|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | Unit Testing | <chare> | 1 | <pext> | Unit testing means testing the written code by small parts.  We usually write programs that have classes  and further the logic in classes is divided into methods.  Unit refers to a small piece of code that can be either method,  class, or component.  The aim of unit tests is to be to verify the logic of individual units. | </end> |
| <hitle> | fragment pros | <chare> | 2 | <pext> | <starting>fragment pros</heading> 1. We can reuse a Fragment. Write the Fragment code once,  and reuse the Fragment in more than one Activity without having  to repeat code. 2. We can add or remove a Fragment dynamically.  Add, replace, or remove a Fragment from an Activity as needed. 3. We can integrate a mini-UI within the Activity. Integrate a Fragment  with an Activity UI or overlay the UI, so that the user can interact  with the Fragment UI without leaving the Activity. 4. We can retain data instances after a configuration change.  Since a Fragment has its own lifecycle, it can retain an instance of its data  after a configuration change (such as changing the device orientation). 5. We can represent sections of a layout for different screen sizes.  Encapsulating an interactive UI within a Fragment makes it easier  to display the interactive UI on different screen sizes. </ending> | </end> |
| <hitle> | Create fragment | <chare> | 2 | <pext> | To create a fragment, extend the AndroidX Fragment class,  and override its methods to insert your app logic,  similar to the way you would create an Activity class.  To create a minimal fragment that defines its own layout,  provide your fragment's layout resource to the base constructor  class ExampleFragment : Fragment(R.layout.example\_fragment)  The Fragment library also provides  more specialized fragment base classes:  1. DialogFragment  Displays a floating dialog.  Using this class to create a dialog is a good alternative  to using the dialog helper methods in the Activity class,  as fragments automatically handle the creation  and cleanup of the Dialog.  2. PreferenceFragmentCompat  Displays a hierarchy of Preference objects as a list.  We can use PreferenceFragmentCompat to create  a settings screen for the app. | </end> |
| <hitle> | Add fragment to activity | <chare> | 2 | <pext> | Generally, a fragment must be embedded within  an AndroidX FragmentActivity to contribute a portion of UI  to that activity's layout.  FragmentActivity is the base class for AppCompatActivity,  so if we're already subclassing AppCompatActivity  to provide backward compatibility in the app,  then we do not need to change the activity base class.  We can add fragment to the activity's view hierarchy  either by defining the fragment in the activity's layout file  or by defining a fragment container in your activity's layout file  and then programmatically adding the fragment  from within your activity.  In either case, we need to add a FragmentContainerView  that defines the location where the fragment should be placed  within the activity's view hierarchy.  It is strongly recommended to always use a FragmentContainerView  as the container for fragments, as FragmentContainerView  includes fixes specific to fragments that other view groups  such as FrameLayout do not provide. | </end> |
| <hitle> | Add fragment via XML | <chare> | 3 | <pext> | To declaratively add a fragment to activity layout's XML,  we use a FragmentContainerView element.  <!-- res/layout/example\_activity.xml -->  <androidx.fragment.app.FragmentContainerView  xmlns:android="http://schemas.android.com/apk/res/android"  android:id="@+id/fragment\_container\_view"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  android:name="com.example.ExampleFragment" />  The android:name attribute specifies the class name  of the Fragment to instantiate.  When the activity's layout is inflated, the specified fragment is instantiated,  onInflate() is called on the newly instantiated fragment,  and a FragmentTransaction is created to add the fragment  to the FragmentManager. | </end> |
| <hitle> | Add fragment programmatically | <chare> | 3 | <pext> | To programmatically add a fragment to your activity's layout,  the layout should include a FragmentContainerView to serve  as a fragment container.  <!-- res/layout/example\_activity.xml -->  <androidx.fragment.app.FragmentContainerView  xmlns:android="http://schemas.android.com/apk/res/android"  android:id="@+id/fragment\_container\_view"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent" />  Unlike the XML approach, the android:name attribute isn't used  on the FragmentContainerView,  so no specific fragment is automatically instantiated.  Instead, a FragmentTransaction is used to instantiate a fragment  and add it to the activity's layout.  While the activity is running, we can make fragment transactions  such as adding, removing, or replacing a fragment.  In FragmentActivity, we can get an instance of the FragmentManager,  which can be used to create a FragmentTransaction.  Then, we can instantiate the fragment  within the activity's onCreate() method using FragmentTransaction.add(),  passing in the ViewGroup ID of the container in your layout  and the fragment class you want to add  and then commit the transaction.  class ExampleActivity : AppCompatActivity(R.layout.example\_activity) {  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  if (savedInstanceState == null) {  supportFragmentManager.commit {  setReorderingAllowed(true)  add<ExampleFragment>(R.id.fragment\_container\_view)  }  }  }  }  Note: we should always use setReorderingAllowed(true)  when performing a FragmentTransaction.  The fragment transaction is only created  when savedInstanceState is null.  This is to ensure that the fragment is added only once,  when the activity is first created.  When a configuration change occurs and the activity is recreated,  savedInstanceState is no longer null,  and the fragment does not need to be added a second time,  as the fragment is automatically restored  from the savedInstanceState.  If the fragment requires some initial data,  arguments can be passed to fragment by providing a Bundle  in the call to FragmentTransaction.add().  class ExampleActivity : AppCompatActivity(R.layout.example\_activity) {  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  if (savedInstanceState == null) {  val bundle = bundleOf("some\_int" to 0)  supportFragmentManager.commit {  setReorderingAllowed(true)  add<ExampleFragment>(R.id.fragment\_container\_view, args = bundle)  }  }  }  }  The arguments Bundle can then be retrieved from within fragment  by calling requireArguments(),  and the appropriate Bundle getter methods can be used  to retrieve each argument.  class ExampleFragment : Fragment(R.layout.example\_fragment) {  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  val someInt = requireArguments().getInt("some\_int")  ...  }  } | </end> |
| <hitle> | FragmentManager | <chare> | 2 | <pext> | FragmentManager is the class responsible for performing actions  on the app's fragments, such as adding, removing, or replacing them,  and adding them to the back stack.  We might never interact with FragmentManager directly  if we're using the Jetpack Navigation library, as it works  with the FragmentManager.  That said, any app using fragments is using FragmentManager  at some level.  The FragmentManager manages the fragment back stack.  At runtime, the FragmentManager can perform back stack operations  like adding or removing fragments in response to user interactions.  Each set of changes are committed together as a single unit  called a FragmentTransaction.  When the user presses the Back button on their device,  or when you call FragmentManager.popBackStack(),  the top-most fragment transaction is popped off of the stack.  In other words, the transaction is reversed.  If there are no more fragment transactions on the stack,  and if you aren't using child fragments,  the back event bubbles up to the activity.  When we call addToBackStack() on a transaction,  the transaction can include any number of operations,  such as adding multiple fragments,  replacing fragments in multiple containers, and so on.  When the back stack is popped, all of these operations are reversed  as a single atomic action.  If we've committed additional transactions prior to the popBackStack() call,  and if you did not use addToBackStack() for the transaction,  these operations are not reversed.  Therefore, within a single FragmentTransaction,  avoid interleaving transactions that affect the back stack  with those that do not. | </end> |
| <hitle> | Access FragmentManager in activity | <chare> | 3 | <pext> | Every FragmentActivity and subclasses thereof,  such as AppCompatActivity, have access  to the FragmentManager through  the getSupportFragmentManager() method. | </end> |
| <hitle> | Access FragmentManager in Fragment | <chare> | 3 | <pext> | Fragments are also capable of hosting one or more child fragments.  Inside a fragment, we can get a reference to the FragmentManager  that manages the fragment's children  through getChildFragmentManager().  If we need to access its host FragmentManager,  we can use getParentFragmentManager(). | </end> |
| <hitle> | Child fragments | <chare> | 3 | <pext> | Generally speaking, the app should consist of a single  or small number of activities in your application project,  with each activity representing a group of related screens.  The activity may provide a point to place top-level navigation  and a place to scope ViewModels and other view-state  between fragments.  Each individual destination in the app should be represented  by a fragment.  If we want to show multiple fragments at once,  such as in a split-view or a dashboard, we should use  child fragments that are managed by the destination fragment  and its child fragment manager.  We can use child fragments when:  1. Screen slides, with a ViewPager2 in a parent fragment  to manage a series of child fragment views.  2. Sub-navigation within a set of related screens.  3. Jetpack Navigation uses child fragments as individual destinations.  An activity hosts a single parent NavHostFragment  and fills its space with different child destination fragments  as users navigate through the app. | </end> |
| <hitle> | Fragment transaction | <chare> | 3 | <pext> | To display a fragment within a layout container,  we can use the FragmentManager to create a FragmentTransaction.  Within the transaction, we can then perform an add()  or replace() operation on the container.  supportFragmentManager.commit {  replace<ExampleFragment>(R.id.fragment\_container)  setReorderingAllowed(true)  addToBackStack("name") // name can be null  }  (\*ExampleFragment replaces the fragment, if any, that is currently in the layout container identified by the R.id.fragment\_container ID.)  Providing the fragment's class to the replace() method  allows the FragmentManager to handle instantiation  using its FragmentFactory.  setReorderingAllowed(true) optimizes the state changes  of the fragments involved in the transaction  so that animations and transitions work correctly.  Calling addToBackStack() commits the transaction to the back stack.  The user can later reverse the transaction  and bring back the previous fragment by pressing the Back button.  If we added or removed multiple fragments within a single transaction,  all of those operations are undone when the back stack is popped.  The optional name provided in the addToBackStack() call  gives you the ability to pop back to that specific transaction  using popBackStack().  If we don't call addToBackStack() when we perform a transaction  that removes a fragment, then the removed fragment is destroyed  when the transaction is committed,  and the user cannot navigate back to it.  If we do call addToBackStack() when removing a fragment,  then the fragment is only STOPPED and is later RESUMED  when the user navigates back.  And its view is destroyed in this case. | </end> |
| <hitle> | Find fragment | <chare> | 4 | <pext> | We can get a reference to the current fragment  within a layout container by using findFragmentById().  We can use findFragmentById() to look up a fragment  either by the given ID when inflated from XML  or by the container ID when added in a FragmentTransaction.  supportFragmentManager.commit {  replace<ExampleFragment>(R.id.fragment\_container)  setReorderingAllowed(true)  addToBackStack(null)  }  ...  val fragment: ExampleFragment =  supportFragmentManager.findFragmentById(R.id.fragment\_container) as ExampleFragment  Alternatively, we can assign a unique tag to a fragment  and get a reference using findFragmentByTag().  We can assign a tag using the android:tag XML attribute on fragments  that are defined within your layout, or during an add()  or replace() operation within a FragmentTransaction.  supportFragmentManager.commit {  replace<ExampleFragment>(R.id.fragment\_container, "tag")  setReorderingAllowed(true)  addToBackStack(null)  }  ...  val fragment: ExampleFragment =  supportFragmentManager.findFragmentByTag("tag") as ExampleFragment | </end> |
| <hitle> | FragmentManager fragment primary navigation | <chare> | 4 | <pext> | Only one FragmentManager is allowed to control  the fragment back stack at any given time.  If the app shows multiple sibling fragments on the screen  at the same time, or if the app uses child fragments,  then one FragmentManager must be designated to handle  the app's primary navigation.  To define the primary navigation fragment  inside of a fragment transaction,  we can call the setPrimaryNavigationFragment() method  on the transaction, passing in the instance of the fragment  whose childFragmentManager should have primary control.  Let’s say the navigation structure as a series of layers,  with the activity as the outermost layer,  wrapping each layer of child fragments underneath.  Each layer must have a single primary navigation fragment.  When the Back event occurs,  the innermost layer controls navigation behavior.  Once the innermost layer has no more fragment transactions  from which to pop back, control returns to the next layer out,  and this process repeats until you reach the activity.  When two or more fragments are displayed at the same time,  only one of them can be the primary navigation fragment.  Setting a fragment as the primary navigation  fragment removes the designation from the previous fragment.  If we set the detail fragment as the primary navigation fragment,  the main fragment's designation is removed. | </end> |
| <hitle> | FragmentManager multiple back stacks | <chare> | 3 | <pext> | In some cases, the app might need to support multiple back stacks.  A common example is if the app uses a bottom navigation bar.  FragmentManager allows to support multiple back stacks  with the saveBackStack() and restoreBackStack() methods.  These methods allow to swap between back stacks  by saving one back stack and restoring a different one.  Alternatively, we can use the NavigationUI component,  which automatically handles multiple back stack support  for bottom navigation.  saveBackStack() works similarly to calling popBackStack()  with the optional name parameter.  The specified transaction and all transactions after it on the stack  are popped.  The difference is that saveBackStack() saves the state of  all fragments in the popped transactions.  supportFragmentManager.commit {  replace<ExampleFragment>(R.id.fragment\_container)  setReorderingAllowed(true)  addToBackStack("replacement")  }  supportFragmentManager.saveBackStack("replacement")  (\*It previously added a fragment to the back stack by committing a FragmentTransaction using addToBackStack().  In that case, you can save this fragment transaction and the state of ExampleFragment by calling saveState().\*)  Note: You can use saveBackStack() only with transactions that call setReorderingAllowed(true) to ensure that the transactions can be restored as a single, atomic operation.  We can call restoreBackStack() with the same name parameter to restore all of the popped transactions and all of the saved fragment states:  supportFragmentManager.restoreBackStack("replacement")  Note: We can't use saveBackStack() and restoreBackStack()  unless we pass an optional name for your fragment transactions  with addToBackStack(). | </end> |
| <hitle> | dependencies to fragments | <chare> | 3 | <pext> | When adding a fragment, we can instantiate the fragment manually  and add it to the FragmentTransaction.  fragmentManager.commit {  // Instantiate a new instance before adding  val myFragment = ExampleFragment()  add(R.id.fragment\_view\_container, myFragment)  setReorderingAllowed(true)  }  When we commit the fragment transaction,  the instance of the fragment we created is the instance used.  However, during a configuration change,  the activity and all of its fragments are destroyed and then recreated  with the most applicable Android resources.  The FragmentManager handles all of this for you.  It recreates instances of your fragments, attaches them to the host,  and recreates the back stack state.  By default, the FragmentManager uses a FragmentFactory  that the framework provides to instantiate a new instance  of your fragment.  This default factory uses reflection to find and invoke  a no-argument constructor for your fragment.  This means that we can't use this default factory  to provide dependencies to your fragment.  It also means that any custom constructor we used to create fragment  the first time is not used during recreation by default.  To provide dependencies to the fragment,  or to use any custom constructor,  we must instead create a custom FragmentFactory subclass  and then override FragmentFactory.instantiate.  We can then override the FragmentManager's default factory  with your custom factory,  which is then used to instantiate your fragments.  Let’s say we have a DessertsFragment that is responsible  for displaying popular desserts in your hometown.  Let's assume that DessertsFragment has a dependency  on a DessertsRepository class that provides it with the information  it needs to display the correct UI to your user.  We might define DessertsFragment to require  a DessertsRepository instance in its constructor.  And then we can implement our own FragmentFactory.  class DessertsFragment(val dessertsRepository: DessertsRepository) : Fragment() {  ...  }  class MyFragmentFactory(val repository: DessertsRepository) : FragmentFactory() {  override fun instantiate(classLoader: ClassLoader, className: String): Fragment =  when (loadFragmentClass(classLoader, className)) {  DessertsFragment::class.java -> DessertsFragment(repository)  else -> super.instantiate(classLoader, className)  }  }  (\*It subclasses FragmentFactory, overriding the instantiate() method to provide custom fragment creation logic for a DessertsFragment. Other fragment classes are handled by the default behavior of FragmentFactory through super.instantiate().  You can then designate MyFragmentFactory as the factory to use when constructing your app's fragments by setting a property on the FragmentManager. You must set this property prior to your activity's super.onCreate() to ensure that MyFragmentFactory is used when recreating your fragments.\*)  class MealActivity : AppCompatActivity() {  override fun onCreate(savedInstanceState: Bundle?) {  supportFragmentManager.fragmentFactory = MyFragmentFactory(DessertsRepository.getInstance())  super.onCreate(savedInstanceState)  }  }  Note that setting the FragmentFactory in the activity  overrides fragment creation throughout  the activity's fragments hierarchy.  In other words, the childFragmentManager  of any child fragments we add uses the custom fragment factory set  unless overridden at a lower level. | </end> |