite INNER JOIN is simplest and most common join.  It combines all rows from both tables where the condition is satisfied.  SELECT ... FROM table1 [INNER] JOIN table2 ON conditional\_expression ... | </end> |
| <hitle> | OUTER JOIN | <chare> | 3 | <pext> | There are three types of OUTER JOINS:  Left outer join  Right outer join  Full outer join  But SQLite only supports left outer join.  The SQLite left outer join returns all rows from left table  and only those rows from the right table where the join condition is satisfied.  SELECT ... FROM table1 LEFT OUTER JOIN table2 ON conditional\_expression | </end> |
| <hitle> | CROSS JOIN | <chare> | 3 | <pext> | The SQLite Cross join is used to match every rows of the first table  with every rows of the second table.  If the first table contains x columns and second table contains y columns  then the resultant Cross join table will contain the x\*y columns.  SELECT ... FROM table1 CROSS JOIN table2 | </end> |
| <hitle> | SQLite date and time () | <chare> | 3 | <pext> | SQLite date and time () functions are used to retrieve  current date and time and also do calculations on the dates and time.  There are mainly 6 types of date and time () function in SQLite:  SQLite date() Function  SQLite datetime() Function  SQLite julianday() Function  SQLite now() Function  SQLite strftime() Function  SQLite time() Function | </end> |
| <hitle> | SQLite date and time () | <chare> | 3 | <pext> | SQLite date and time () functions are used to retrieve  current date and time and also do calculations on the dates and time.  There are mainly 6 types of date and time () function in SQLite:  SQLite date() Function  date(timestring, [ modifier1, modifier2, ... modifier\_n ] )  SQLite datetime() Function  datetime(timestring, [ modifier1, modifier2, ... modifier\_n ] )  SQLite julianday() Function  A Julian Day is the number of days since Nov 24, 4714 BC 12:00pm Greenwich time in the Gregorian calendar. So, the julianday() function is used to return number of days since Nov 24, 4714 BC 12:00pm.  julianday(timestring [, modifier1, modifier2, ... modifier\_n ] )  SQLite now() Function  date('now')  or  time('now')  SQLite strftime() Function  strftime(format, timestring [, modifier1, modifier2, ... modifier\_n ] )  SQLite time() Function  time(timestring, [ modifier1, modifier2, ... modifier\_n ] ) | </end> |
| <hitle> | SQLite date and time () | <chare> | 3 | <pext> | SQLite date and time () functions are used to retrieve  current date and time and also do calculations on the dates and time.  There are mainly 6 types of date and time () function in SQLite:  SQLite date() Function  SQLite datetime() Function  SQLite julianday() Function  SQLite now() Function  SQLite strftime() Function  SQLite time() Function | </end> |
| <hitle> | Jetpack Room Database | <chare> | 1 | <pext> | Room Database provides an abstraction layer over SQLite.  It reduces lots of overhead from developer by adding annotation  as it is annotation based library.  It segregated the database into multiple components  like Database, Dao, Entity.  ----------Components of Room Database-----------  Entity, Dao, Database  ----------Room Entity Data component-------  1. Entity : Entity is data class that is annotated by @Entity.  This class contains the details of table and tableName, columns of table.  -----------Room Dao component-----------  2. Dao : Data access objects are created  to keep all the CRUD operation into it.  It is an interface that annotated by @Dao  and keep all @Create, @Read, @Update, @Delete methods into it.  -----------Room Database component--------  3. Database : It is an abstract class where database details are added  to Room database and it return the database builder instance  for using room through out the app.  It require context, we should pass application Context  to keep instance of it global rather than scoping to activity.  ------------Room annotations-----------------  @Entity, @PrimaryKey, @ColumnInfo, @Ignore, @Embeded,  @ForeignKey,  -----------Room @Entity-------------  @Entity  data class Person()  Declaring the class as a Room Entity and the table name is class name.  @Entity(tableName = "people")  data class Person()  We can specify the table name.  @Entity(indices = arrayOf(Index(value = "first\_name", name = "name")))  data class Person()  If you want to index a column for faster searching of it,  we can add indices (you may need to add an @ before Index in Java).  ------------@PrimaryKey--------------  @PrimaryKey(autoGenerate = true)  val id: Long  It declares a field the primary key for the database.  We can allow the database to auto increment using autoGenerate = true.  -----------@ColumnInfo-------------  @ColumnInfo(name = "first\_name")  val firstName: String = ""  Lets say that you like underscores in your SQL columns,  but to put them in Java / Kotlin is not good.  So with this annotation you can specify the SQL name  and a nice Java / Kotlin name.  --------------@Ignore--------------  @Ignore  val bitmap: Bitmap  If we have something in our Pojo  that doesn’t need to go into the database, just add this annotation.  ------------@Embeded-------------  @Entity  data class Person(  ...  @Embedded  val address: Address  )  data class Address(val postcode: String,  val addressLine1: String)  We have a person, and that person has an address.  We don’t really want to store all the address fields directly in the person.  An Address object makes more sense.  At the same time we want to be able to query parts of the address.  If we embed an Address object, it will save as fields,  but map back to an Address object.  -----------@ForeignKey-------------  @Entity  data class Person(  @PrimaryKey(autoGenerate = true)  val uid: Long,  val firstName: String = "",  val lastName: String = "",  val email: String = ""  )  @Entity(foreignKeys = arrayOf(ForeignKey(entity = Person::class,  parentColumns = arrayOf("uid"),  childColumns = arrayOf("ownerId"),  onDelete = ForeignKey.CASCADE)))  data class Pet(  @PrimaryKey(autoGenerate = true) val id: Long,  val ownerId: Long,  val name: String  )  It specify foreign keys for entities,  like a person has a pet, when the person is deleted,  the pet is automatically deleted. | </end> |
| <hitle> | Room migration | <chare> | 2 | <pext> | In Room DB, if we change the database schema like adding any new table  or update the name, column of table or adding any new column  is considered to be migration process.  Room provides an abstraction layer by having an Migration class.  A Migration class defines the actions that should be performed  when migrating from one version to another.  For example remaining an table name is also part of Migration class.  --------Room migration crashes-----------  1. Forgot to add the database new version : app crashes.  When you just updating the existing table schema  and forgot to update new version then the result will be  an IllegalStateException.  2. Version increased, but no migration provided : app crashes.  when I had updated the database version from current by 1  and forgot to add migration to database and then it will result  get a crash IllegalStateException.  3. Version increased, fallback to destructive migration enabled  to database builder and it results -> database got cleared.  ------howto Room migration----------  Version increased, migration provided,  then data is kept safe and schema also updated.  1. First step is to update the Entity if you want to update existing table  then move towards creating a migration file.  val MIGRATION\_1\_2 = object : Migration(1, 2) {  override fun migrate(database: SupportSQLiteDatabase) {  database.execSQL("Alter TABLE Blog ADD COLUMN body TEXT ")  }  }  2. And then update database class by adding migration in builder.  @Database(entities = [Blog::class], version = 2, exportSchema = false)  abstract class AppDatabase : RoomDatabase() {  abstract fun getBlogDao(): BlogDao  companion object {  private var INSTANCE: AppDatabase? = null  fun getInstance(context: Context): AppDatabase? {  if (INSTANCE == null) {  INSTANCE = Room.databaseBuilder(  context.applicationContext, AppDatabase::class.java,  "blogDB"  ).addMigrations(MIGRATION\_1\_2)  .allowMainThreadQueries().build()  }  return INSTANCE  }  }  } | </end> |
| <hitle> | Room Type Converter | <chare> | 2 | <pext> | In general, primitives data types are available in SQLite/Room  but when you require some custom data types to store  in single database column then we have to think something out of the box.  Room provides a type converter which allows you to convert  a custom data type to available/ primitive data type and vice versa.  -----howto Room Type converter------  1. We have @TypeConverter that will be apply over an method  to create converter.  class Converter {  companion object {  @TypeConverter  @JvmStatic  fun fromBigDecimal(value: BigDecimal): String {  return value.toString()  }  @TypeConverter  @JvmStatic  fun toBigDecimal(value: String): BigDecimal {  return value.toBigDecimal()  }  }  }  2. Make entities of Room  @Entity(tableName = "product\_table")  data class Product(  @PrimaryKey var name: String,  var initialPrice: BigDecimal,  var price: BigDecimal  )  and have one price change data class  data class PriceChange(  var name: String,  var initialPrice: BigDecimal,  var price: BigDecimal,  var change: BigDecimal  )  3. Make Dao of Room  @Dao  interface ProductDao {  @Query("SELECT \* FROM product\_table")  fun allProducts(): List<Product>  @Query("SELECT \*, price-initialPrice as change FROM product\_table")  fun priceChange(): List<PriceChange>  @Insert(onConflict = OnConflictStrategy.REPLACE)  fun insert(product: Product)  @Insert(onConflict = OnConflictStrategy.REPLACE)  fun insert(products: List<Product>)  }  4. Use Dao  val pPriceChanges = appDatabase.productDao().priceChange()  val first = pPriceChanges[0]  first.name  first.initialPrice  first.price | </end> |
| <hitle> | relations Room multiple tables | <chare> | 2 | <pext> | --------One-to-one relations-------  We create two tables: Dog and Owner,  where the Dog table has a reference to the owner id,  or the Owner has a reference to a dog id.  @Entity  data class Dog(  @PrimaryKey val dogId: Long,  val dogOwnerId: Long,  val name: String,  val cuteness: Int,  val barkVolume: Int,  val breed: String  )  @Entity  data class Owner(@PrimaryKey val ownerId: Long, val name: String)  We would create a DogAndOwner data class.  data class DogAndOwner(  val owner: Owner,  val dog: Dog  )  To get a List<DogAndOwner> using Room,  we can use the @Relation annotation.  In our example, since Dog has the owner’s information, we add the @Relation annotation to the dog variable,: specifying that the ownerId column on the parent (i.e. the Owner entity) corresponds to the dogOwnerId:  data class DogAndOwner(  @Embedded val owner: Owner,  @Relation(  parentColumn = "ownerId",  entityColumn = "dogOwnerId"  )  val dog: Dog  )  Now we can implement Dao to do relation transation.  Room runs the two queries for us under the hood,  add the @Transaction annotation, to ensure that this happens atomically.  @Transaction  @Query("SELECT \* FROM Owner")  fun getDogsAndOwners(): List<DogAndOwner>  ----------One-to-many relations---------  An owner can have multiple dogs;  we’d have a one-to-many relation between Dog and Owner.  Now, to display the list of owners with their dogs,  we need to create a new data class.  To avoid running two separate queries,  we can define a one-to-many relation between Dog and Owner,  by annotating the List<Dog> with @Relation.  data class OwnerWithDogs(  @Embedded val owner: Owner,  @Relation(  parentColumn = "ownerId",  entityColumn = "dogOwnerId"  )  val dogs: List<Dog>  )  Now can do Dao to query OwnerWithDogs  @Transaction  @Query("SELECT \* FROM Owner")  fun getDogsAndOwners(): List<OwnerWithDogs>  --------Many-to-many relations--------  An owner can have multiple dogs, and that a dog can have multiple owners.  To model this schema, we need to create an associative table  (also known as cross-reference table) that keeps (dogId,ownerId) pairs:  @Entity(primaryKeys = ["dogId", "ownerId"])  data class DogOwnerCrossRef(  val dogId: Long,  val ownerId: Long  )  We need OwnerWithDogs data class and tell Room that  in order to get the Dogs,  it needs to use the DogOwnerCrossRef associate table.  We reference the table by using a Junction.  data class OwnerWithDogs(  @Embedded val owner: Owner,  @Relation(  parentColumn = "ownerId",  entityColumn = "dogId",  associateBy = Junction(DogOwnerCrossRef::class)  )  val dogs: List<Dog>  )  And then in Dao, we need to select from Owners and return the right data class.  @Transaction  @Query("SELECT \* FROM Owner")  fun getOwnersWithDogs(): List<OwnerWithDogs>  -----------Advanced relation------------  When using the @Relation annotation, Room infers the entity  to use from the type of the annotated property by default.  For example, we annotated a Dog (or a List<Dog>) with @Relation,  telling Room how to model the class and which columns to query.  If we want to return a different object, for example a Pup,  that is not an entity but contains some of the fields,  we can specify the entity to use in the @Relation annotation.  data class Pup(  val name: String,  val cuteness: Int = 11  )  data class OwnerWithPups(  @Embedded val owner: Owner,  @Relation(  parentColumn = "ownerId",  entity = Dog::class,  entityColumn = "dogOwnerId"  )  val dogs: List<Pup>  )  If we want to return only specific columns from an entity  you need to tell Room which these are by defining them  in the projection property of the @Relation.  For example, let’s say that we just want to get the names of all the dogs  in OwnerWithDogs data class.  Since we would need a List<String>,  we need to specify the column in the projection:  data class OwnerWithDogs(  @Embedded val owner: Owner,  @Relation(  parentColumn = "ownerId",  entity = Dog::class,  entityColumn = "dogOwnerId",  projection = ["name"]  )  val dogNames: List<String>  )  If we want to define a stricter relationship  between the dogOwnerId and ownerId,  independent of what kind of relation you’re creating,  we can use a ForeignKey constraint between the fields. | </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.  1-1. A Thread is a concurrent unit of execution. 2. Service exposes few functionalities to other applications  by calling Context.bindService().  2-1. For Thread, Google has brought in handlers and loopers into threads. 3. When an application is killed, service is not killed.  3-1. When an application is killed, the thread is killed. </ending> | </end> |
| <hitle> | Content provider | <chare> | 1 | <pext> | Content Providers is one of the android application important component  that serves the purpose of a relational database  to store the data of applications.  Content provider in the android system is like a “central repository”  in which data of the applications are stored,  and it allows other applications to securely access or supplies  and even modifies that data based on the requirements.  --------------Content URI-------------  Content URI is the key concept of content provider.  To query a content provider, we can specify the query string  in the form of URI.  URI: content://authority/data\_type/id  content:// →This is prefix of all the URI and it is mandatory part of URI.  authority →This part should be always unique for every content provider,  it specify the name of content provider. for example contacts, browser etc.  data\_type → This indicates the type of data provider by content provider.  id → it is a numeric value that is used to access particular record.  ----------Content provider Operations----------------  Operations of Content Provider are CRUD  CREATE : you can create a new data.  READ : you can access the available data.  UPDATE: you can modify the existing data.  DELETE: you can delete the data permanently from device as well.  -------restrict content provider operation-------------------  Define a permission with protectionLevel signature,  WRITE permission will restricted to only apps which are signed  with same private key  <permission android:name="com.yourapp.WRITE.PERMISSION"  android:protectionLevel="signature"  android:label="@string/permission\_label"  android:description="@string/permission\_desc">  </permission>  <permission android:name="com.yourapp.READ.PERMISSION"  android:label="@string/permission\_label"  android:description="@string/permission\_desc">  </permission>  Then in contentprovider tag use read and write permission tags.  You can either enforce read permission or you could altogether remove it  android:readPermission="com.yourapp.READ.PERMISSION"  android:writePermission="com.yourapp.WRITE.ERMISSION"  So only apps that are signed by same signature can use the content provider  private Collection<String> getCallingPackages() {  int caller = Binder.getCallingUid();  if (caller == 0) {  return null;  }  return Lists.newArrayList(mContext.getPackageManager().getPackagesForUid(caller));  }  And check if your packagename is present in this list. I think it is safe  -----------howto Content provider walkthrough------------  Android applications UI components like Activities/Fragments  use an object CursorLoader to send query requests to ContentResolver.  The ContentResolver object sends requests  like “create, read, update, and delete” to the ContentProvider as a client.  After receiving a request, ContentProvider process it  and returns the expected result.  -----------CursorLoader-----------  Android applications UI components like Activities/Fragments  use an object CursorLoader to send query requests to ContentResolver.  ----------- ContentResolver -----------  The ContentResolver object sends requests  like “create, read, update, and delete” to the ContentProvider as a client.  ContentProvider and ContentResolver are part of android.content package.  These two classes work together to provide robust,  secure data sharing model among applications.  ContentProvider exposes data stored in the SQLite database  to other application without telling them  the underlying implementation of your database. | </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> | Create job, JobService | <chare> | 2 | <pext> | To implement a Job, extend the JobService class and implement  the onStartJob and onStopJob.  If the job fails for some reason, return true from on the onStopJob  to restart the job.  The onStartJob is performed in the main thread,  if you start asynchronous processing in this method,  return true otherwise false. | </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> | 1. App-specific storage: store files that are meant for the app's use only,  either in dedicated directories within an internal storage volume  or different dedicated directories within external storage.  We can use the directories within internal storage to save  sensitive information that other apps shouldn't access.  2. Shared storage: store files that the app intends to share  with other apps, including media, documents, and other files.  3. Preferences: store private, primitive data in key-value pairs.  4. Databases: store structured data in a private database  using the Room persistence library. | </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, dialog, switch tasks | <chare> | 1 | <pext> | --------------------screen rotate, portrait to/from landscape ----------------------  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() onStop() onSaveInstanceState() onDestroy() onCreate() onStart() onRestoreInstanceState() onResume()  ------------------activity launched----------------  2. When the activity is launched (or show in foreground) onCreate() onStart() onResume()  ------------------back button pressed , app goes background-------------  3. When the app goes into background  (shutdown or home button is pressed) onPause() onStop() onDestroy()  -------------------home button pressed --------------------- 4. When we press home button,  onPause()  onStop()  ------------------open app from recent apps------------------  5. When we open the app from recent apps,  onRestart()  onStart()  onResume()  ------------------pressed recent button when activity is active--------------  6. When the activity is in active state and we press recent button,  onRestart()  onStart()  onResume()  -----------------open Alertdialog/BottomSheetDialog----------------  7. When you open a Alertdialog or a bottom sheet dialog on activity?  No lifecycle method will call when Alertdialog will shown on activity.  -----------------screen locked-------------  8. When screen got locked then what lifecycle method should call of Activity?  onPause()  onStop()  -----------------screen unlocked-------------  9. When your Screen got unlocked then what lifecycle method should call of Activity?  onRestart()  onStart()  onResume()  -------------device low battery enabled power saving mode-----------  10. When your device got low battery and you enabled power saving mode then lifecycle methods of activity is:  onPause() onStop() onSaveInstanceState() onDestroy() onCreate() onStart() onRestoreInstanceState() onResume()  --------------switch application by tasks---------------  11. When switch from current application to another application by tasks  onPause() onStop() onSaveInstanceState()  12. When switch back to your app from last app then lifecycle methods are  onRestart()  onStart()  onResume() | </end> |
| <hitle> | Activity lifecycle flow between activity | <chare> | 1 | <pext> | Let’s say activity A is a default activity and activity A starts another activity B.  1. When the app launch,  A’s onCreate()  A’s onStart()  A’s onResume()  2. When activity A starts another activity B,  A’s onPause()  B’s onCreate()  B’s onStart()  B’s onResume()  A’s onStrop() --- Until new activity is visible, the previous activity is not stopped.  2. When press back (go back to the previous activity A)  B’s onPause()  A’s onRestart()  A’s onStart()  A’s onResume()  B’s onStop()  B’s onDestory() --- activity B is not launcher activity, so it is destroyed. | </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> | 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> | ConstraintLayout 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 We cannot have a circular dependency in constraints.  ---------ConstraintLayout pros------------  1. ConstraintLayout provides the ability to completely design your UI  with the drag and drop feature provided by the Android Studio design editor.  2. It helps to improve the UI performance over other layouts.  3. We can control the group of widgets through a single line of code.  4. We can easily add animations to the UI components  which we used in our app.  --------ConstraintLayout const------------  1. When we use the Constraint Layout in our app,  the XML code generated becomes a bit difficult to understand.  2. In most of the cases, the result obtain will not be the same  as we got to see in the design editor.  3. Sometimes we have to create a separate layout file  for handling the UI for the landscape mode.  ---------ConstraintLayout vs LinearLayout-------------  1. By ConstraintLayout, we can position our UI components  in any sort of order whether it may be horizontal or vertical.  1-1. But in the case of Linear Layout,  we can only arrange our UI components either in a horizontal  or in a vertical manner.  2. Linear Layout provides usability with which we can equally divide  all the UI components in a horizontal or vertical manner  using weight sum.  2-1. But in Constraint Layout, we have to arrange  this UI component manually.  3. In Linear Layout the UI which is actually seen in the Design editor  of Android Studio will be the same as that we will get to see in the app.  3-1. But in the case of Constraint layout if the UI component  is not Constrained then the UI will not look the same as that of in design editor.  --------------ConstraintLayout vs RelativeLayout------------  1. In Constraint Layout, we have to add constraints to the view  on all four sides whereas in Relative Layout we can simply align  our UI component relative to its ID using the ids of UI components.  2. In Relative Layout, the UI which is actually seen in the Design editor  will be the same as that we will get to see in the app,  but in the case of Constraint layout if the UI component is not Constrained  then the UI will not look same as that of in design editor.  -------------Constraint Layout vs Grid Layout--------------  1. In Grid Layout the UI components are only arranged in Grid Manner  and we cannot arrange the UI components according to requirement,  whereas in Constraint Layout we can align UI components  according to the requirement.  2. In Grid Layout, the UI which is actually seen in the Design editor  will be the same as that we will get to see in the app,  but in the case of Constraint layout if the UI component is not Constrained  then the UI will not look same as that of in design editor.  --------------Guideline-----------------  Guidelines in Constraint Layout are invisible lines that are not visible  to users but help developers to design the layout easily  and constrain views to these guidelines,  so that design can be more clear and interactive.  But guidelines only work in Constraint Layout  as guidelines require something to be constrained to them.  There are two types of guidelines:  1. Horizontal Guidelines: These guidelines have a height of zero  and its width is equal to its parent Constraint Layout.  2. Vertical Guidelines: These guidelines have a width of zero  and its height is equal to its parent Constraint Layout.  -------------Barrier----------------  A Barrier references multiple widgets as input,  and creates a virtual guideline based on the most extreme widget  on the specified side.  For example, a left barrier will align to the left of all the referenced views.  a Barrier cannot work as a stand-alone component.  It needs to rely on one more View components  inside the respective ConstraintLayout.  -----------Bias--------------  The center positioning of widgets may not be desired every time  we try to create a layout.  Sometimes the widget may need to favor a particular side more than the other.  To achieve this we can use Bias constraint.  We can have horizontal and vertical bias of a widget.  app:layout\_constraintHorizontal\_bias="0.3"  This will keep the left side of the widget with a 30% bias  instead of the default 50%.  app:layout\_constraintVertical\_bias="0.2"  This will keep the top of the widget with a 20% bias  instead of the default 50%.  ------------Chain--------------  Chains allow to share space between the views within the chain  and control how the available space is divided between them.  It looks like weights in LinearLayout, but chains do more thigns than that.  There are 3 different kind of chains available.  1. Spread : The widgets will be equidistant from each other and the margin.  2. Spread inside : The widgets will be equidistant from each other and not the margin.  3. Packed : The widgets will be packed together closely. | </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- 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> | ListView | <chare> | 1 | <pext> | Android ListView is a view which groups several items and display them  in vertical scrollable list.  The list items are automatically inserted to the list using an Adapter  that pulls content from a source such as an array or database.  An adapter actually bridges between UI components  and the data source that fill data into UI Component.  Adapter holds the data and send the data to adapter view,  the view can takes the data from adapter view and shows the data  on different views like as spinner, list view, grid view etc.  The ListView and GridView are subclasses of AdapterView  and they can be populated by binding them to an Adapter,  which retrieves data from an external source  and creates a View that represents each data entry. | </end> |
| <hitle> | RecyclerView vs ListView | <chare> | 1 | <pext> | <starting>RecyclerView vs ListView</heading> ViewHolder pattern is mandatory in RecyclerView.  RecyclerView supports various layout managers  such as LinearLayout, GridLayoutManager, StaggeredLayoutManager  which is not available in ListView  RecyclerView is more memory and CPU efficient than ListView  For RecyclerView items, we can use item decorators which can used  to provide extra decoration to every element in RecyclerView  but Item decorators are not available for ListView  Swap list items for more options is possible for RecyclerView items  and not possible for ListView items  Implementing animations for each RecyclerView item  is more simple that providing Animation to ListView items  </ending> | </end> |
| <hitle> | RecyclerView | <chare> | 1 | <pext> | <starting>RecyclerView</heading> Recyclerview is part of of Android jetpack.  It is a UI component that allows the development  to create a scrolling list.  It is a ViewGroup used to render the data set  with the help of adapter and viewholder.  It has various view options like linear, grid and straggerd grid  using Viewholder pattern. </ending> | </end> |
| <hitle> | Internal work of RecyclerView | <chare> | 2 | <pext> | 1. ViewPort:  All visible view item on the screen of recyclerview is part  of Android ViewPort.  It varies when screen size is changed.  2. Scrap View:  The immediate view item of recyclerview that just gone away  either from top or bottom of screen.  Also the view that is just moved from view port.  This view is not longer going to display and moving into mechanism  to recycle the view with new data.  3. Recycle View:  After getting view into scrap stage it is going to recycler  for further new data.  4. Dirty View:  Your old view is recycled and ready to attach with new data.  5. Binding View:  New data is will set over the recycled item view  and going to visible into recyclerview.  6. Visible View:  Visible views are part of android viewport.  It may going to scrap view or it may just came from binding  with new data form recylerview.  RecyclerView Methods of RecyclerView.Adapter  onCreateViewHolder()  onBindViewHolder()  getItemCount() | </end> |
| <hitle> | Howto RecyclerView | <chare> | 2 | <pext> | 1. Implement xml for item view of recyclerview adapter.  2. Implement data class for View  data class MailDetails( val title: String, val body: String, ) : Serializable  3. Implement Adapter of recyclerview  class MailAdapter(private val context: Context, private val mailDetails: List<MailDetails>) :  RecyclerView.Adapter<MailAdapter.MailViewHolder>() {  private lateinit var binding: ItemViewBinding  override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): MailViewHolder {  val layoutInflater = LayoutInflater.from(parent.context)  binding = ItemViewBinding.inflate(layoutInflater, parent, false)  return MailViewHolder(binding.root)  }  override fun onBindViewHolder(holder: MailViewHolder, position: Int) {  holder.apply {  txtSubject.text = mailDetails[position].title  txtTime.text = mailDetails[position].time  itemView.setOnClickListener {  val intent = Intent(context, DetailActivity::class.java)  intent.putExtra(MAIL\_DATA, mailDetails[position] as Serializable)  context.startActivity(intent)  }  }  }  inner class MailViewHolder(val view: View) : RecyclerView.ViewHolder(view) {  val txtSubject: TextView = binding.txtSubject  val txtBody: TextView = binding.txtBody  }  companion object{  const val MAIL\_DATA = "mail data"  }  }  4. Implement the view for activity  <androidx.recyclerview.widget.RecyclerView  android:id="@+id/recycler\_View\_fruits"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  android:background="#F2F2F2" />  5. Implement RecyclerView in Activity  // create mailList  mailAdapter = MailAdapter(this, mailList)  binding.apply {  recyclerViewFruits.layoutManager = LinearLayoutManager(this@MainActivity)  recyclerViewFruits.adapter = mailAdapter  } | </end> |
| <hitle> | RecyclerView.ViewHolder | <chare> | 2 | <pext> | A ViewHolder describes an item view and metadata  about its place within the RecyclerView.  Adapter implementations should subclass ViewHolder  and add fields for caching potentially expensive findViewById results.  While LayoutParams belong to the LayoutManager,  ViewHolders belong to the adapter.  Adapters should feel free to use  their own custom ViewHolder implementations to store data  that makes binding view contents easier.  Individual item views will hold strong references to ViewHolder objects  and that RecyclerView instances may hold strong references  to extra off-screen item views for caching purposes. | </end> |
| <hitle> | ViewHolder Pattern | <chare> | 2 | <pext> | ListView Adapters do not require the use of ViewHolder  but RecyclerView require the use of ViewHolder  that is used to store the reference of View’s. | </end> |
| <hitle> | onCreateViewHolder() | <chare> | 2 | <pext> | onCreateViewHolder() is called right when the adapter is created  and is used to initialize ViewHolders.  One of the interesting things and one of the more powerful features  is that the getItemViewType(int position) method allows us  to return the actual layout id that the Android framework saves for us  as a layout resource.  Returning the layout id resource in this method is very powerful  because it allows us to use the resource id with the LayoutInflater  in the onCreateViewHolder() method and then return the ViewHolder. | </end> |
| <hitle> | onBindViewHolder() | <chare> | 2 | <pext> | This method is called for each ViewHolder to bind it to the adapter.  This is where we will pass our data to our ViewHolder. | </end> |
| <hitle> | LayoutManager | <chare> | 2 | <pext> | A LayoutManager is responsible for measuring  and positioning item views within a RecyclerView  as well as determining the policy for when to recycle item views  that are no longer visible to the user.  By changing the LayoutManager a RecyclerView can be used  to implement a standard vertically scrolling list, a uniform grid,  staggered grids, horizontally scrolling collections and more. | </end> |
| <hitle> | LayoutManager types | <chare> | 2 | <pext> | 1. LinearLayoutManger to support Vertical/Horizontal List.  2. GridLayoutManager to support Uniform Grid List.  3. StaggeredGridLayoutManager to support Staggered Grid List. | </end> |
| <hitle> | How many viewholders created | <chare> | 2 | <pext> | If there is a recycled view available that MATCHES to the new one  that's becoming visible, then it will re-use.  Otherwise it'll create the new object for the new view.  It uses getViewType () to find that. | </end> |
| <hitle> | Grid format RecyclerView | <chare> | 2 | <pext> | We can use Grid Layout Manager.  In Grid Layout manager we can align our recycler in the form of a grid.  We have to mention the number of columns that are to be displayed  in the Grid of Recycler View. | </end> |
| <hitle> | Horizontal scroll RecyclerView | <chare> | 2 | <pext> | We can use Linear Layout Manager.  In linear layout manager, we can align our recycler view in a horizontal  or vertical scrolling manner by specifying its orientation  as vertical or horizontal.  LinearLayoutManager layoutManager  = new LinearLayoutManager(requireContext(), LinearLayoutManager.HORIZONTAL, false); | </end> |
| <hitle> | Multiple various view type RecyclerView adapter | <chare> | 2 | <pext> | We can Just implement getItemViewType(), and take care  of the viewType parameter in onCreateViewHolder().  public class MyAdapter extends RecyclerView.Adapter<RecyclerView.ViewHolder> {  class ViewHolder0 extends RecyclerView.ViewHolder {  ...  public ViewHolder0(View itemView){  ...  }  }  class ViewHolder2 extends RecyclerView.ViewHolder {  ...  public ViewHolder2(View itemView){  ...  }  @Override  public int getItemViewType(int position) {  // Just as an example, return 0 or 2 depending on position  // Note that unlike in ListView adapters, types don't have to be contiguous  return position % 2 \* 2;  }  @Override  public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup parent, int viewType) {  switch (viewType) {  case 0: return new ViewHolder0(...);  case 2: return new ViewHolder2(...);  ...  }  }  @Override  public void onBindViewHolder(final RecyclerView.ViewHolder holder, final int position) {  switch (holder.getItemViewType()) {  case 0:  ViewHolder0 viewHolder0 = (ViewHolder0)holder;  ...  break;  case 2:  ViewHolder2 viewHolder2 = (ViewHolder2)holder;  ...  break;  }  }  } | </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> | 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> | 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> | 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> | 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 testing | <chare> | 1 | <pext> | Unit testing is a testing methodology where we test individual functionality  of application code, to ensure that code written must be work as expected. | </end> |
| <hitle> | Unit testing Triangle concept | <chare> | 2 | <pext> | Testing triangle is a way to segregate the unit testing for whole app  by their requirement like UI’s should be tested  using Espresso, Robolectric test run.  Another classes like ViewModels, repositories, helpers, presenters, utils  should be run via Junit/Mockito/Mockk local testing.  While classes that are using android processes like Room Database  that is require to pass context to instantiate will go under  Instrumentation/Integration testing.  ------------Unit Testing--------------  This is also called as local testing.  Junit and Mockito library can be used to test classes like ViewModel  if using MVVM, Presenter if using MVP,  repository if using repository pattern, Helpers/ Utils etc.  These tests run using local JVM so the speed of running the test case  is faster among other testing ways.  These testing require dependencies of library Junit and Mockito.  These test placed under (test) dir.  ---------------------Integration testing--------------------  This is a way of testing the classes which is using context  or any android process so that require an android process  like emulator to run the test.  Example like room DB require context so it can be tested  under integration testing and these tests placed under (android test) dir.  These testing require android emulator to run the test cases  so it is slower than local testing but faster than UI testing.  These testing require dependencies of library Junit and Mockito.  ---------------UI Test--------------------------  This is a kind of testing where we can test our UI’s of Android app  like activities and fragments.  We need Espresso library to run these test.  UI tests run over emulator or real device so these are slowest  in terms of speed of testing.  These tests placed under (android test) dir. | </end> |
| <hitle> | Unit testing pros | <chare> | 2 | <pext> | 1. Unit Testing helps in finding bugs early stage rather than getting bugs  after deployment of app and making bad impact  on impression of application.  2. Unit Testing reduces the testing efforts because all the scenario’s  is covered and then there is very less time required  to check functionality.  3. Unit testing simplifies refactoring and provides documentation  because each scenario must be on a documented form  to make all team on same page.  4. Unit testing insures the quality of code. | </end> |
| <hitle> | Unit tests | <chare> | 2 | <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> | 2 | <pext> | <starting>Robolectric vs Mockito</heading> Mockito is a library that allows 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> | mock object | <chare> | 2 | <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> | Types of Test Doubles | <chare> | 2 | <pext> | The five types of test doubles  1. Dummy.  2. Stub.  3. Spy.  4. Mock.  5. Fake.  ----------------Dummy--------------  A dummy is a test double that throws an exception  when a method on it is invoked.  ----------------Stub----------------  A stub is a test double that returns pre-configured,  hard-coded data when its methods are called.  ----------------Spy----------------  A spy is a test double that records the invocations of its methods,  along with any parameters passed in, so that you can find out afterward  whether and how its methods were called.  ----------------Mock----------------  A mock is a combination of stub and spy, like a test double  that allows you to control the data it returns  while also recording the calls made to it so that  you can check that it was called with the correct parameters.  ----------------Fake----------------  A fake is a test double that has the same business-logic behavior  as the real component it’s doubling, but using a simplified implementation.  Because it has behavior, fakes must themselves be tested,  and ideally, tested using the same test cases  as those used for the real component. | </end> |
| <hitle> | Mockito | <chare> | 2 | <pext> | Mockito is a Java-based mocking framework used for unit testing  of Java application.  Mockito plays a crucial role in developing testable applications.  It internally uses the Java Reflection API to generate mock objects  for a specific interface.  Mock objects are referred to as the dummy or proxy objects  used for actual implementations. | </end> |
| <hitle> | Create mock objects with the Mockito | <chare> | 3 | <pext> | 1. Using the @ExtendWith(MockitoExtension.class) extension  for JUnit 5 in combination with the @Mock annotation on fields.  2. Using the static mock() method.  3. Using the @Mock annotation.  If we use the @Mock annotation,  we must trigger the initialization of the annotated fields.  The MockitoExtension does this by calling the static method  MockitoAnnotations.initMocks(this). | </end> |
| <hitle> | Return values with Mockito | <chare> | 3 | <pext> | 1. Using when().thenReturn() and when().thenThrow()--------  Mocks can return different values depending on arguments  passed into a method.  The when(…​.).thenReturn(…​.) method chain is used  to specify a return value for a method call with pre-defined parameters.  2. doReturn when and doThrow when  The doReturn(…​).when(…​) method configuration can be used  to configure the reply of a mocked method call. | </end> |
| <hitle> | Mockito Verify | <chare> | 3 | <pext> | Mockito Verify methods are used to check that certain behavior happened.  We can use Mockito verify methods at the end of the testing method code  to make sure that specified methods are called.  -----------------Mockito verify()-----------------  Mockito verify() method can be used to test  number of method invocations too.  We can test exact number of times, at least once, at least, at most number  of invocation times for a mocked method.  -----------------Mockito verifyNoMoreInteractions()-----------------  We can use verifyNoMoreInteractions() after all the verify() method calls  to make sure everything is verified.  If any method verification is still left, it will fail and provide proper message.  -----------------Mockito verifyZeroInteractions()-----------------  verifyZeroInteractions() behavior is same as verifyNoMoreInteractions() method.  We can use it after all the verify() method calls  to make sure everything is verified.  If any method verification is still left, it will fail and provide proper message.  -----------------Mockito inOrder()-----------------  We can use inOrder() method to verify the order of method invocation.  We can skip a method invocation but the methods being verified  must be in the same order.  -----------------Mockito only()-----------------  If we want to verify that only one method is being called,  then we can use only() with verify method. | </end> |
| <hitle> | ArgumentCaptor capture | <chare> | 3 | <pext> | The ArgumentCaptor class allows to access the arguments  of method calls during the verification.  This allows to capture these arguments of method calls  and to use them for tests. | </end> |
| <hitle> | @InjectMocks dependency injection Mockito | <chare> | 3 | <pext> | We have the @InjectMocks annotation which tries  to do constructor, method or field dependency injection  of mock objects in to other type. | </end> |
| <hitle> | Answers for complex mocks | <chare> | 3 | <pext> | thenReturn returns a predefined value every time.  With an Answer object we can calculate a response based on the arguments  given to your stubbed method. | </end> |
| <hitle> | MockK | <chare> | 2 | <pext> | To mock DataSource, mockk<DataSource>(relaxed = true)  coVerify { }, for suspend functions  verify { }, for regular functions  coEvery { } for stubbing coroutines  every { } for regular functions. | </end> |
| <hitle> | unit testing tools | <chare> | 2 | <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> | coverage with unit tests | <chare> | 2 | <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> | 2 | <pext> | TDD stands for Test Driven Development,  In this software development technique, we create the test cases first  and then write the code underlying those test cases.  Although TDD is a development technique, it can also be used  for automation testing development.  --------howto TDD-------------  1. Write a test case. Based on the requirements,  write an automated test case.  2. Run the test cases and watch test fails.  3. Develop the code for that test cases.  If the test case fails, then, write the code to make  that test-case work as expected.  4. Run the test cases again.  5. Refactor the code. This is an optional step.  However, it’s important to refactor the code to make it  more readable and reusable.  6. Repeat step 1 to step 5 cycle for other test cases. | </end> |
| <hitle> | Behavior Driven Development (BDD) | <chare> | 2 | <pext> | BDD stands for Behavior Driven Development,  BDD is an extension to TDD where instead of writing the test cases,  we start by writing a behavior.  ---------------howto BDD---------------  1. Write the behavior of the application.  The behavior of an application is written in simple English  like language by the product owner or the business analysts.  2. Write the automated scripts.  This simple English like language is then converted into programming tests.  3. Implement the functional code.  The functional code underlying the behavior is then implemented.  4. Check if the behavior is successful.  Run the behavior and see if it is successful.  If successful, move to the next behavior otherwise fix the errors  in the functional code to achieve the application behavior.  5. Refactor or organize code.  Refactor or organize the code to make it more readable and re-usable.  6. Repeat the steps 1–5 for new behavior. | </end> |
| <hitle> | TDD vs BDD, Test Driven Development vs Behavior Driven Development | <chare> | 2 | <pext> | 1. TDD is software development process in which the requirements  are turned into specific test cases and then software is improved  to pass the new tests  1-1. BDD is Agile software development process that encourages  collaboration between developers, QA and non-technical  or business participants in the software project.  2. TDD stans for Test Driven Development  but BDD stands for Behavior Driven Development.  3. TDD involves designing high level scenarios.  3-1. BDD involves designing low level scenarios. | </end> |
| <hitle> | unit testing cons | <chare> | 2 | <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> | 2 | <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> | 2 | <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> | 2 | <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> | 2 | <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> | 2 | <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> | Unit test annotations | <chare> | 2 | <pext> | -----------------@Test---------------  This is the annotation that will make any method to be run under Junit.  Once we add @Test annotation we will be able to test  any existing functionality.  -----------------@Before-----------------  This is the annotation that will be used for pre-requisite  for any unit test methods.  This method will be executed before executing any @test method  of any test class.  It is recommended to use @Before method when we have scenario’s  like a particular class instance is going to reuse in many test methods  then initialize it in @Before method will be a best choice.  -----------------@After-----------------  This is the annotation that will be used after all the test method  of a test class.  @After annotation method can be used for tear down feature  like shutdown feature.  This is also very important to keep your variables reset  once unit testing done.  -----------------@Rule-----------------  This annotation means that this is a JUnit4 test rule.  JUnit4 test rules are run before and after every test method.  -----------------@RunWith-----------------  JUnit will invoke the class  it references to run the tests in that class  instead of the runner built into JUnit.  It is also recommended annotating the test  with the @RunWith(AndroidJUnit4.class) annotation.  AndroidJUnit4 extends JUnit4,  so if we use pure Junit4 syntax and ActivityTestRule it is not required.  But we need it, if we want to run,  like Espresso tests with ActivityTestRule and JUnit4  -----------------@Mock--------------  In case we have multiple dependencies that must be mocked,  it gets cumbersome to create each and every mock manually  with the variant shown above.  So, we can also create mocks by using the @Mock annotation. | </end> |
| <hitle> | Espresso | <chare> | 2 | <pext> | Espresso is a testing framework for Android to make it easy to write  reliable user interface tests.  Espresso automatically synchronizes test actions with the user interface  of the application.  The framework also ensures that the activity is started before the tests run.  It also let the test wait until all observed background activities  have finished.  --------------Espresso three components-------------  1. ViewMatchers - allows to find view in the current view hierarchy  2. ViewActions - allows to perform actions on the views  3. ViewAssertions - allows to assert state of a view  onView(ViewMatcher).perform(ViewAction).check(ViewAssertion)  ------------- Espresso functions---------------------  1. View Matchers  1-1. User Properties  withId(), withText(), withTagKey(), withTagValue(),  hasContentDescription(), withContentDescription()  withHint(), withSpinnerText(),  hasLinks(), hasEllipsizedText(), hasMultilineTest()  1-2 UI Prperties  isDisplayed(), isCompletelyDisplayed(), isEnabled(), hasFocus(),  isClickable(), isChecked(), isNotChecked(),  withEffectiveVisibility(), isSelected()  1-3. Object Matcher  allOf(Matchers), anyOf(M…), isOf(), not(),  endsWith(String), startsWith(S…), instanceOf(Class)  1-4. Hierarchy  withParent(Matcher), withChild(Matcher)  hasDecendant(M…), isDescendantOfA(M…)  hasSibling(M..), isRoot()  1-5. Input  supportsInputMethos(), hasIMEAction()  1-6. Class  isAssignableForm(), withClassName()  1-7. Root Matchers  isFocusable(), isTouchable(), isDialog(), withDecorview(), isPlatformPopup()  2. ViewActions  2-1. Click/Press  Click(), doubleClick(), longClick(), pressBack(), pressIMEActionButton(),  pressKey(), pressMenuKey(), closeSoftKeyboard(), openLink()  2-2. Gestures  scrollTo(), swipeLeft(), swipeRight(), swipeUp(), swipeDown()  2-3. Text  clearText(), typeText(String), typeTextIntoFocusedView(S…), replaceText(String)  3. ViewAssertions  Matches(Matcher), doesNotExist(), selectedDescendantsMatch()  3-1. LayoutAssertions  noEllipseizedText(Matcher), noMultilineButtons(), noOverlaps([Matcher])  3-2. PositonAssertions  isLeftOf(Matcher), isRightOf(M..) isLeftAlignedWith(M..), isRightAlignedWith(M…)  isAbove(M…), isBelow(M…), isBottomAlignedWith(M…), isTopAlignedWith(M…)  4. IntentMatchers  Intended(IntentMatcher)  Intending(IntentMatcher).respondWith(ActivityResult)  4-1. Intent  hasAction(), hasCategories(), hasData(), hasComponent(), hasExtras(),  hasExtras(Matcher), hasExtraWithKey(), hasType(), hasPackage(), toPackage(),  hasFlag(), hasFlags(), isInternal()  4-2. URI  hasHost(), hasParamWithName(), hasPath(), hasParamWithValue(), hasScheme(),  hasSchemeSpecificPart(),  4-3. ComponentName  hasClassName(), hasPackageName(), hasShortClassName(), hasPackageName()  4-4. Bundle  hasEntry(), hasKey(), hasValue()  -----------Espresso Pros-------------------  1. Test scripts once created can be reused;  this makes testing easily scalable.  2. Improves performance as efficient test scripts deliver fast  and accurate results.  3. Simplified testing by letting QAs refer to previous builds/test results  and proceed further.  4. Machines are less prone to errors than humans,  which leads to a higher likelihood of accurate results  even with a large number of tests.  -----------Espresso Cons----------------  1. Restricted support for programming languages.  The scripts are mainly written in Java and Kotlin  and don’t support other languages.  2. Only compatible with Android UI testing.  ------------Espresso ListView, RecyclerView Test--------------  1. ListView  ListView is an AdapterView. AdapterViews present a problem  when doing UI testing.  Since an AdapterView doesn't load all of it's items upfront  (only as the user scrolls through the items),  AdapterView's don't work well with onView(...) since the particular view  might not be part of the view hierarchy yet.  Fortunately, Espresso provides an onData(...) entry point  that makes sure to load the AdapterView item  before performing any operations on it.  onData(allOf(is(instanceOf(String.class)), is("two")))  // Use Hamcrest matchers to match item  .inAdapterView(withId(R.id.lvItems))  // Specify the explicit id of the ListView  .perform(click());  // Standard ViewAction  Alternately, if we know the position of the particular item,  we can directly specify the position  instead of using a data Matcher to find it.  onData(anything())  // We are using the position so don't need to specify a data matcher  .inAdapterView(withId(R.id.lvItems))  // Specify the explicit id of the ListView  .atPosition(1)  // Explicitly specify the adapter item to use  .perform(click());  // Standard ViewAction  2. RecyclerView  Unfortunately, RecyclerView is not an AdapterView  so we can't use onData(...) for a RecyclerView,  but Espresso does support RecyclerView.  We can use RecyclerViewActions to interact with our RecyclerView.  // Click on the RecyclerView item at position 2  onView(withId(R.id.rvItems))  .perform(RecyclerViewActions.actionOnItemAtPosition(2, click()));  ------------Espresso mock out the intent------------  // CameraActivityInstrumentationTest.java  public class CameraActivityInstrumentationTest {  // IntentsTestRule is an extension of ActivityTestRule. IntentsTestRule sets up Espresso-Intents  // before each Test is executed to allow stubbing and validation of intents.  @Rule  public IntentsTestRule<CameraActivity> intentsRule = new IntentsTestRule<>(CameraActivity.class);  @Test  public void validateCameraScenario() {  // Create a bitmap we can use for our simulated camera image  Bitmap icon = BitmapFactory.decodeResource(  InstrumentationRegistry.getTargetContext().getResources(),  R.mipmap.ic\_launcher);  // Build a result to return from the Camera app  Intent resultData = new Intent();  resultData.putExtra("data", icon);  Instrumentation.ActivityResult result = new Instrumentation.ActivityResult(Activity.RESULT\_OK, resultData);  // Stub out the Camera. When an intent is sent to the Camera, this tells Espresso to respond  // with the ActivityResult we just created  intending(toPackage("com.android.camera2")).respondWith(result);  // Now that we have the stub in place, click on the button in our app that launches into the Camera  onView(withId(R.id.btnTakePicture)).perform(click());  // We can also validate that an intent resolving to the "camera" activity has been sent out by our app  intended(toPackage("com.android.camera2"));  // ... additional test steps and validation ...  }  } | </end> |
| <hitle> | Espresso | <chare> | 3 | <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> | Automate Testing vs Manual Testing | <chare> | 2 | <pext> | 1. Accuracy  1-1. Manual Testing shows lower accuracy due to the higher possibility  of human errors.  1-2. Automation Testing shows a higher accuracy due to  computer-based testing eliminating the chances of errors.  2. Testing at Scale  2-1. Manual Testing needs time when testing is needed at a large scale.  2-2. Automation Testing easily performs testing at a large scale  with the utmost efficiency.  3. Turnaround time  3-1. Manual Testing takes more time to complete a testing cycle,  and thus the turnaround time is higher.  3-2. Automation Testing completes a testing cycle within record time;  thus, the turnaround time is much lower.  4. Cost Efficiency  4-1. Manual Testing requires more cost as it involves  hiring expert professionals.  4-2. Automation Testing saves costs as once the software infrastructure  is integrated; it works for a long time.  5. User Experience  5-1. Manual Testing ensures a high-end User Experience  to the software’s end user, as it requires human observation  and cognitive abilities.  5-2. Automation Testing cannot guarantee a good User Experience  since the machine lacks human observation and cognitive abilities.  6. Areas of Specialization  6-1. To exhibit the best results, manual Testing should be used  to perform Exploratory, Usability, and Ad-hoc Testing.  6-2. Automation Testing should be used to perform Regression Testing,  Load Testing, Performance Testing, and Repeated Execution for best results.  7. User Skills  7-1. Users must be able to mimic user behavior and build test plans  to cover all the scenarios.  7-2. Users must be highly skilled at programming and scripting  to build test cases and automate as many scenarios as possible. | </end> |
| <hitle> | Junit vs Espresso | <chare> | 2 | <pext> | 1. Juint is faster because it run on local JVM but Espresso is slower  because it runs on actual android device or virtual machine.  2. Junit is used to test logic, classes, functions  but Espresso is used to test user interfaction and application flow. | </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> | 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> | clean architectural principles | <chare> | 1 | <pext> | 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. | </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> | LifecycleOwner | <chare> | 1 | <pext> | LifecycleOwner is an interface implemented by the AppCompatActivity  and Fragment classes. You can subscribe other components  to owner objects which implement this interface, to observe changes  to the lifecycle of the owner. | </end> |
| <hitle> | Android version vs API level | <chare> | 1 | <pext> | 1. Android 13 티라미수  version 13.0 > API level 33  2. Android 12 스노우 콘  version 12.1 > API level 32  version 12 > API level 31  3. Android 11레드 벨벳 케이크  version 11 > API level 30  4. Android 10퀸 케이크  version 10 > API level 29  5. Pie 파이  version 9 > API level 28  6. Oreo 오레오  version 8.1 > API level 27  version 8.0 > API level 26  7. Nougat 누가  version 7.1 > API level 25  version 7.0 > API level 24  8. Marshmallow 마시멜로  version 6.0.1, 6.0 > API level 23  9. Lollipop 롤리팝  version 5.1.0, 5.1.1 > API level 22  version 5.0, 5.0.1, 5.0.2 > API level 21  10. KitKat 킷캣  version 4.4, 4.4.1, 4.4.2, 4.4.3, 4.4.4 > API level 19  11. Jelly Bean 젤리빈  version 4.3, 4.3.1 > API level 18  version 4.2, 4.2.1, 4.2.2 > API level 17  version 4.1, 4.1.1, 4.1.2 > API level 16  12. Ice Cream Sandwich 아이스크림 샌드위치  version 4.0.3, 4.0.4 > API level 15, NDK 8  version 4.0, 4.0.1, 4.0.2 > API level 14, NDK 7  13. Honeycomb 허니콤  version 3.2.X > API level 13  version 3.1 > API level 12, NDK 6  version 3.0 > API level 11  14. Gingerbread 진저브레드  version 2.3.3 - 2.3.7 > API level 10  version 2.3 - 2.3.2 > API level 9, NDK 5  15. Froyo(Frozen yogurt) 프로요  version 2.2 - 2.2.3 > API level 8, NDK 4  16. Éclair 에클레어  version 2.1 > API level 7, NDK 3  version 2.0.1 > API level 6  version 2.0 > API level 5  17. Donut 도넛  version 1.6 > API level 4, NDK 2  18. Cupcake 컵케이크  version 1.5 > API level 3, NDK 1  19. Petit Four 쁘띠 푸르  version 1.1 > API level 2  20. Android 1.0 애플파이  version 1.0 > API level 1 | </end> |
| <hitle> | Menu | <chare> | 1 | <pext> | Menu is an important UI component which is used  to provide various options, features behind an hidden UI functionality  around the application.  There are three types of Menu in android applications:  1. Option Menu  Android Options Menu is a primary collection of menu items  in an android application and useful for actions over toolbar  for few direct features like settings/logout etc.  We have various action mode like visible always, visible ifRoom available etc.  2. PopUp Menu  Android Popup Menu displays a list of items in a vertical list  which presents to the view that invoked the menu and useful  to provide an overflow of actions that related to specific content.  It need an view as anchor to open popUp menu.  3. Context Menu  Android Context Menu is a floating menu that only appears  when the user clicks for a long time on an element and useful for elements  that affect the selected content or context frame.  For example over any view like a listview of contacts have options  over menu to call/sms etc. | </end> |
| <hitle> | Database | <chare> | 1 | <pext> | Database is an organized collection of related data where the data is stored  and organized to serve some specific purpose.  ---------Database transactions ACID-----------  Atomicity, Consistency, Isolation, Durability  ---------Atomicity----------  Each statement in a transaction(CRUD) is treated as a single Unit.  --------Consistency----------  Ensures that transactions only make changes to table in predefined way.  --------Isolation-------------  Concurrent transactions should not interfere with one another.  -----------Durability-----------  Ensures that changes to your data made successfully executed transaction will be saved, no matter after it there is a system failure.  ------------CRUD-----------  C-Create-creating a table/database  R-Read-reading a tables or data into it  U-Update-updating data on table  D-Delete-delete a row or table items. | </end> |
| <hitle> | DBMS | <chare> | 2 | <pext> | DBMS stands for Database Management System.  It is a collection of application programs which allow the user to organize,  restore and retrieve information about data efficiently  and as effectively as possible. | </end> |
| <hitle> | RDBMS | <chare> | 2 | <pext> | Relational Database Management System(RDBMS) is based on  a relational model of data that is stored in databases in separate tables  and they are related to the use of a common column.  Data can be accessed easily from the relational database  using Structured Query Language (SQL). | </end> |
| <hitle> | DBMS pros | <chare> | 2 | <pext> | 1. Data is stored in a structured way  and hence redundancy is controlled.  2. Validates the data entered and provide restrictions  on unauthorized access to the database.  3. Provides backup and recovery of the data when required.  4. It provides multiple user interfaces. | </end> |
| <hitle> | Data Redundancy | <chare> | 2 | <pext> | Duplication of data in the database is known as data redundancy.  As a result of data redundancy, duplicated data is present  at multiple locations, hence it leads to wastage of the storage space  and the integrity of the database is destroyed. | </end> |
| <hitle> | relationships in Database | <chare> | 2 | <pext> | 1. One-to-one: One table has a relationship with another table  having the similar kind of column.  Each primary key relates to only one or no record in the related table.  2. One-to-many: One table has a relationship with another table  that has primary and foreign key relations.  The primary key table contains only one record  that relates to none, one or many records in the related table.  3. Many-to-many: Each record in both the tables can relate  to many numbers of records in another table. | </end> |
| <hitle> | Normalization and De-Normalization | <chare> | 2 | <pext> | 1. Normalization is the process of removing redundant data  from the database by splitting the table in a well-defined manner  in order to maintain data integrity.  This process saves much of the storage space.  2. De-normalization is the process of adding up redundant data  on the table in order to speed up the complex queries  and thus achieve better performance. | </end> |
| <hitle> | types of Normalization | <chare> | 2 | <pext> | 1. First Normal Form (1NF):  A relation is said to be in 1NF only when all the entities of the table  contain unique or atomic values.  2. Second Normal Form (2NF):  A relation is said to be in 2NF only if it is in 1NF and all the non-key attribute  of the table is fully dependent on the primary key.  3. Third Normal Form (3NF):  A relation is said to be in 3NF only if it is in 2NF  and every non-key attribute of the table is not transitively dependent  on the primary key. | </end> |
| <hitle> | What is SQL | <chare> | 2 | <pext> | Structured Query language, SQL is an ANSI  (American National Standard Institute)  standard programming language that is designed specifically  for storing and managing the data  in the relational database management system  using all kinds of data operations. | </end> |
| <hitle> | SQL statements | <chare> | 2 | <pext> | SQL statements are basically divided into three categories,  DDL, DML, and DCL. | </end> |
| <hitle> | Data Definition Language (DDL) command | <chare> | 2 | <pext> | Data Definition Language (DDL) commands are used to define the structure  that holds the data.  These commands are auto-committed  like changes done by the DDL commands on the database  are saved permanently.  CREATE to create a new table or database.  ALTER for alteration.  TRUNCATE to delete data from the table.  DROP to drop a table.  RENAME to rename a table. | </end> |
| <hitle> | Data Manipulation Language (DML) command | <chare> | 2 | <pext> | Data Manipulation Language (DML) commands are used  to manipulate the data of the database.  These commands are not auto-committed and can be rolled back.  INSERT to insert a new row.  UPDATE to update an existing row.  DELETE to delete a row.  MERGE for merging two rows or two tables. | </end> |
| <hitle> | Data Control Language (DCL) command | <chare> | 2 | <pext> | Data Control Language (DCL) commands are used to control  the visibility of the data in the database like revoke access permission  for using data in the database.  COMMIT to permanently save.  ROLLBACK to undo the change.  SAVEPOINT to save temporarily. | </end> |
| <hitle> | DELETE vs TRUNCATE vs DROP | <chare> | 2 | <pext> | After the execution of ‘DELETE’ operation,  COMMIT and ROLLBACK statements can be performed  to retrieve the lost data.  After the execution of ‘TRUNCATE’ operation,  COMMIT, and ROLLBACK statements cannot be performed  to retrieve the lost data.  ‘DROP’ command is used to drop the table or key  like the primary key/foreign key. | </end> |
| <hitle> | B-Trees | <chare> | 2 | <pext> | B-Tree represents the data structure in the form  of a tree for external memory that reads and writes large blocks of data.  It is commonly used in databases and file systems  where all the insertions, deletions, sorting, etc.,  are done in logarithmic time. | </end> |
| <hitle> | Stored procedures | <chare> | 2 | <pext> | A Stored procedure is a collection of pre-compiled SQL Queries,  which when executed denotes a program taking input,  process and gives the output. | </end> |
| <hitle> | Database Triggers | <chare> | 2 | <pext> | A set of commands that automatically get executed when an event  like Before Insert, After Insert, On Update, On Delete of row occurs  in a table is called as Database trigger. | </end> |
| <hitle> | Unique key | <chare> | 2 | <pext> | A Unique key is the same as the primary key whose every row data  is uniquely identified with a difference of null value.  Unique key allows one value as a NULL value. | </end> |
| <hitle> | Primary Key, Composite Key, Foreign Key | <chare> | 2 | <pext> | --------Primary Key---------  1. Primary Key is that column of the table  whose every row data is uniquely identified.  Every row in the table must have a primary key  and no two rows can have the same primary key.  Primary key value can never be null nor can it be modified or updated.  ------------ Composite Key -------------------  2. Composite Key is a form of the candidate key  where a set of columns will uniquely identify every row in the table.  ----------------foreign key----------  3. A foreign key is a column or group of columns  in a relational database table that provides a link  between data in two tables.  It acts as a cross-reference between tables because it references  the primary key of another table, thereby establishing a link between them.  ----------Primary Key vs Foreign Key-------------  While a primary key may exist on its own,  a foreign key must always reference to a primary key somewhere.  The original table containing the primary key is the parent/referenced table.  This key can be referenced by multiple foreign keys from other tables,  known as “child” tables.  For any column acting as a foreign key,  a corresponding value should exist in the linked table. | </end> |
| <hitle> | Data Dictionary | <chare> | 2 | <pext> | Data dictionary is a set of information  describing the content and structure of the tables and database objects.  The job of the information stored in the data dictionary is to control,  manipulate and access the relationship between database elements. | </end> |
| <hitle> | Database partitioning | <chare> | 2 | <pext> | Database partitioning is the process of partitioning tables,  indexes into smaller pieces in order to manage  and access the data at a finer level.  Database partitioning:  1. Improves query performance and manageability.  2. Simplifies common administration tasks.  3. Acts as a key tool for building systems  with extremely high availability requirements.  4. Allows accessing a large part of a single partition. | </end> |
| <hitle> | Checkpoint | <chare> | 2 | <pext> | Checkpoint declares a point before which all the logs  are stored permanently in the storage disk and is the inconsistent state.  In the case of crashes, the amount of work and time is saved  as the system can restart from the checkpoint. | </end> |
| <hitle> | Phantom deadlock | <chare> | 2 | <pext> | Phantom deadlock detection is the condition where the deadlock  does not actually exist but due to a delay  in propagating local information,  deadlock detection algorithms identify the deadlocks. | </end> |
| <hitle> | Fragmentation | <chare> | 2 | <pext> | Fragmentation is a feature that controls the logical data units,  also known as fragments that are stored at different sites  of a distributed database system. | </end> |
| <hitle> | Cluster vs Non-cluster index | <chare> | 2 | <pext> | Clustered index alters the table and re-order the way in which  the records are stored in the table.  Data retrieval is made faster by using the clustered index.  A Non-clustered index does alter the records that are stored in the table  but creates a completely different object within the table. | </end> |
| <hitle> | query performance | <chare> | 2 | <pext> | Index hunting help in improving query performance by:  1. Using a query optimizer to coordinate queries with the workload.  2. Observing the performance and effect of index and query distribution. | </end> |
| <hitle> | Index hunting | <chare> | 2 | <pext> | Index hunting is the process of boosting the collection of indexes  which helps in improving the query performance as well as  the speed of the database. | </end> |
| <hitle> | Data Warehousing | <chare> | 2 | <pext> | The storage as well as access to data, that is being derived  from the transactions and other sources, from a central location  in order to perform the analysis is called Data Warehousing. | </end> |
| <hitle> | Database Lock | <chare> | 2 | <pext> | Database lock basically signifies the transaction  about the current status of the data item like,  whether that data is being used by other transactions  or not at the present point of time.  There are two types of Database lock: Shared Lock and Exclusive Lock. | </end> |
| <hitle> | Cursor and types | <chare> | 2 | <pext> | Cursor is a temporary work area that stores the data,  as well as the result set, occurred after manipulation of data retrieved.  A cursor can hold only one row at a time.  The 2 types of Cursor are:  1. Implicit cursors are declared automatically when DML statements  like INSERT, UPDATE, DELETE is executed.  2. Explicit cursors have to be declared when SELECT statements  that are returning more than one row are executed. | </end> |
| <hitle> | VDL and SDL | <chare> | 2 | <pext> | VDL is View Definition Language which represents user views  and their mapping to the conceptual schema.  SDL is Storage Definition Language which specifies  the mapping between two schemas. | </end> |
| <hitle> | Attribute and Relations | <chare> | 2 | <pext> | Attribute is described as the properties or characteristics of an entity.  Relation is a two-dimensional table containing  a number of rows and columns  where every row represents a record of the relation.  Rows are also known as ‘Tuples’ and columns are known as ‘Attributes’. | </end> |
| <hitle> | Weak Entity set | <chare> | 2 | <pext> | Weak Entity set is the one whose primary key comprises its partial key  as well as the primary key of its parent entity.  This is the case because the entity set may not have sufficient attributes  to form a primary key. | </end> |
| <hitle> | Entity, Entity type, Entity set | <chare> | 2 | <pext> | Entity can be anything, be it a place, class or object  which has an independent existence in the real world.  Entity Type represents a set of entities that have similar attributes.  Entity Set in the database represents a collection of entities  having a particular entity type. | </end> |
| <hitle> | E-R model | <chare> | 2 | <pext> | E-R model is an Entity-Relationship model which defines the conceptual view of the database.  The E-R model basically shows the real-world entities and their association/relations. Entities here represent the set of attributes in the database. | </end> |
| <hitle> | functional dependency fully dependent | <chare> | 2 | <pext> | To fulfill the criteria of fully functional dependency,  the relation must meet the requirement of functional dependency.  A functional dependency ‘A’ and ‘B’ are said to be fully functional dependent  when removal of any attribute say ‘X’ from ‘A’ means  the dependency does not hold anymore. | </end> |
| <hitle> | Functional dependency | <chare> | 2 | <pext> | A relation is said to be in functional dependency  when one attribute uniquely defines another attribute.  For Example, R is a Relation, X and Y are two attributes.  T1 and T2 are two tuples. Then,  T1[X]=T2[X] and T1[Y]=T2[Y]  Means, the value of component X uniquely define the value of component Y.  Also, X->Y means Y is functionally dependent on X. | </end> |
| <hitle> | Database pros/cons | <chare> | 2 | <pext> | Pros:  1. As there is no physical location where the data in the view is stored,  it generates output without wasting resources.  2. Data access is restricted as it does not allow commands like insertion,  updation, and deletion.  Cons:  1. The view becomes irrelevant if we drop a table related to that view.  2. Much memory space is occupied when the view is created for large tables. | </end> |
| <hitle> | View and Data Independence | <chare> | 2 | <pext> | View is a virtual table that does not have its data on its own  rather the data is defined from one or more underlying base tables.  Views account for logical data independence as the growth  and restructuring of base tables are not reflected in views. | </end> |
| <hitle> | Data Independence | <chare> | 2 | <pext> | Data Independence refers to the ability to modify the schema definition  in one level in such a way that it does not affect the schema definition  in the next higher level.  The 2 types of Data Independence are:  1. Physical Data Independence:  It modifies the schema at the physical level  without affecting the schema at the conceptual level.  2. Logical Data Independence:  It modifies the schema at the conceptual level without affecting  or causing changes in the schema at the view level. | </end> |
| <hitle> | Record, Field, Table | <chare> | 2 | <pext> | Record: Record is a collection of values or fields of a specific entity. For Example, An employee, Salary account, etc.  Field: A field refers to an area within a record that is reserved for specific data. For Example, Employee ID.  Table: Table is the collection of records of specific types. For Example, the Employee table is a collection of records related to all the employees. | </end> |
| <hitle> | SQL pros | <chare> | 2 | <pext> | 1. Simple SQL queries can be used to retrieve a large amount of data  from the database very quickly and efficiently.  2. SQL is easy to learn and almost every DBMS supports SQL.  3. It is easier to manage the database using SQL  as no large amount of coding is required. | </end> |
| <hitle> | DDL interpreter | <chare> | 2 | <pext> | DDL Interpreter interprets the DDL statements  and records the generated statements in the table containing metadata. | </end> |
| <hitle> | DML Compiler | <chare> | 2 | <pext> | DML compiler translates DML statements in a query language  into a low-level instruction and the generated instruction  can be understood by Query Evaluation Engine. | </end> |
| <hitle> | Google Map | <chare> | 1 | <pext> | Android provides facility to integrate Google map in our application.  Google map displays your current location, navigate location direction,  search location etc.  We can also customize Google map according to our requirement.  -----------Types of Google Maps--------------  googleMap.setMapType(GoogleMap.MAP\_TYPE\_NORMAL);  1. Normal: This type of map displays typical road map, natural features  like river and some features build by humans.  2. Hybrid: This type of map displays satellite photograph data  with typical road maps. It also displays road and feature labels.  3. Satellite: Satellite type displays satellite photograph data,  but doesn't display road and feature labels.  4. Terrain: This type displays photographic data.  This includes colors, contour lines and labels and perspective shading.  5. None: This type displays an empty grid with no tiles loaded.  -----------Methods of Google map------------  1. addCircle(CircleOptions options)  This method add circle to map.  2. addPolygon(PolygonOptions options)  This method add polygon to map.  3. addTileOverlay(TileOverlayOptions options)  This method add tile overlay to the map.  4. animateCamera(CameraUpdate update)  This method moves the map according to the update with an animation.  5. clear()  This method removes everything from the map.  6. getMyLocation()  This method returns the currently displayed user location.  7. moveCamera(CameraUpdate update)  This method reposition the camera according to the instructions  defined in the update.  8. setTrafficEnabled(boolean enabled)  This method set the traffic layer on or off.  9. snapshot(GoogleMap.SnapshotReadyCallback callback)  This method takes a snapshot of the map.  10. stopAnimation()  This method stops the camera animation if there is any progress.  ---------------- onMapReady()---------------  The onMapReady() callback is triggered when the map is ready.  We use this function to add contents to the map.  ------------------Marker------------------  We can create markers on the map via the Marker class.  -----------------Current Location-----------------  Most Android devices allow to determine the current geo location.  This can be done via a GPS (Global Positioning System) module,  via cell tower triangulation and via wifi networks.  Google Play provides the fused location provider  to retrieve the device’s last known location. | </end> |
| <hitle> | Notification | <chare> | 1 | <pext> | Notifications are one of the fastest and easiest ways  to add more interactivity and engagement to your application.  Android Notification provides short, timely information  about the action happened in the application, even it is not running.  The notification displays the icon, title and some amount of the content text.  -----------------Notification properties------------------  The properties of Android notification are set  using NotificationCompat.Builder object.  1. Small icon  2. App Name  3. Timestamp  4. Large icon  5. Title  6. Text  ---------------Types of Notifications in Android--------------------  1. Heads-up Notification  2. Status bar Notification  3. Notification Drawer Notification  4. Lock screen Notification  --------------howto Notification--------------------  1. create Notification channel  2. Make a Pending Intent  3. Implement Notification builder.  ------------- Notification channel----------------  Channels allow users to select which notifications  their applications can show them.  If an application uses only one channel for all of its notifications,  the user would not be able to select which notifications they want to see  and if they blocked one channel, they would no longer get notifications  from the application. | </end> |
| <hitle> | Custom View | <chare> | 1 | <pext> | If none of the prebuilt widgets or layouts meet our needs,  we can create our own View subclass.  If we only need to make small adjustments to an existing widget or layout,  we can subclass the widget or layout and override its methods.  ----------Howto custom view primary control/modify-------------  1. Drawing –  Control the rendering of the view on screen visually  by overriding the onDraw method.  2. Interaction –  Control the ways the user can interact with the view  with the onTouchEvent and gestures.  3. Measurement –  Control the content dimensions of the view on screen  by overriding the onMeasure method.  4. Attributes - Defining custom XML attributes for your view  and using them to control behavior with TypedArray  5. Persistence - Storing and restoring state on configuration changes  to avoid losing the state with onSaveInstanceState and onRestoreInstanceState  -----------Howto custom view---------------  ---------1. Define Custom View Class-----------  To create our own custom view, we start by defining a class  which extends from View and implements the required constructor.  public class ShapeSelectorView extends View {  // We must provide a constructor that takes a Context and an AttributeSet.  // This constructor allows the UI to create and edit an instance of your view.  public ShapeSelectorView(Context context, AttributeSet attrs) {  super(context, attrs);  }  }  ---------2. Add View to Layout---------  We add the custom view to activity layout  <com.codepath.example.customviewdemo.ShapeSelectorView/>  ---------3. Define Custom Attributes---------  Well-written custom views can be configured and styled via XML attributes.  We might want to let the user select the properties of the view.  We might want the view to be configurable in XML.  <com.codepath.example.customviewdemo.ShapeSelectorView  app:shapeColor="#7f0000"  app:displayShapeName="true"  ... />  In order to be able to define shapeColor and displayShapeName,  we need to define these as attributes within res/values/attrs.xml:  <?xml version="1.0" encoding="utf-8"?>  <resources>  <declare-styleable name="ShapeSelectorView">  <attr name="shapeColor" format="color" />  <attr name="displayShapeName" format="boolean" />  </declare-styleable>  </resources>  ---------3. Apply Custom Attributes---------  To extract the custom attributes,  we can use a TypedArray and the obtainStyledAttributes on the AttributeSet.  public class ShapeSelectorView extends View {  private int shapeColor;  private boolean displayShapeName;  public ShapeSelectorView(Context context, AttributeSet attrs) {  super(context, attrs);  setupAttributes(attrs);  }  private void setupAttributes(AttributeSet attrs) {  // Obtain a typed array of attributes  TypedArray a = getContext().getTheme().obtainStyledAttributes(attrs, R.styleable.ShapeSelectorView, 0, 0);  // Extract custom attributes into member variables  try {  shapeColor = a.getColor(R.styleable.ShapeSelectorView\_shapeColor, Color.BLACK);  displayShapeName = a.getBoolean(R.styleable.ShapeSelectorView\_displayShapeName, false);  } finally {  // TypedArray objects are shared and must be recycled.  a.recycle();  }  }  }  ---------4. Add Property Methods---------  We need to expose property methods to allow us  to get and set the important properties after a view has been created.  When the view properties are changed and might require a redraw,  We need to be sure to call invalidate() and requestLayout()  to update the appearance.  public class ShapeSelectorView extends View {  // ...  public boolean isDisplayingShapeName() {  return displayShapeName;  }  public void setDisplayingShapeName(boolean state) {  this.displayShapeName = state;  invalidate();  requestLayout();  }  public int getShapeColor() {  return shapeColor;  }  public void setShapeColor(int color) {  this.shapeColor = color;  invalidate();  requestLayout();  }  }  --------5. Drawing Custom View--------------  We draw something that we want using the properties defined already.  All view drawing happens within the onDraw method  using the Canvas object to paint onto the view.  public class ShapeSelectorView extends View {  @Override  protected void onDraw(Canvas canvas) {  super.onDraw(canvas);  canvas.drawRect(0, 0, shapeWidth, shapeHeight, paintShape);  if (displayShapeName) {  canvas.drawText("Square", 0 + textXOffset, shapeHeight + textXOffset, paintShape);  }  }  }  -------6. Calculating the Dimensions----------------  In order to understand the width and height of a view  that is being custom drawn, we need to define the onMeasure method  which determines the width and height of the view based on it's contents.  onMeasure method must call the setMeasuredDimension.  Widths and heights are discussed using the MeasureSpec  which encapsulates all the different types of constraints  imposed by the parent layout for a view.  The helper method resolveSizeAndState() returns an appropriate value  by comparing the view's desired size to the spec passed into the method.  public class ShapeSelectorView extends View {  @Override  protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {  // Defines the extra padding for the shape name text  int textPadding = 10;  int contentWidth = shapeWidth;    // Resolve the width based on our minimum and the measure spec  int minw = contentWidth + getPaddingLeft() + getPaddingRight();  int w = resolveSizeAndState(minw, widthMeasureSpec, 0);    // Ask for a height that would let the view get as big as it can  int minh = shapeHeight + getPaddingBottom() + getPaddingTop();  if (displayShapeName) {  minh += textYOffset + textPadding;  }  int h = resolveSizeAndState(minh, heightMeasureSpec, 0);    // Calling this method determines the measured width and height  // Retrieve with getMeasuredWidth or getMeasuredHeight methods later  setMeasuredDimension(w, h);  }  }  -------7. Handle Events in custom view-------  We want to do something when the view is clicked.  Then we can setup a touch handler using the onTouchEvent method.  public class ShapeSelectorView extends View {  // Change the currentShapeIndex whenever the shape is clicked  @Override  public boolean onTouchEvent(MotionEvent event) {  boolean result = super.onTouchEvent(event);  if (event.getAction() == MotionEvent.ACTION\_DOWN) {  currentShapeIndex = (currentShapeIndex ++) % shapeValues.length;  postInvalidate();  return true;  }  return result;  }  }  -----------8. Use custom view------------  We can now use our custom view as normal view in activities or anywhere.  public class MainActivity extends Activity {  @Override  protected void onCreate(Bundle savedInstanceState) {  super.onCreate(savedInstanceState);  setContentView(R.layout.activity\_main);  shapeSelector = (ShapeSelectorView) findViewById(R.id.shapeSelector);  btnSelect = (Button) findViewById(R.id.btnSelect);  btnSelect.setOnClickListener(new OnClickListener() {  @Override  public void onClick(View v) {  Toast.makeText(MainActivity.this, "You selected: " +  shapeSelector.getSelectedShape(), Toast.LENGTH\_LONG).show();  }  });  }  } | </end> |
| <hitle> | SharedPreferences | <chare> | 1 | <pext> | SharedPreferences is the way that we can store and retrieve  small amounts of primitive data as key/value pairs to a file  on the device storage in an XML file inside the app.  The sharedPreferences class provides APIs for reading, writing,  and managing this data.  ---------Whereto store SharedPreferences--------------  SharedPreferences are stored in an xml file in the app data folder.  /data/data/YOUR\_PACKAGE\_NAME/shared\_prefs/YOUR\_PREFS\_NAME.xml  or the default preferences at:  /data/data/YOUR\_PACKAGE\_NAME/shared\_prefs/YOUR\_PACKAGE\_NAME\_preferences.xml  ---------SharedPreferences mode--------------  1. MODE\_PUBLIC will make the file public which could be accessible  by other applications on the device.  2. MODE\_PRIVATE keeps the files private and secures the user’s data.  3. MODE\_APPEND is used while reading the data from the SP file.  ---------Shared Preferences method/Operation -------------  1. getBoolean(key, default), getFloat(), getInt(), getLong(), getString(),  getStringSet()  This method is used to retrieve value from the preferences.  2. Editor - putBoolean(key, value), putFloat(), putInt(), putLong(),  putString(), getStringSet()  It will save value in a preference editor  3. apply() – Editor  apply() changes the in-memory SharedPreferences object immediately  but writes the updates to disk asynchronously.  4. commit() - Editor  We can use commit() to write the data to disk synchronously.  But because commit() is synchronous, we should avoid calling it  from the main thread because it could pause your UI rendering.  5. clear() - Editor  It will remove all values from the editor.  4. remove(String key)  It will remove the value whose key has been passed as a parameter.  5. contains(String key)  This method is used to check whether the preferences  contain a preference.  6. getAll()  This method is used to retrieve all values from the preferences.  7. edit() - SharedPreferences  This method is used to create a new Editor for these preferences,  through which we can make modifications to the data in the preferences  and atomically commit those changes back  to the SharedPreferences object.  8. listeners – SharedPreferences  8-1. registerOnSharedPreferenceChangeListener  This is used to register a callback to be invoked  when a change happens to a preference.  8-2. unregisterOnShared PreferenceChangeListener  unregisterOnSharedPreferencechangeListener  This method is used to unregister a previous callback.  --------- Read/Write SharedPreferences------------  1. Write  sharedPreferences = getSharedPreferences(MainActivity.LOGIN\_FILE, MODE\_PRIVATE)  editor = sharedPreferences.edit()  editor.apply {  putString(EMAIL, "xxx")  apply()  }  2. Read  sharedPreferences = getSharedPreferences(MainActivity.LOGIN\_FILE, MODE\_PRIVATE)  sharedPreferences.getString(EMAIL, "")  -------------commit() vs apply()----------------  Data is stored in SharedPreferences in the form of a key-value pair  like HashMap.  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. | </end> |
| <hitle> | Android Security | <chare> | 1 | <pext> | Android has built-in security features  that significantly reduce the frequency  and impact of application security issues.  ----------------Android Security Features------------  1. The Android application sandbox, which isolates your app data  and code execution from other apps.  2. An application framework with robust implementations  of common security functionality such as cryptography, permissions,  and secure interprocess communication (IPC).  3. Technologies like address space layout randomization (ASLR),  no-execute (NX), ProPolice, safe\_iop, OpenBSD dlmalloc and calloc,  and Linux mmap\_min\_addr to mitigate risks associated with  common memory management errors.  4. An encrypted file system that can be enabled to protect data  on lost or stolen devices.  5. User-granted permissions to restrict access  to system features and user data.  6. Application-defined permissions to control application  data on a per-app basis. | </end> |
| <hitle> | Key management | <chare> | 2 | <pext> | The Security library uses a two-part system for key management.  1. A keyset that contains one or more keys to encrypt a file  or shared preferences data.  The keyset itself is stored in SharedPreferences.  2. A primary or master key that encrypts all keysets.  This key is stored using the Android keystore system. | </end> |
| <hitle> | Secure EncryptedFile | <chare> | 2 | <pext> | EncryptedFile Provides custom implementations  of FileInputStream and FileOutputStream,  granting your app more secure streaming read and write operations.  To provide secure read and write operations from file streams,  the Security library uses the Streaming Authenticated Encryption  with Associated Data (AEAD) primitive.  -----------Read/Write EncryptedFile-------------  1. Read  val keyGenParameterSpec = MasterKeys.AES256\_GCM\_SPEC  val mainKeyAlias = MasterKeys.getOrCreate(keyGenParameterSpec)  val fileToRead = "my\_sensitive\_data.txt"  val encryptedFile = EncryptedFile.Builder(  File(DIRECTORY, fileToRead),  applicationContext,  mainKeyAlias,  EncryptedFile.FileEncryptionScheme.AES256\_GCM\_HKDF\_4KB  ).build()  val inputStream = encryptedFile.openFileInput()  val byteArrayOutputStream = ByteArrayOutputStream()  var nextByte: Int = inputStream.read()  while (nextByte != -1) {  byteArrayOutputStream.write(nextByte)  nextByte = inputStream.read()  }  val plaintext: ByteArray = byteArrayOutputStream.toByteArray()  2. Write  val keyGenParameterSpec = MasterKeys.AES256\_GCM\_SPEC  val mainKeyAlias = MasterKeys.getOrCreate(keyGenParameterSpec)  val fileToWrite = "my\_sensitive\_data.txt"  val encryptedFile = EncryptedFile.Builder(  File(DIRECTORY, fileToWrite),  applicationContext,  mainKeyAlias,  EncryptedFile.FileEncryptionScheme.AES256\_GCM\_HKDF\_4KB  ).build()  val fileContent = "MY SUPER-SECRET INFORMATION"  .toByteArray(StandardCharsets.UTF\_8)  encryptedFile.openFileOutput().apply {  write(fileContent)  flush()  close()  } | </end> |
| <hitle> | Secure EncrypedSharedPreferences | <chare> | 1 | <pext> | Shared preferences are not secure as we can simply view the data stored  within the shared preferences and can easily access data within that file.  To make the data stored in shared preferences secure  we use encrypted share preferences which are more secure  and the data stored in them is encrypted.  Anyone cannot read the data within the file easily.  ------------AEAD Algorithms Secure/Encrypt--------------  EncrypedSharedPreferences uses AEAD Algorithms.  Authenticated Encryption with Associated Data (AEAD)  -------------Read/Write EncrypedSharedPreference-------------  1. Read  val keyGenerateParameters = MasterKeys.AES256\_GCM\_SPEC  val mainKeyAlias = MasterKeys.getOrCreate(keyGenerateParameters)  encryptedSharedPreferences = EncryptedSharedPreferences.create(  MainActivity.ENCRYPTED\_LOGIN\_FILE,  mainKeyAlias,  this,  EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256\_SIV,  EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256\_GCM  )  encryptedSharedPreferences.getString(EMAIL, "")  2. Write  val keyGenerateParameters = MasterKeys.AES256\_GCM\_SPEC  val mainKeyAlias = MasterKeys.getOrCreate(keyGenerateParameters)  encryptedSharedPreferences = EncryptedSharedPreferences.create(  MainActivity.ENCRYPTED\_LOGIN\_FILE,  mainKeyAlias,  this,  EncryptedSharedPreferences.PrefKeyEncryptionScheme.AES256\_SIV,  EncryptedSharedPreferences.PrefValueEncryptionScheme.AES256\_GCM  )  editor = encryptedSharedPreferences.edit()  editor.apply {  putString(EMAIL, email)  putString(PASSWORD, password)  } | </end> |
| <hitle> | XXXXXXXXXXXXX | <chare> | 2 | <pext> | X  Xx  X  x | </end> |