|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | Activity | <chare> | 1 | <pext> | 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. | </end> |
| <hitle> | Activity in manifest | <chare> | 2 | <pext> | For the app to be able to use activities, we must declare the activities,  and certain of their attributes, in the manifest. | </end> |
| <hitle> | Declare activity | <chare> | 3 | <pext> | To declare activity, in the manifest file and we need to add  an <activity> element as a child of the <application> element.  <manifest ... >  <application ... >  <activity android:name=".ExampleActivity" />  ...  </application ... >  ...  </manifest >  The only required attribute for this element is android:name,  which specifies the class name of the activity.  We can also add attributes that define activity characteristics  such as label, icon, or UI theme.  Note: After you publish the app, wqe should not change activity names.  If we do, we might break some functionality, such as app shortcuts. | </end> |
| <hitle> | Declare permission | <chare> | 3 | <pext> | We can use the manifest's <activity> tag to control  which apps can start a particular activity.  A parent activity cannot launch a child activity  unless both activities have the same permissions in their manifest.  If we declare a <uses-permission> element for a parent activity,  each child activity must have a matching <uses-permission> element.  <manifest>  <activity android:name="...."  android:permission=”com.google.socialapp.permission.SHARE\_POST”  />  (\*For example, if the app wants to use a hypothetical app named SocialApp  to share a post on social media, SocialApp itself must define the permission that an app calling it must have. Then, to be allowed to call SocialApp, the app must match the permission set in SocialApp's manifest.\*)  <manifest>  <uses-permission android:name="com.google.socialapp.permission.SHARE\_POST" />  </manifest> | </end> |
| <hitle> | activity lifecycle | <chare> | 2 | <pext> | An activity goes through a number of states.  We can use a series of callbacks to handle transitions between states.  These callbacks are:  1. onCreate()  It is called when activity is created.  What we write in onCreate() callback’s implementation,  We should initialize the essential components of the activity.  For example, the app should create views and bind data to lists here.  Most importantly, this is where we must call setContentView()  to define the layout for the activity's user interface.  2. onStart()  It is called if the activity is becoming visible to the user.  As onCreate() exits, the activity enters the Started state,  and the activity becomes visible to the user.  This callback contains what amounts to the activity’s final preparations  for coming to the foreground and becoming interactive.  3. onResume()  It is called when the activity will start an interaction with the user.  The system invokes this callback just before the activity  starts interacting with the user.  At this point, the activity is at the top of the activity stack,  and captures all user input.  4. onPause()  It is called when the activity is moving to the background  but hasn’t been killed yet.  The system calls onPause() when the activity loses focus  and enters a Paused state.  This state occurs when,  for example, the user taps the Back or Recents button.  When the system calls onPause() for the activity,  it technically means the activity is still partially visible,  but most often is an indication that the user is leaving the activity,  and the activity will soon enter the Stopped or Resumed state.  An activity in the Paused state may continue to update the UI  if the user is expecting the UI to update.  We should not use onPause() to save application or user data,  to make network calls, or to execute database transactions.  5. onStop()  This is called when an activity is no longer visible to the user.  This may happen because the activity is being destroyed,  a new activity is starting, or an existing activity is entering a Resumed state  and is covering the stopped activity.  6. onRestart()  This is called when an activity in the Stopped state is about to restart.  onRestart() restores the state of the activity from the time  that it was stopped.  This callback is always followed by onStart().  7. onDestroy()  This is called when the activity is finished or destroyed.  onDestroy() is usually implemented to ensure  that all of an activity’s resources are released  when the activity is destroyed. | </end> |
| <hitle> | Activity state change | <chare> | 2 | <pext> | Different events, some user-triggered and some system-triggered,  can cause an Activity to transition from one state to another. | </end> |
| <hitle> | Configuration change | <chare> | 3 | <pext> | There are a number of events that can trigger a configuration change.  The most example is a change  between portrait and landscape orientations.  Other cases that can cause configuration changes  include changes to language or input device.  When a configuration change occurs,  the activity is destroyed and recreated.  The original activity instance will have some callbacks triggered:  onPause()  onStop()  onDestroy()  A new instance of the activity will be created and have  some callbacks triggered:  onCreate(),  onStart(),  onResume()  We can use a combination of ViewModels,  the onSaveInstanceState() method, and persistent local storage  to preserve an activity’s UI state across configuration changes.  Deciding how to combine these options depends on  the complexity of your UI data, use cases for the app,  and consideration of speed of retrieval versus memory usage. | </end> |
| <hitle> | Activity state change in multi-window cases | <chare> | 3 | <pext> | When an app enters multi-window mode,  the system notifies the currently running activity  of a configuration change, thus going through the lifecycle transitions described above.  This behavior also occurs  if an app already in multi-window mode gets resized.  The activity can handle the configuration change itself,  or it can allow the system to destroy the activity and recreate it  with the new dimensions.  In multi-window mode, although there are two apps  that are visible to the user, only the one with which the user is interacting  is in the foreground and has focus.  That activity is in the Resumed state,  while the app in the other window is in the Paused state.  When the user switches from app A to app B,  the system calls onPause() on app A, and onResume() on app B.  It switches between these two methods  each time the user toggles between apps. | </end> |
| <hitle> | Activity/dialog appears in foreground | <chare> | 3 | <pext> | If a new activity or dialog appears in the foreground,  taking focus and partially covering the activity in progress,  the covered activity loses focus and enters the Paused state.  Then, the system calls onPause() on it.  When the covered activity returns to the foreground  and regains focus, it calls onResume().  If a new activity or dialog appears in the foreground,  taking focus and completely covering the activity in progress,  the covered activity loses focus and enters the Stopped state.  The system then, calls onPause() and onStop().  When the same instance of the covered activity  comes back to the foreground, the system calls onRestart(), onStart(),  and onResume() on the activity.  If it is a new instance of the covered activity  that comes to the background, the system does not call onRestart(),  only calling onStart() and onResume().  Note: When the user taps the Overview or Home button,  the system behaves as if the current activity has been completely covered. | </end> |
| <hitle> | User presses or gestures Back | <chare> | 3 | <pext> | If an activity is in the foreground, and the user presses or gestures Back,  the activity transitions through  the onPause(), onStop(), and onDestroy() callbacks.  In addition to being destroyed, the activity is also removed from the back stack.  It is important to note that, by default, the onSaveInstanceState() callback  does not fire in this case.  This behavior is based on the assumption that the user used Back  with no expectation of returning to the same instance of the activity.  However, we can override the onBackPressed() method  to implement custom behavior, such as displaying a dialog that asks  the user to confirm that they want to exit the app.  If we override the onBackPressed() method, we need to invoke  super.onBackPressed() from your overridden method.  Otherwise the system Back behavior may be jarring to the user. | </end> |
| <hitle> | System kills app process | <chare> | 3 | <pext> | If an app is in the background and the system needs to free up  additional memory for a foreground app,  the background app can be killed by the system to free up more memory.  A user can also kill a process by using the Application Manager  under Settings to kill the corresponding app.  onDestroy() will be called. | </end> |
| <hitle> | transient UI state | <chare> | 2 | <pext> | A user expects an activity’s UI state to remain the same  throughout a configuration change, such as rotation  or switching into multi-window mode.  However, the system destroys the activity by default  when such a configuration change occurs,  wiping away any UI state stored in the activity instance.  Similarly, a user expects UI state to remain the same  if they temporarily switch away from your app to a different app  and then come back to your app later.  However, the system may destroy your application’s process  while the user is away and your activity is stopped.  When the activity is destroyed due to system constraints,  you should preserve the user’s transient UI state  using a combination of ViewModel, onSaveInstanceState(), or local storage.  If UI data is simple and lightweight, such as a primitive data type  or a simple object like String, we can use onSaveInstanceState() alone  to persist the UI state across both configuration changes  and system-initiated process death.  In most cases, though, we should use both ViewModel and  onSaveInstanceState() since onSaveInstanceState() incurs  serialization/deserialization costs. | </end> |
| <hitle> | Instance state | <chare> | 3 | <pext> | There are a few scenarios in which activity is destroyed  due to normal app behavior, such as when the user  presses the Back button or the activity signals its own destruction  by calling the finish() method.  When activity is destroyed because the user presses Back  or the activity finishes itself, both the system's and the user's concept  of that Activity instance is gone forever.  In these scenarios, the user's expectation matches the system's behavior  and we do not have any extra work to do.  However, if the system destroys the activity due to system constraints  such as a configuration change or memory pressure,  then although the actual Activity instance is gone,  the system remembers that it existed.  If the user attempts to navigate back to the activity,  the system creates a new instance of that activity  using a set of saved data that describes the state of the activity  when it was destroyed.  The saved data that the system uses to restore the previous state  is called the instance state and is a collection of key-value pairs stored  in a Bundle object.  By default, the system uses the Bundle instance state to save information  about each View object in your activity layout  such as the text value entered into an EditText widget.  So, if your activity instance is destroyed and recreated,  the state of the layout is restored to its previous state  with no code required by you.  However, activity might have more state information  that you'd like to restore, such as member variables  that track the user's progress in the activity.  Note: In order for the Android system to restore the state of the views  in activity, each view must have a unique ID,  supplied by the android:id attribute.  A Bundle object isn't appropriate for preserving  more than a trivial amount of data because it requires serialization  on the main thread and consumes system-process memory.  To preserve more than a very small amount of data,  we should take a combined approach to preserving data,  using persistent local storage, the onSaveInstanceState() method,  and the ViewModel class. | </end> |
| <hitle> | Save UI state onSaveInstanceState() | <chare> | 3 | <pext> | As activity begins to stop, the system calls the onSaveInstanceState() method  so activity can save state information to an instance state bundle.  The default implementation of this method saves transient information  about the state of the activity's view hierarchy,  such as the text in an EditText widget or the scroll position of a ListView widget.  To save additional instance state information for activity,  we must override onSaveInstanceState() and add key-value pairs  to the Bundle object that is saved in the event  that activity is destroyed unexpectedly.  If we override onSaveInstanceState(),  we must call the superclass implementation if we want  the default implementation to save the state of the view hierarchy.  override fun onSaveInstanceState(outState: Bundle?) {  // Save the user's current game state  outState?.run {  putInt(STATE\_SCORE, currentScore)  putInt(STATE\_LEVEL, currentLevel)  }  // Always call the superclass so it can save the view hierarchy state  super.onSaveInstanceState(outState)  }  companion object {  val STATE\_SCORE = "playerScore"  val STATE\_LEVEL = "playerLevel"  }  Note: onSaveInstanceState() is not called when the user explicitly closes  the activity or in other cases when finish() is called.  To save persistent data, such as user preferences or data for a database,  we should take appropriate opportunities when activity is in the foreground.  If no such opportunity arises, we should save such data  during the onStop() method. | </end> |
| <hitle> | Restore UI state using saved instance state onSaveInstanceState() | <chare> | 3 | <pext> | When activity is recreated after it was previously destroyed,  we can recover the saved instance state from the Bundle  that the system passes to activity.  Both the onCreate() and onRestoreInstanceState() callback methods  receive the same Bundle that contains the instance state information.  Because the onCreate() method is called whether the system  is creating a new instance of your activity or recreating a previous one,  we must check whether the state Bundle is null  before you attempt to read it.  If it is null, then the system is creating a new instance of the activity,  instead of restoring a previous one that was destroyed.  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState) // Always call the superclass first  // Check whether we're recreating a previously destroyed instance  if (savedInstanceState != null) {  with(savedInstanceState) {  // Restore value of members from saved state  currentScore = getInt(STATE\_SCORE)  currentLevel = getInt(STATE\_LEVEL)  }  } else {  // Probably initialize members with default values for a new instance  }  // ...  }  Instead of restoring the state during onCreate()  we may choose to implement onRestoreInstanceState(),  which the system calls after the onStart() method.  The system calls onRestoreInstanceState()  only if there is a saved state to restore,  so we do not need to check whether the Bundle is null.  override fun onRestoreInstanceState(savedInstanceState: Bundle?) {  // Always call the superclass so it can restore the view hierarchy  super.onRestoreInstanceState(savedInstanceState)  // Restore state members from saved instance  savedInstanceState?.run {  currentScore = getInt(STATE\_SCORE)  currentLevel = getInt(STATE\_LEVEL)  }  }  Caution: Always call the superclass implementation  of onRestoreInstanceState() so the default implementation  can restore the state of the view hierarchy. | </end> |
| <hitle> | Navigate between activity | <chare> | 2 | <pext> | An app is likely to enter and exit an activity, perhaps many times,  during the app’s lifetime.  For example, the user may tap the device’s Back button,  or the activity may need to launch a different activity. | </end> |
| <hitle> | Start one activity from another | <chare> | 3 | <pext> | An activity often needs to start another activity at some point.  This need arises, for instance, when an app needs to move  from the current screen to a new one.  Depending on whether activity wants a result back  from the new activity it’s about to start, we start the new activity  using either the startActivity() or the startActivityForResult() method.  In either case, we pass in an Intent object.  The Intent object specifies either the exact activity we want to start  or describes the type of action you want to perform  and the system selects the appropriate activity for you,  which can even be from a different application.  An Intent object can also carry small amounts of data  to be used by the activity that is started. | </end> |
| <hitle> | startActivity() | <chare> | 4 | <pext> | If the newly started activity does not need to return a result,  the current activity can start it by calling the startActivity() method.  When working within application, we often need to simply launch  a known activity.  val intent = Intent(this, SignInActivity::class.java)  startActivity(intent)  (\*It shows how to launch an activity called SignInActivity.\*)  The application might also want to perform some action,  such as send an email, text message, or status update,  using data from your activity.  In this case, the application might not have its own activities  to perform such actions, so we can instead leverage the activities  provided by other applications on the device,  which can perform the actions for you.  This is where intents are really valuable: we can create an intent  that describes an action we want to perform and the system launches  the appropriate activity from another application.  If there are multiple activities that can handle the intent,  then the user can select which one to use.  val intent = Intent(Intent.ACTION\_SEND).apply {  putExtra(Intent.EXTRA\_EMAIL, recipientArray)  }  startActivity(intent)  (\*If we want to allow the user to send an email message, we can create intent like this.  The EXTRA\_EMAIL extra added to the intent is a string array of email addresses to which the email should be sent. When an email application responds to this intent, it reads the string array provided in the extra and places them in the "to" field of the email composition form. In this situation, the email application's activity starts and when the user is done, activity resumes.\*) | </end> |
| <hitle> | startActivityForResult() | <chare> | 4 | <pext> | Sometimes we want to get a result back from an activity when it ends.  For example, we may start an activity that lets the user pick a person  in a list of contacts; when it ends, it returns the person that was selected.  To do this, we call the startActivityForResult(Intent, int) method,  where the integer parameter identifies the call.  This identifier is meant to disambiguate between multiple calls  to startActivityForResult(Intent, int) from the same activity.  It's not global identifier and is not at risk of conflicting  with other apps or activities.  The result comes back through onActivityResult(int, int, Intent) method.  When a child activity exits, it can call setResult(int)  to return data to its parent.  The child activity must always supply a result code,  which can be the standard results RESULT\_CANCELED, RESULT\_OK,  or any custom values starting at RESULT\_FIRST\_USER.  In addition, the child activity can optionally return an Intent object  containing any additional data it wants.  The parent activity uses the onActivityResult(int, int, Intent) method,  along with the integer identifier the parent activity originally supplied,  to receive the information.  If a child activity fails for any reason, such as crashing,  the parent activity receives a result with the code RESULT\_CANCELED.  class MyActivity : Activity() {  // ...  override fun onKeyDown(keyCode: Int, event: KeyEvent?): Boolean {  if (keyCode == KeyEvent.KEYCODE\_DPAD\_CENTER) {  // When the user center presses, let them pick a contact.  startActivityForResult(  Intent(Intent.ACTION\_PICK,Uri.parse("content://contacts")),  PICK\_CONTACT\_REQUEST)  return true  }  return false  }  override fun onActivityResult(requestCode: Int, resultCode: Int, intent: Intent?) {  when (requestCode) {  PICK\_CONTACT\_REQUEST ->  if (resultCode == RESULT\_OK) {  startActivity(Intent(Intent.ACTION\_VIEW, intent?.data))  }  }  }  companion object {  internal val PICK\_CONTACT\_REQUEST = 0  }  } | </end> |
| <hitle> | startActivityFor() vs startActivityForResult() | <chare> | 4 | <pext> | 1. startActivity will start a new activity and not care  when where and how that activity finishes.  1-2. Evidently, startActivityForResult waits for callbacks  when the started activity decided to finish  2. startActivity() will start the activity we want to start  without worrying about getting any result from new child activity  started by startActivity to parent activity.  2-1. startActivityForResult() starts another activity from the activity  and it expect to get some data from newly started child activity  by startAcitvityForResult() and return that to parent activity. | </end> |
| <hitle> | Pass data between activity | <chare> | 4 | <pext> | There are several strategies that we can pass data between activities:  1. Intents  We can pass data usin Intent  and Intent supports three ways to pass data:  1-1. Direct: put our data into intents directly  1-2. Bundle: create a bundle and set the data here  1-3. Parcelable: It is a way of “serializing” our object.  2. Static class  We can use a static class to exchange data between activities.  In this case we can create several static method to hold the data  we want to share.  Otherwise, we can use the singleton pattern that ensures  we have only one instance of this class in all our JVM.  In this case we create several method to hold the data.  3. Shared Preferences  4. Database  Shared Preferences and Database are valid when we need to pass data  and at the same time make this information persistent. | </end> |
| <hitle> | Activity Pass data Intent direct | <chare> | 5 | <pext> | Intent has several method called putExtra(), getExtra().  that allows us to save inside the Intent our information.  Intent i = new Intent(EditActivity.this, ViewActivity.class);  i.putExtra("name", edtName.getText().toString();  i.putExtra("surname", edtSurname.getText().toString();  i.putExtra("email", edtEmail.getText().toString();  startActivity(i);  When we get the data from Intent from new started activity,  protected void onCreate(Bundle savedInstanceState) {  Intent i = getIntent();  String name = i.getStringExtra("name"));  String surname = i.getStringExtra("surname"));  String email = i.getStringExtra("email"));  . . .  } | </end> |
| <hitle> | Activity Pass data Intent Bundle | <chare> | 5 | <pext> | Android has a class called Bundle where we can store our data  and supports several data types  like strings, chars, boolean, integer and so on.  Instead of using the Intent, as data container,  we store our info directly into the bundle and then  we save the bundle into the Intent.  Intent i = new Intent(EditActivity.this, ViewActivity.class);  Bundle b = new Bundle();  b.putString("name", edtName.getText().toString());  b.putString("surname", edtSurname.getText().toString());  b.putString("email", edtEmail.getText().toString());  i.putExtra("personBdl", b);  startActivity(i);  In the activity where receiving the data,  Intent i = getIntent();  Bundle b = i.getBundleExtra("personBdl");  String name = b.getString("name");  String surname = b.getString("surname");  String email = b.getString("email"); | </end> |
| <hitle> | Activity Pass data Intent Parcelable | <chare> | 5 | <pext> | This is the most elegant way to pass data between activities.  The “parcelable” operation is like serialization in Java,  but while the java serialization in android is inefficient  the “parcelable” operation is much more faster.  -----------Serializable solution------------------------  We can use java Serializable.  Data class Fruits(val name): Serializable  And we will create an intent and use putExtra to pass data.  val intent = Intent(this@MainActivity, SecondActivity::class.java)  intent.putExtra(“pass\_data”, Fruits(“sending”))  And then we will get the data using getExtra.  Override fun onCreate()  {  val fruits = intent.extra?.get(“pass\_data”) as Fruits?  fruits?.let {  Text.text = fruits.name  }  }  -------------Parcelable solution---------------------------  Only difference from Serialization solution is  1. use ‘kotlin-parcelable’ plugin  In Gradle file, we need to add ‘kotlin-parcelable’ plugin.  Plugins {  …  id ‘kotlin-parcelable’  …  }  2. Use @Parcelize annotation and inherite Parcelize when declare classes.  @Parcelize  Data class Fruits(val name): Parcelable | </end> |
| <hitle> | Activity pass/return data to start/launching activity, get activity result, startActivityForResult() | <chare> | 5 | <pext> | We can use startActivityForResult(). | </end> |
| <hitle> | Coordinate activity | <chare> | 4 | <pext> | When one activity starts another, they both experience lifecycle transitions.  The first activity stops operating and enters the Paused or Stopped state,  while the other activity is created.  In case these activities share data saved to disc or elsewhere,  it's important to understand that the first activity is not completely stopped  before the second one is created.  Rather, the process of starting the second one overlaps with the process  of stopping the first one.  The order of lifecycle callbacks is well defined, particularly  when the two activities are in the same process  and one is starting the other.  Here's the order of operations that occur when Activity A starts Activity B:  Activity A's onPause() method executes.  Activity B's onCreate(), onStart(), and onResume() methods  execute in sequence. Activity B now has user focus.  Then, if Activity A is no longer visible on screen,  its onStop() method executes.  This predictable sequence of lifecycle callbacks allows to manage  the transition of information from one activity to another. | </end> |
| <hitle> | Coordinate activity | <chare> | 4 | <pext> | When one activity starts another, they both experience lifecycle transitions.  The first activity stops operating and enters the Paused or Stopped state,  while the other activity is created.  In case these activities share data saved to disc or elsewhere,  it's important to understand that the first activity is not completely stopped  before the second one is created.  Rather, the process of starting the second one overlaps with the process  of stopping the first one.  The order of lifecycle callbacks is well defined, particularly  when the two activities are in the same process  and one is starting the other.  Here's the order of operations that occur when Activity A starts Activity B:  Activity A's onPause() method executes.  Activity B's onCreate(), onStart(), and onResume() methods  execute in sequence. Activity B now has user focus.  Then, if Activity A is no longer visible on screen,  its onStop() method executes.  This predictable sequence of lifecycle callbacks allows to manage  the transition of information from one activity to another. | </end> |
| <hitle> | Test activity | <chare> | 2 | <pext> | -------------Create an activity--------------------------------  To create the activity under test, we can do like this.  @RunWith(AndroidJUnit4::class)  class MyTestSuite {  @Test fun testEvent() {  launchActivity<MyActivity>().use {  }  }  }  Alternatively, we can use ActivityScenarioRule to automatically  call ActivityScenario.launch before each test,  and ActivityScenario.close at test teardown.  @RunWith(AndroidJUnit4::class)  class MyTestSuite {  @get:Rule var activityScenarioRule = activityScenarioRule<MyActivity>()  @Test fun testEvent() {  val scenario = activityScenarioRule.scenario  }  }  ---------------Drive the activity to a new state-------------------  To drive the activity to a different state, such as CREATED or STARTED,  we can call moveToState().  This action simulates a situation where the activity is stopped or paused,  respectively, because it's interrupted by another app or a system action.  @RunWith(AndroidJUnit4::class)  class MyTestSuite {  @Test fun testEvent() {  launchActivity<MyActivity>().use { scenario ->  scenario.moveToState(State.CREATED)  }  }  }  -----------------Determine the current activity state----------------------  To determine the current state of an activity under test,  we can get the value of the state field within the ActivityScenario object.  It's particularly helpful to check the state of an activity under test  if the activity redirects to another activity or finishes itself.  @RunWith(AndroidJUnit4::class)  class MyTestSuite {  @Test fun testEvent() {  launchActivity<MyActivity>().use { scenario ->  scenario.onActivity { activity ->  startActivity(Intent(activity, MyOtherActivity::class.java))  }  val originalActivityState = scenario.state  }  }  }  --------------------Recreate the activity------------------------  If a device is low on resources, the system might destroy an activity,  requiring the app to recreate that activity  when the user returns to your app.  To simulate these conditions, we can call recreate().  @RunWith(AndroidJUnit4::class)  class MyTestSuite {  @Test fun testEvent() {  launchActivity<MyActivity>().use { scenario ->  scenario.recreate()  }  }  }  The ActivityScenario class maintains the activity's saved instance state  and any objects annotated using @NonConfigurationInstance.  These objects are loaded into the new instance of your activity under test.  ----------Retrieve activity results-------------------------------  To get the result code or data associated with a finished activity,  we can get the value of the result field within the ActivityScenario object.  @RunWith(AndroidJUnit4::class)  class MyTestSuite {  @Test fun testResult() {  launchActivity<MyActivity>().use {  onView(withId(R.id.finish\_button)).perform(click())  // Activity under test is now finished.  val resultCode = scenario.result.resultCode  val resultData = scenario.result.resultData  }  }  }  -------------Trigger actions in the activity-----------------------  All methods within ActivityScenario are blocking calls,  so the API requires you to run them in the instrumentation thread.  To trigger actions in your activity under test,  we can use Espresso view matchers to interact with elements in your view.  @RunWith(AndroidJUnit4::class)  class MyTestSuite {  @Test fun testEvent() {  launchActivity<MyActivity>().use {  onView(withId(R.id.refresh)).perform(click())  }  }  }  If we need to call a method on the activity itself, however,  we can do so safely by implementing ActivityAction.  @RunWith(AndroidJUnit4::class)  class MyTestSuite {  @Test fun testEvent() {  launchActivity<MyActivity>().use { scenario ->  scenario.onActivity { activity ->  activity.handleSwipeToRefresh()  }  }  }  } | </end> |
| <hitle> | Process and app lifecycle | <chare> | 2 | <pext> | In most cases, every Android application runs in its own Linux process.  This process is created for the application  when some of its code needs to be run, and will remain running  until it is no longer needed and the system needs to reclaim  its memory for use by other applications.  To determine which processes should be killed when low on memory,  Android places each process into an "importance hierarchy"  based on the components running in them  and the state of those components.  These process types are:  1. A foreground process is one that is required for  what the user is currently doing.  A process is considered to be in the foreground  if any of the following conditions hold:  1-1. It is running an Activity at the top of the screen  that the user is interacting with.  1-2. It has a BroadcastReceiver that is currently running.  1-3. It has a Service that is currently executing code  in one of its callbacks.  2. A visible process is doing work that the user is currently aware of,  so killing it would have a noticeable negative impact  on the user experience.  A process is considered visible in the following conditions:  2-1. It is running an Activity that is visible to the user on-screen  but not in the foreground (its onPause() method has been called).  2-2. It has a Service that is running as a foreground service,  through Service.startForeground().  2-3. It is hosting a service that the system is using  for a particular feature that the user is aware,  such as a live wallpaper, input method service, etc.  3. A service process is one holding a Service that has been started  with the startService() method.  4. A cached process is one that is not currently needed,  so the system is free to kill it as desired when resources  like memory are needed elsewhere. | </end> |
| <hitle> | UpdatingRecords | <chare> | 4 | <pext> | UpdatingRecords  UpdatingRecords  UpdatingRecords  UpdatingRecords  UpdatingRecords | </end> |