|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | Navigation | <chare> | 1 | <pext> | Navigation refers to the interactions that allow users to navigate across,  into, and back out from the different pieces of content within your app.  Android Jetpack's Navigation component helps us implement navigation,  from simple button clicks to more complex patterns,  such as app bars and the navigation drawer.  The Navigation component also ensures a consistent  and predictable user experience by adhering to an established  set of principles.  Note: The Navigation component is designed for apps  that have one main activity with multiple fragment destinations.  The main activity is associated with a navigation graph and contains  a NavHostFragment that is responsible for swapping destinations  as needed.  In an app with multiple activity destinations,  each activity has its own navigation graph.  ------------------Three key parts Navigation graph, NavHost, NavController------------------  The Navigation component consists of three key parts:  1. Navigation graph:  An XML resource that contains all navigation-related information  in one centralized location.  This includes all of the individual content areas within your app,  called destinations, as well as the possible paths that a user  can take through your app.  2. NavHost:  An empty container that displays destinations  from your navigation graph.  The Navigation component contains a default NavHost implementation,  NavHostFragment, that displays fragment destinations.  3. NavController:  An object that manages app navigation within a NavHost.  The NavController orchestrates the swapping of destination content  in the NavHost as users move throughout your app. | </end> |
| <hitle> | Navigation pros | <chare> | 2 | <pext> | The Navigation component provides benefits of:  1. Handling fragment transactions.  2. Handling Up and Back actions correctly by default.  3. Providing standardized resources for animations and transitions.  4. Implementing and handling deep linking.  5. Including Navigation UI patterns, such as navigation drawers  and bottom navigation, with minimal additional work.  6. Safe Args - a Gradle plugin that provides type safety  when navigating and passing data between destinations.  7. ViewModel support - we can scope a ViewModel  to a navigation graph to share UI-related data  between the graph's destinations. | </end> |
| <hitle> | navigation graph | <chare> | 2 | <pext> | A navigation graph is a resource file that contains  all of the destinations and actions.  The graph represents all of the app's navigation paths.  Each destination is represented by a preview thumbnail,  and connecting actions are represented by arrows that show  how users can navigate from one destination to another. | </end> |
| <hitle> | navigation Destinations and Actions | <chare> | 3 | <pext> | Destinations are the different content areas in your app.  Actions are logical connections between the destinations  that represent paths that users can take. | </end> |
| <hitle> | NavHost | <chare> | 3 | <pext> | The navigation host is an empty container  where destinations are swapped in and out as a user navigates  through the app.  A navigation host must derive from NavHost.  The Navigation component's default NavHost implementation,  NavHostFragment, handles swapping fragment destinations. | </end> |
| <hitle> | Add a NavHostFragment via XML | <chare> | 4 | <pext> | We can add a NavHostFragment via XML.  <?xml version="1.0" encoding="utf-8"?>  <androidx.constraintlayout.widget.ConstraintLayout  xmlns:android="http://schemas.android.com/apk/res/android"  xmlns:app="http://schemas.android.com/apk/res-auto"  xmlns:tools="http://schemas.android.com/tools"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  tools:context=".MainActivity">  <androidx.appcompat.widget.Toolbar  .../>  <androidx.fragment.app.FragmentContainerView  android:id="@+id/nav\_host\_fragment"  android:name="androidx.navigation.fragment.NavHostFragment"  android:layout\_width="0dp"  android:layout\_height="0dp"  app:layout\_constraintLeft\_toLeftOf="parent"  app:layout\_constraintRight\_toRightOf="parent"  app:layout\_constraintTop\_toTopOf="parent"  app:layout\_constraintBottom\_toBottomOf="parent"  app:defaultNavHost="true"  app:navGraph="@navigation/nav\_graph" />  <com.google.android.material.bottomnavigation.BottomNavigationView  .../>  </androidx.constraintlayout.widget.ConstraintLayout>  1. The android:name attribute contains the class name  of your NavHost implementation.  2. The app:navGraph attribute associates the NavHostFragment  with a navigation graph.  The navigation graph specifies all of the destinations  in this NavHostFragment to which users can navigate.  3. The app:defaultNavHost="true" attribute ensures  that the NavHostFragment intercepts the system Back button.  Note that only one NavHost can be the default.  If you have multiple hosts in the same layout (two-pane layouts, for example),  we need to be sure to specify only one default NavHost. | </end> |
| <hitle> | Add destinations to the navigation graph | <chare> | 3 | <pext> | We can create a destination from an existing fragment or activity.  We can also use the Navigation Editor to create a new destination  or create a placeholder to later replace with a fragment or activity.  To add a new destination using the Navigation Editor:  1. In the Navigation Editor, click the New Destination icon ,  and then click Create new destination.  2. In the New Android Component dialog that appears, create your fragment. | </end> |
| <hitle> | Anatomy of a destination | <chare> | 3 | <pext> | In the NavigationEditor, click on a destination to select it,  and note the following attributes in the Attributes panel:  1. The Type field indicates whether the destination is implemented  as a fragment, activity, or other custom class in your source code.  2. The Label field contains the user-readable name of the destination.  This might be surfaced to the UI—for example, if we connect  the NavGraph to a Toolbar using setupWithNavController().  For this reason, it is recommended that we use resource strings  for this value.  3. The ID field contains the ID of the destination which is used  to refer to the destination in code.  4. The Class dropdown shows the name of the class that is associated  with the destination.  We can click this dropdown to change the associated class  to another destination type.  Click the Text tab to show the XML view of the navigation graph.  The XML contains the same id, name, label, and layout attributes  for the destination.  <?xml version="1.0" encoding="utf-8"?>  <navigation xmlns:app="http://schemas.android.com/apk/res-auto"  xmlns:tools="http://schemas.android.com/tools"  xmlns:android="http://schemas.android.com/apk/res/android"  app:startDestination="@id/blankFragment">  <fragment  android:id="@+id/blankFragment"  android:name="com.example.cashdog.cashdog.BlankFragment"  android:label="@string/label\_blank"  tools:layout="@layout/fragment\_blank" />  </navigation> | </end> |
| <hitle> | screen as start destination | <chare> | 3 | <pext> | The start destination is the first screen users see when opening your app,  and it's the last screen users see when exiting your app.  The Navigation editor uses a house icon to indicate the start destination.  Once we have all of destinations in place, we can choose  a start destination like this  In the Design tab, click on the destination to highlight it.  Click the Assign start destination button .  Alternatively, we can right-click on the destination  and click Set as Start Destination. | </end> |
| <hitle> | Connect destinations | <chare> | 3 | <pext> | An action is a logical connection between destinations.  Actions are represented in the navigation graph as arrows.  Actions usually connect one destination to another,  though we can also create global actions that take us  to a specific destination from anywhere in your app.  With actions, we're representing the different paths  that users can take through your app.  Note that to actually navigate to destinations,  we still need to write the code to perform the navigation.  In Navigation Editor, We can connect two destinations like this:  1. In the Design tab, hover over the right side of the destination  that we want users to navigate from.  A circle appears over the right side of the destination.  2. Click and drag cursor over the destination we want users to navigate to,  and release.  The resulting line between the two destinations represents an action.  3. Click on the arrow to highlight the action.  You can see attributes appear in the Attributes panel:  3-1. The Type field contains “Action”.  2-2. The ID field contains the ID for the action.  3-3The Destination field contains the ID for the destination fragment or activity.  4. Click the Text tab to toggle to the XML view.  An action element is now added to the source destination.  The action has an ID and a destination attribute that contains  the ID of the next destination.  <?xml version="1.0" encoding="utf-8"?>  <navigation xmlns:app="http://schemas.android.com/apk/res-auto"  xmlns:tools="http://schemas.android.com/tools"  xmlns:android="http://schemas.android.com/apk/res/android"  app:startDestination="@id/blankFragment">  <fragment  android:id="@+id/blankFragment"  android:name="com.example.cashdog.cashdog.BlankFragment"  android:label="@string/label\_blank"  tools:layout="@layout/fragment\_blank" >  <action  android:id="@+id/action\_blankFragment\_to\_blankFragment2"  app:destination="@id/blankFragment2" />  </fragment>  <fragment  android:id="@+id/blankFragment2"  android:name="com.example.cashdog.cashdog.BlankFragment2"  android:label="@string/label\_blank\_2"  tools:layout="@layout/fragment\_blank\_fragment2" />  </navigation>  In navigation graph, actions are represented by <action> elements.  At a minimum, an action contains its own ID and the ID  of the destination to which a user should be taken. | </end> |
| <hitle> | Navigate to a destination | <chare> | 2 | <pext> | Navigating to a destination is done using a NavController,  an object that manages app navigation within a NavHost.  Each NavHost has its own corresponding NavController.  We can retrieve a NavController by using:  1. Fragment.findNavController()  2. View.findNavController()  3. Activity.findNavController(viewId: Int)  When creating the NavHostFragment using FragmentContainerView  or if manually adding the NavHostFragment to the activity  via a FragmentTransaction, attempting to retrieve the NavController  in onCreate() of an Activity  via Navigation.findNavController(Activity, @IdRes int) will fail.  We should retrieve the NavController directly  from the NavHostFragment instead.  val navHostFragment =  supportFragmentManager.findFragmentById(R.id.nav\_host\_fragment) as NavHostFragment  val navController = navHostFragment.navController | </end> |
| <hitle> | Safe Args navigate type-safety | <chare> | 2 | <pext> | The recommended way to navigate between destinations is to use  the Safe Args Gradle plugin.  This plugin generates simple object and builder classes that enable  type-safe navigation between destinations.  Safe Args is recommended both for navigating  as well as passing data between destinations.  After enabling Safe Args, generated code contains classes and methods  for each action defined as well as classes that correspond  to each sending and receiving destination.  Safe Args generates a class for each destination  where an action originates.  The generated class name adds "Directions"  to the originating destination class name.  For example, if the originating destination is named SpecifyAmountFragment,  the generated class is named SpecifyAmountFragmentDirections.  The generated class contains a static method for each action  defined in the originating destination.  This method takes any defined action parameters as arguments  and returns a NavDirections object that we can pass directly to navigate().  As an example, let’s see we have a navigation graph with a single action that connects two destinations, SpecifyAmountFragment and ConfirmationFragment. The ConfirmationFragment takes a single float parameter that you provide as part of the action.  Safe Args generates a SpecifyAmountFragmentDirections class with a single method, actionSpecifyAmountFragmentToConfirmationFragment(), and an inner class called ActionSpecifyAmountFragmentToConfirmationFragment.  The inner class is derived from NavDirections and stores the associated action ID and float parameter.  The returned NavDirections object can then be passed directly to navigate().  override fun onClick(v: View) {  val amount: Float = ...  val action =  SpecifyAmountFragmentDirections  .actionSpecifyAmountFragmentToConfirmationFragment(amount)  v.findNavController().navigate(action)  } | </end> |
| <hitle> | Navigate using ID | <chare> | 2 | <pext> | navigate(int) takes the resource ID of either an action or a destination.  viewTransactionsButton.setOnClickListener { view ->  view.findNavController().navigate(R.id.viewTransactionsAction)  }  (\*The following code snippet shows how to navigate to the ViewTransactionsFragment.\*)  For buttons, there are three variants of  Navigation.createNavigateOnClickListener().  If you're using Kotlin, OnClickListener is a SAM interface,  so we can use a trailing lambda.  This approach can be shorter and easier to read than calling  createNavigateOnClickListener() directly. | </end> |
| <hitle> | NavOptions programmatically | <chare> | 3 | <pext> | We can create NavOptions and set programmatically.  findNavController().navigate(  R.id.action\_fragmentOne\_to\_fragmentTwo,  null,  navOptions { // Use the Kotlin DSL for building NavOptions  anim {  enter = android.R.animator.fade\_in  exit = android.R.animator.fade\_out  }  }  )  (\*It uses an extended form of navigate() and contains additional Bundle and NavOptions arguments.\*)  We can also programmatically apply NavOptions when navigating to implicit deep links:  findNavController().navigate(  deepLinkUri,  navOptions { // Use the Kotlin DSL for building NavOptions  anim {  enter = android.R.animator.fade\_in  exit = android.R.animator.fade\_out  }  }  )  (\*This variant of navigate() takes a Uri for the implicit deep link, as well as the NavOptions instance.\*) | </end> |
| <hitle> | Pass data between destinations | <chare> | 2 | <pext> | ----------pass Only by Android Studio Editor-----------  1. Define destination arguments  To pass data between destinations, first we need define the argument  by adding it to the destination that receives it  In the Navigation editor, click on the destination that receives the argument.  In the Attributes panel, click Add (+).  <fragment android:id="@+id/myFragment" >  <argument  android:name="myArg"  app:argType="integer"  android:defaultValue="0" />  </fragment>  2. Override a destination argument in an action  We can override the default value of an argument  or set one if it doesn't already exist by defining an argument at the action.  This argument must be of the same name and type as the argument  declared in the destination.  <action android:id="@+id/startMyFragment"  app:destination="@+id/myFragment">  <argument  android:name="myArg"  app:argType="integer"  android:defaultValue="1" />  </action>  --------------------- Safe Args--------------------  The Navigation component has a Gradle plugin called Safe Args  that generates simple object and builder classes for type-safe navigation  and access to any associated arguments.  Safe Args is strongly recommended for navigating and passing data,  because it ensures type-safety.  After enabling Safe Args, the generated code contains  the type-safe classes and methods for each action  as well as with each sending and receiving destination.  ------------------Howto Pass with Safe Args---------------------  We need to set an argument and pass it to the navigate() method.  override fun onClick(v: View) {  val amountTv: EditText = view!!.findViewById(R.id.editTextAmount)  val amount = amountTv.text.toString().toInt()  val action = SpecifyAmountFragmentDirections.confirmationAction(amount)  v.findNavController().navigate(action)  }  In the receiving destination’s code, we need to use  the getArguments() method to retrieve the bundle and use its contents.  we can also use the by navArgs() property delegate to access arguments.  val args: ConfirmationFragmentArgs by navArgs()  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  val tv: TextView = view.findViewById(R.id.textViewAmount)  val amount = args.amount  tv.text = amount.toString()  }  ----------------Safe Args with a global action-------------  When using Safe Args with a global action,  we must provide an android:id value for the root <navigation> element.  <?xml version="1.0" encoding="utf-8"?>  <navigation xmlns:app="http://schemas.android.com/apk/res-auto"  xmlns:tools="http://schemas.android.com/tools"  xmlns:android="http://schemas.android.com/apk/res/android"  android:id="@+id/main\_nav"  app:startDestination="@id/mainFragment">  ...  </navigation>  Navigation generates a Directions class for the <navigation> element  that is based on the android:id value.  Like, if we have a <navigation> element with android:id=@+id/main\_nav,  the generated class is called MainNavDirections.  All destinations within the <navigation> element have generated methods  for accessing all associated global actions.  -------------Pass with Bundle--------------------  We can still pass arguments between destinations by using Bundle objects.  We need to create a Bundle object and pass it to the destination  using navigate().  val bundle = bundleOf("amount" to amount)  view.findNavController().navigate(R.id.confirmationAction, bundle)  In the receiving destination’s code,  we can use the getArguments() method to retrieve the Bundle  and use its contents.  val tv = view.findViewById<TextView>(R.id.textViewAmount)  tv.text = arguments?.getString("amount")  ---------------Pass data to the start destination--------------------  We can pass data to the app's start destination.  First, we must explicitly construct a Bundle that holds the data.  Next, we can use several methods to pass the Bundle.  1. If we're creating NavHost programmatically,  we can call NavHostFragment.create(R.navigation.graph, args),  where args is the Bundle that holds your data.  2. Otherwise, we can set start destination arguments  by calling NavController.setGraph()  2.1. Using the ID of the graph, navController.setGraph(R.navigation.graph, args)  2.2. Using the graph itself, navController.setGraph(navGraph, args)  To retrieve the data in your start destination,  We can call Fragment.getArguments().  -----------SafeArgs types-----------  app:argType="integer"  app:argType="float"  app:argType="long"  app:argType="boolean"  app:argType="string"  app:argType="reference"  app:argType="<type>", where <type> is the fully-qualified class name of the Parcelable  app:argType="<type>", where <type> is the fully-qualified class name of the Serializable  app:argType="<type>", where <type> is the fully-qualified name of the enum | </end> |
| <hitle> | Animation transition navigation graph component | <chare> | 2 | <pext> | To add animations to an action,  1. In the Navigation editor, click on the action  where the animation should occur.  2. In the Animations section of the Attributes panel,  click the dropdown arrow next to the animation we'd like to add.  We can choose:  1. Entering a destination  2. Exiting a destination  3. Entering a destination via a pop action  4. Exiting a destination via a pop action  3. And then we can choose an animation  from the list of project animations that appears.  -------------Shared element animation transitions----------  The FragmentNavigator.Extras class allows to map shared elements  from one destination to the next by their transition name,  similar to using FragmentTransaction.addSharedElement().  We can then pass the extras to navigate().  val extras = FragmentNavigatorExtras(view1 to "hero\_image")  view.findNavController().navigate(  R.id.confirmationAction,  null, // Bundle of args  null, // NavOptions  extras)  Activities rely on ActivityOptionsCompat  to control Shared Element Transitions.  val options = ActivityOptionsCompat.makeSceneTransitionAnimation(activity,  UtilPair.create(view1, "hero\_image"))  val extras = ActivityNavigatorExtras(options)  view.findNavController().navigate(  R.id.details,  null, // Bundle of args  null, // NavOptions  extras)  -----------Apply pop animations to activity transitions----------  When we navigate to or from an Activity,  pop animations are not applied automatically.  Instead, we must call  ActivityNavigator.applyPopAnimationsToPendingTransition()  from the target Activity destination where the animation should occur.  override fun finish() {  super.finish()  ActivityNavigator.applyPopAnimationsToPendingTransition(this)  } | </end> |
| <hitle> | NavigationUI components | <chare> | 1 | <pext> | The Navigation component includes a NavigationUI class.  This class contains static methods that manage navigation  with the top app bar, the navigation drawer, and bottom navigation.  ----------Top app bar------------  The top app bar provides a consistent place along the top of the app  for displaying information and actions from the current screen.  -----------Navigation Destination label---------------  NavigationUI uses the destination labels from navigation graph  to keep the title of the top app bar up-to-date.  <navigation>  <fragment ...  android:label="Page title">  ...  </fragment>  </navigation>  -----------top app bar types------------  1. Toolbar  2. CollapsingToolbarLayout  3. ActionBar  -----------AppBarConfiguration-------------  NavigationUI uses an AppBarConfiguration object  to manage the behavior of the Navigation button  in the upper-left corner of your app's display area.  --------------create Toolbar-----------  To create a Toolbar with NavigationUI,  first define the bar in the main activity.  <LinearLayout>  <androidx.appcompat.widget.Toolbar  android:id="@+id/toolbar" />  <androidx.fragment.app.FragmentContainerView  android:id="@+id/nav\_host\_fragment"  ... />  ...  </LinearLayout>  Next, call setupWithNavController()  from the main activity's onCreate() method.  override fun onCreate(savedInstanceState: Bundle?) {  setContentView(R.layout.activity\_main)  ...  val navController = findNavController(R.id.nav\_host\_fragment)  val appBarConfiguration = AppBarConfiguration(navController.graph)  findViewById<Toolbar>(R.id.toolbar)  .setupWithNavController(navController, appBarConfiguration)  }  ---------------CollapsingToolbarLayout-----------  To include a CollapsingToolbarLayout with Toolbar,  first define the Toolbar and surrounding layout in the activity.  <LinearLayout>  <com.google.android.material.appbar.AppBarLayout  android:layout\_width="match\_parent"  android:layout\_height="@dimen/tall\_toolbar\_height">  <com.google.android.material.appbar.CollapsingToolbarLayout  android:id="@+id/collapsing\_toolbar\_layout"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  app:contentScrim="?attr/colorPrimary"  app:expandedTitleGravity="top"  app:layout\_scrollFlags="scroll|exitUntilCollapsed|snap">  <androidx.appcompat.widget.Toolbar  android:id="@+id/toolbar"  android:layout\_width="match\_parent"  android:layout\_height="?attr/actionBarSize"  app:layout\_collapseMode="pin"/>  </com.google.android.material.appbar.CollapsingToolbarLayout>  </com.google.android.material.appbar.AppBarLayout>  <androidx.fragment.app.FragmentContainerView  android:id="@+id/nav\_host\_fragment"  ... />  ...  </LinearLayout>  Next, call setupWithNavController()  from main activity's onCreate method.  override fun onCreate(savedInstanceState: Bundle?) {  setContentView(R.layout.activity\_main)  ...  val layout = findViewById<CollapsingToolbarLayout>(R.id.collapsing\_toolbar\_layout)  val toolbar = findViewById<Toolbar>(R.id.toolbar)  val navHostFragment =  supportFragmentManager.findFragmentById(R.id.nav\_host\_fragment) as NavHostFragment  val navController = navHostFragment.navController  val appBarConfiguration = AppBarConfiguration(navController.graph)  layout.setupWithNavController(toolbar, navController, appBarConfiguration)  }  -------------Action bar-----------------------  To add navigation support to the default action bar,  call setupActionBarWithNavController()  from main activity's onCreate() method.  We need to declare your AppBarConfiguration outside of onCreate(),  since we also use it when overriding onSupportNavigateUp().  private lateinit var appBarConfiguration: AppBarConfiguration  ...  override fun onCreate(savedInstanceState: Bundle?) {  ...  val navHostFragment =  supportFragmentManager.findFragmentById(R.id.nav\_host\_fragment) as NavHostFragment  val navController = navHostFragment.navController  appBarConfiguration = AppBarConfiguration(navController.graph)  setupActionBarWithNavController(navController, appBarConfiguration)  }  Next, override onSupportNavigateUp() to handle Up navigation.  override fun onSupportNavigateUp(): Boolean {  val navController = findNavController(R.id.nav\_host\_fragment)  return navController.navigateUp(appBarConfiguration)  || super.onSupportNavigateUp()  }  -----------Navigation menu items--------  NavigationUI also provides helpers for tying destinations  to menu-driven UI components. NavigationUI contains a helper method, onNavDestinationSelected(), which takes a MenuItem  along with the NavController that hosts the associated destination.  If the id of the MenuItem matches the id of the destination,  the NavController can then navigate to that destination.  <?xml version="1.0" encoding="utf-8"?>  <navigation xmlns:app="http://schemas.android.com/apk/res-auto"  xmlns:tools="http://schemas.android.com/tools"  xmlns:android="http://schemas.android.com/apk/res/android"  ... >  ...  <fragment android:id="@+id/details\_page\_fragment"  android:label="@string/details"  android:name="com.example.android.myapp.DetailsFragment" />  </navigation>  <menu xmlns:android="http://schemas.android.com/apk/res/android">  ...  <item  android:id="@+id/details\_page\_fragment"  android:icon="@drawable/ic\_details"  android:title="@string/details" />  </menu>  If menu was added via the Activity's onCreateOptionsMenu(),  for example, we can associate the menu items with destinations  by overriding the Activity's onOptionsItemSelected()  to call onNavDestinationSelected().  override fun onOptionsItemSelected(item: MenuItem): Boolean {  val navController = findNavController(R.id.nav\_host\_fragment)  return item.onNavDestinationSelected(navController) || super.onOptionsItemSelected(item)  }  -----------navigation drawer--------------  The navigation drawer is a UI panel  that shows your app's main navigation menu.  The drawer appears when the user touches the drawer icon  or when the user swipes a finger from the left edge of the screen.  First. declare a DrawerLayout as the root view.  Inside the DrawerLayout, add a layout for the main UI content  and another view that contains the contents of the navigation drawer.  For example, Let’s say we use a layout uses a DrawerLayout with two child views  A NavHostFragment to contain the main content and a NavigationView  for the contents of the navigation drawer.  <?xml version="1.0" encoding="utf-8"?>  <!-- Use DrawerLayout as root container for activity -->  <androidx.drawerlayout.widget.DrawerLayout xmlns:android="http://schemas.android.com/apk/res/android"  xmlns:app="http://schemas.android.com/apk/res-auto"  android:id="@+id/drawer\_layout"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  android:fitsSystemWindows="true">  <!-- Layout to contain contents of main body of screen (drawer will slide over this) -->  <androidx.fragment.app.FragmentContainerView  android:name="androidx.navigation.fragment.NavHostFragment"  android:id="@+id/nav\_host\_fragment"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent"  app:defaultNavHost="true"  app:navGraph="@navigation/nav\_graph" />  <!-- Container for contents of drawer - use NavigationView to make configuration easier -->  <com.google.android.material.navigation.NavigationView  android:id="@+id/nav\_view"  android:layout\_width="wrap\_content"  android:layout\_height="match\_parent"  android:layout\_gravity="start"  android:fitsSystemWindows="true" />  </androidx.drawerlayout.widget.DrawerLayout>  Next, connect the DrawerLayout to navigation graph  by passing it to AppBarConfiguration  val appBarConfiguration = AppBarConfiguration(navController.graph, drawerLayout)  Next, in the main activity class, call setupWithNavController()  from main activity's onCreate() method.  override fun onCreate(savedInstanceState: Bundle?) {  setContentView(R.layout.activity\_main)  ...  val navHostFragment =  supportFragmentManager.findFragmentById(R.id.nav\_host\_fragment) as NavHostFragment  val navController = navHostFragment.navController  findViewById<NavigationView>(R.id.nav\_view)  .setupWithNavController(navController)  }  ---------Bottom navigation-------------  NavigationUI can also handle bottom navigation.  When a user selects a menu item,  the NavController calls onNavDestinationSelected()  and automatically updates the selected item in the bottom navigation bar.  How to Bottom navigation?  1. first define the bar in your main activity.  <LinearLayout>  ...  <androidx.fragment.app.FragmentContainerView  android:id="@+id/nav\_host\_fragment"  ... />  <com.google.android.material.bottomnavigation.BottomNavigationView  android:id="@+id/bottom\_nav"  app:menu="@menu/menu\_bottom\_nav" />  </LinearLayout>  2. Next, in main activity class, call setupWithNavController()  from main activity's onCreate() method.  override fun onCreate(savedInstanceState: Bundle?) {  setContentView(R.layout.activity\_main)  ...  val navHostFragment =  supportFragmentManager.findFragmentById(R.id.nav\_host\_fragment) as NavHostFragment  val navController = navHostFragment.navController  findViewById<BottomNavigationView>(R.id.bottom\_nav)  .setupWithNavController(navController)  }  -----------communicate/pass data in navigation  We can use Argument-based listeners.  We can also use arguments with default values within the navigation graph,  navController.addOnDestinationChangedListener { \_, \_, arguments ->  appBar.isVisible = arguments?.getBoolean("ShowAppBar", false) == true  } | </end> |
| <hitle> | App deep linking | <chare> | 1 | <pext> | We can set up Android App Links to take users  to a link's specific content directly in the app  bypassing the app-selection dialog,  also known as the disambiguation dialog.  Because Android App Links leverage HTTP URLs and association  with a website, users who don't have the target app installed  go directly to content on the app site.  -------------types of links/linking-------------  1. Deep links  2. Web links  3. Android App Links  ------------ Deep links------------  Deep links are URIs of any scheme that take users directly  to a specific part of your app.  To create deep links, add intent filters to drive users  to the right activity in your app.  <activity  android:name=".MyMapActivity"  android:exported="true"  ...>  <intent-filter>  <action android:name="android.intent.action.VIEW" />  <category android:name="android.intent.category.DEFAULT" />  <category android:name="android.intent.category.BROWSABLE" />  <data android:scheme="geo" />  </intent-filter>  </activity>  When the user clicks a deep link, a disambiguation dialog might appear.  This dialog allows the user to select one of multiple apps,  including your app, that can handle the given deep link.  ---------------Web links-------------  Web links are deep links that use the HTTP and HTTPS schemes.  <intent-filter>  <action android:name="android.intent.action.VIEW" />  <category android:name="android.intent.category.DEFAULT" />  <category android:name="android.intent.category.BROWSABLE" />  <data android:scheme="http" />  <data android:host="myownpersonaldomain.com" />  </intent-filter>  -------------Android App Links---------------  Android App Links are web links that use the HTTP and HTTPS schemes  and contain the autoVerify attribute.  This attribute allows your app to designate itself as the default handler  of a given type of link.  So when the user clicks on an Android App Link,  your app opens immediately if it's installed.  <intent-filter android:autoVerify="true">  <action android:name="android.intent.action.VIEW" />  <category android:name="android.intent.category.DEFAULT" />  <category android:name="android.intent.category.BROWSABLE" />  <data android:scheme="http" />  <data android:scheme="https" />  <data android:host="myownpersonaldomain.com" />  </intent-filter>  ----------Android App Links Deep Linking pros--------------  1. Secure and specific:  Android App Links use HTTP URLs that link to a website domain you own,  so no other app can use your links.  One of the requirements for Android App Links is  that you verify ownership of your domain  through one of our website association methods.  2. Seamless user experience:  Since Android App Links use a single HTTP URL for the same content  on your website and in your app, users who don’t have the app installed  simply go to your website instead of the app.  3. Android Instant Apps support:  With Android Instant Apps, your users can run your Android app  without installing it.  4. Engage users from Google Search:  Users directly open specific content in your app  by clicking a URL from Google in a mobile browser,  in the Google Search app, in screen search on Android,  or through Google Assistant.  -----------App Deep Linking Read incoming data--------------------  Once the system starts your activity through an intent filter,  we can use data provided by the Intent to determine  what you need to render.  We can call the getData() and getAction() methods to retrieve the data  and action associated with the incoming Intent.  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  setContentView(R.layout.main)  val action: String? = intent?.action  val data: Uri? = intent?.data  } | </end> |
| <hitle> | swipe views with tabs ViewPager2 | <chare> | 1 | <pext> | ------------Implement swipe views-----------  We can create swipe views using ViewPager.  To set up your layout with ViewPager,  add the <ViewPager2> element to your XML layout.  <androidx.viewpager.widget.ViewPager  xmlns:android="http://schemas.android.com/apk/res/android"  android:id="@+id/pager"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent" />  To insert child views that represent each page, hook this layout to a PagerAdapter.  --------------------- types of hook PagerAdapter------------  1. FragmentPagerAdapter:  use this when navigating between a small,  fixed number of sibling screens.  2. FragmentStatePagerAdapter:  use this when paging across an unknown number of pages.  3. FragmentStatePagerAdapter  It optimizes memory usage by destroying fragments as the user navigates away.  ------------ TabLayout-------------  A TabLayout provides a way to display tabs horizontally.  When used with a ViewPager, a TabLayout provides a familiar interface  for navigating between pages in a swipe view.  -----------howto Add tabs using a TabLayout/ViewPager2---------  1. To include a TabLayout in a ViewPager, add a <TabLayout> element  inside the <ViewPager> element.  <androidx.viewpager.widget.ViewPager  xmlns:android="http://schemas.android.com/apk/res/android"  android:id="@+id/pager"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent">  <com.google.android.material.tabs.TabLayout  android:id="@+id/tab\_layout"  android:layout\_width="match\_parent"  android:layout\_height="wrap\_content" />  </androidx.viewpager.widget.ViewPager>  2. Use setupWithViewPager() to link the TabLayout to the ViewPager.  The individual tabs in the TabLayout are automatically populated  with the page titles from the PagerAdapter.  class CollectionDemoFragment : Fragment() {  ...  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  val tabLayout = view.findViewById(R.id.tab\_layout)  tabLayout.setupWithViewPager(viewPager)  }  ...  }  class DemoCollectionPagerAdapter(fm: FragmentManager) : FragmentStatePagerAdapter(fm) {  override fun getCount(): Int = 4  override fun getPageTitle(position: Int): CharSequence {  return "OBJECT ${(position + 1)}"  }  ...  } | </end> |
| <hitle> | UpdatingRecords | <chare> | 1 | <pext> | UpdatingRecords  UpdatingRecords  UpdatingRecords  UpdatingRecords  UpdatingRecords  UpdatingRecords  UpdatingRecords | </end> |
| <hitle> | fragment | <chare> | 1 | <pext> | In Android, the fragment is the part of Activity that represents a portion  of the User Interface(UI) on the screen.  It is the modular section of the android activity that is very helpful  in creating UI designs that are flexible in nature and auto-adjustable  based on the device screen size.  ---------why fragment-----------  1. modularity  2. re-usability  3. adaptability  ---------important points fragment-----------  1. fragment cannot live alone they require at least one activity  Which can be the host of multiple fragments.  2. an activity can have a single or multiple fragments.  3. fragment can have fragment inside like child fragment.  4. We cannot have an activity inside a fragment  5. fragment has its own layout and behaviours.  6. fragment has its own lifecycle methods.  7. we can add or remove fragment while activity is running.  8. we can combine multiple fragment in a single activity  to create multi-pane UI.  9. At least one activity require to be host of fragment.  10. We can share data or we are able to send data from fragment  to another fragment, from fragment to activity.  11. we can have a fragment inside another fragment  which is as child fragment.  ------types of fragments---------  Single Fragment  List Fragment  Fragment Transaction  -------add() vs replace() fragment--------  So add() method keeps adding fragments on top  of the previous fragment in the FragmentContainer.  But replace() method clears all the previous fragment from container  and then add it in FragmentContainer.  -------pass data communicate between fragments--------  1. By using interface.  Make an Interface in your FragmentA.  Implement the Interface of the FragmentA in your Activity.  Call the Interface method from your Activity.  In the Activity, call FragmentB to do the required changes.  2. By using Shared ViewModel concept.  -------pass data communicate between fragment activity--------  1. By using interface  To allow a Fragment to communicate up to its Activity,  we can define an interface in the Fragment class and implement it  within the Activity.  The Fragment captures the interface implementation  during its onAttach() lifecycle method and then can call  the Interface methods to communicate with the Activity.  2. By using Shared ViewModel concept.  ------------Shared ViewModel------------  We can create SharedViewModel using the same owner.  The shared view model is the normal view model only  but it’s created by using the same owner. | </end> |
| <hitle> | fragment pros | <chare> | 2 | <pext> | 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. | </end> |
| <hitle> | fragment pros | <chare> | 2 | <pext> | 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. | </end> |
| <hitle> | fragment lifecycle | <chare> | 2 | <pext> | 1. onAttach() The very first method to be called  when the fragment has been associated with the activity.  This method executes only once during the lifetime of a fragment.  2. onCreate() This method initializes the fragment  by adding all the required attributes and components.  3. onCreateView() System calls this method  to create the user interface of the fragment.  The root of the fragment’s layout  is returned as the View component by this method to draw the UI.  4. onActivityCreated() It indicates that the activity has been created  in which the fragment exists.  View hierarchy of the fragment also instantiated before this function call.  5. onStart() The system invokes this method  to make the fragment visible on the user’s device.  6. onResume() This method is called to make the visible fragment interactive.  7. onPause() It indicates that the user is leaving the fragment.  System calls this method to commit the changes made to the fragment.  8. onStop() Method to terminate the functioning and visibility of fragments  from the user’s screen.  9. onDestroyView() System calls this method to clean up all kinds of resources  as well as view hierarchy associated with the fragment.  10. onDestroy() It is called to perform the final clean-up  of the fragment’s state and its lifecycle.  11. onDetach() The system executes this method  to disassociate the fragment from its host activity. | </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 Two2 ways | <chare> | 2 | <pext> | We can add fragment to the activity's view hierarchy either by  1. defining the fragment in the activity's layout file  2. 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" />  ---------------Add fragment Steps---------------  1. We have to call supportFragmentManager to get the instance  of the FragmentManager class.  2. We will get the fragmentTransaction instance  we will call beginTransaction() method on fragmentManager  3. To add a new fragment, we can call add()/replace() method  on the fragmentTransaction class.  4. In the end, we have to call commit() method to save the changes.  Only after we call commit() the transaction completed.    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> | dependency 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> |
| <hitle> | Testing with FragmentFactory | <chare> | 3 | <pext> | In a single activity architecture, we should test fragments in isolation  using the FragmentScenario class.  Since we cannot rely on the custom onCreate logic of the activity,  we can instead pass the FragmentFactory in as an argument  to fragments test.  // Inside your test  val dessertRepository = mock(DessertsRepository::class.java)  launchFragment<DessertsFragment>(factory = MyFragmentFactory(dessertRepository)).onFragment {  // Test Fragment logic  } | </end> |
| <hitle> | Fragment transactions | <chare> | 2 | <pext> | At runtime, a FragmentManager can add, remove, replace,  and perform other actions with fragments in response to user interaction.  Each set of fragment changes that wecommit is called a transaction,  and we can specify what to do inside the transaction  using the APIs provided by the FragmentTransaction class.  We can group multiple actions into a single transaction  For example, a transaction can add or replace multiple fragments.  This grouping can be useful for when you have multiple sibling fragments  displayed on the same screen, such as with split views.  We can save each transaction to a back stack  managed by the FragmentManager,  allowing the user to navigate backward through the fragment changes  similar to navigating backward through activities.  We can get an instance of FragmentTransaction  from the FragmentManager by calling beginTransaction().  val fragmentManager = ...  val fragmentTransaction = fragmentManager.beginTransaction()  The final call on each FragmentTransaction must commit the transaction.  The commit() call signals to the FragmentManager that all operations  have been added to the transaction.  val fragmentManager = ...  // The fragment-ktx module provides a commit block that automatically  // calls beginTransaction and commit for you.  fragmentManager.commit {  // Add operations here  } | </end> |
| <hitle> | reordering fragment state changes | <chare> | 3 | <pext> | Each FragmentTransaction should use setReorderingAllowed(true).  supportFragmentManager.commit {  ...  setReorderingAllowed(true)  }  For behavior compatibility, the reordering flag is not enabled by default.  It is required, however, to allow FragmentManager  to properly execute FragmentTransaction,  particularly when it operates on the back stack  and runs animations and transitions.  Enabling the flag ensures that if multiple transactions are executed together,  any intermediate fragments,  like ones that are added and then immediately replaced,  we do not go through lifecycle changes or have their animations  or transitions executed.  Note that this flag affects both the initial execution of the transaction  and reversing the transaction with popBackStack(). | </end> |
| <hitle> | FragmentManager Add/remove/replace fragments | <chare> | 3 | <pext> | To add a fragment to a FragmentManager, call add() on the transaction.  This method receives the ID of the container for the fragment,  as well as the class name of the fragment you wish to add.  The added fragment is moved to the RESUMED state.  It is strongly recommended that the container is  a FragmentContainerView that is part of the view hierarchy.  To remove a fragment from the host, call remove(),  passing in a fragment instance that was retrieved  from the fragment manager through findFragmentById()  or findFragmentByTag().  If the fragment's view was previously added to a container,  the view is removed from the container at this point.  The removed fragment is moved to the DESTROYED state.  We can use replace() to replace an existing fragment in a container  with an instance of a new fragment class that we provide.  Calling replace() is equivalent to calling remove() with a fragment  in a container and adding a new fragment to that same container.  // Create new fragment  val fragmentManager = // ...  // Create and commit a new transaction  fragmentManager.commit {  setReorderingAllowed(true)  // Replace whatever is in the fragment\_container view with this fragment  replace<ExampleFragment>(R.id.fragment\_container)  }  (\*It shows how you can replace one fragment with another.  In this example, a new instance of ExampleFragment replaces the fragment, if any, that is currently in the layout container identified by R.id.fragment\_container.\*)  Note: It is strongly recommended to always use fragment operations  that take a Class rather than a fragment instance to ensure  that the same mechanisms for creating the fragment are also used  for restoring the fragment from a saved state.  By default, the changes made in a FragmentTransaction are not added  to the back stack.  To save those changes, we can call addToBackStack()  on the FragmentTransaction. | </end> |
| <hitle> | Fragment Commit() asynchronous | <chare> | 3 | <pext> | Calling commit() doesn't perform the transaction immediately.  Rather, the transaction is scheduled to run on the main UI thread  as soon as it is able to do so.  If necessary, however, we can call commitNow()  to run the fragment transaction on your UI thread immediately.  Note that commitNow is incompatible with addToBackStack.  Alternatively, we can execute all pending FragmentTransactions  submitted by commit() calls that have not yet run  by calling executePendingTransactions().  This approach is compatible with addToBackStack.  For the vast majority of use cases, commit() is all we need. | </end> |
| <hitle> | Operation order FragmentTransaction | <chare> | 3 | <pext> | The order in which you perform operations  within a FragmentTransaction is significant,  particularly when using setCustomAnimations().  This method applies the given animations to all fragment operations  that follow it.  supportFragmentManager.commit {  setCustomAnimations(enter1, exit1, popEnter1, popExit1)  add<ExampleFragment>(R.id.container) // gets the first animations  setCustomAnimations(enter2, exit2, popEnter2, popExit2)  add<ExampleFragment>(R.id.container) // gets the second animations  } | </end> |
| <hitle> | Limit fragment lifecycle | <chare> | 3 | <pext> | FragmentTransactions can affect the lifecycle state  of individual fragments added within the scope of the transaction.  When creating a FragmentTransaction, setMaxLifecycle()  sets a maximum state for the given fragment.  For example, ViewPager2 uses setMaxLifecycle() to limit  the off-screen fragments to the STARTED state. | </end> |
| <hitle> | Show/hide fragment views | <chare> | 3 | <pext> | We can use the FragmentTransaction methods show() and hide()  to show and hide the view of fragments that have been added  to a container.  These methods set the visibility of the fragment's views  without affecting the lifecycle of the fragment.  While you don't need to use a fragment transaction to toggle  the visibility of the views within a fragment, these methods are useful  for cases where you want changes to the visibility state  to be associated with transactions on the back stack. | </end> |
| <hitle> | Attach/detach fragments | <chare> | 3 | <pext> | The FragmentTransaction method detach()  detaches the fragment from the UI, destroying its view hierarchy.  The fragment remains in the same state (STOPPED)  as when it is put on the back stack.  This means that the fragment was removed from the UI  but is still managed by the fragment manager.  The attach() method reattaches a fragment from which  it was previously detached.  This causes its view hierarchy to be recreated, attached to the UI,  and displayed.  As a FragmentTransaction is treated as a single atomic set of operations,  calls to both detach and attach on the same fragment instance  in the same transaction effectively cancel each other out,  thus avoiding the destruction and immediate recreation of the fragment's UI.  We need to use separate transactions,  separated by executePendingOperations() if using commit(),  if we want to detach and then immediately re-attach a fragment. | </end> |
| <hitle> | Communicate/share/pass data with fragments | <chare> | 2 | <pext> | The Fragment library provides two options for communication:  a shared ViewModel and the Fragment Result API.  The recommended option depends on the use case.  To share persistent data with any custom APIs,  we should use a ViewModel.  For a one-time result with data that can be placed in a Bundle,  we should use the Fragment Result API. | </end> |
| <hitle> | Share/pass data ViewModel fragment | <chare> | 3 | <pext> | ViewModel is an ideal choice when we need to share data  between multiple fragments or between fragments and their host activity.  ViewModel objects store and manage UI data.  1. We can share data with the host activity.  2. We can share data between fragments. | </end> |
| <hitle> | Share/pass data fragment with host activity ViewModel | <chare> | 4 | <pext> | Let’s say, we want to toggle a global UI component  based on an interaction within a fragment.  class ItemViewModel : ViewModel() {  private val mutableSelectedItem = MutableLiveData<Item>()  val selectedItem: LiveData<Item> get() = mutableSelectedItem  fun selectItem(item: Item) {  mutableSelectedItem.value = item  }  }  (\*The data being stored is wrapped in a MutableLiveData class.  LiveData is a lifecycle-aware observable data holder class.  MutableLiveData allows its value to be changed. \*)  Both fragment and its host activity can retrieve a shared instance  of a ViewModel with activity scope by passing the activity  into the ViewModelProvider constructor.  The ViewModelProvider handles instantiating the ViewModel  or retrieving it if it already exists.  Both components can observe and modify this data.  class MainActivity : AppCompatActivity() {  // Using the viewModels() Kotlin property delegate from the activity-ktx  // artifact to retrieve the ViewModel in the activity scope  private val viewModel: ItemViewModel by viewModels()  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  viewModel.selectedItem.observe(this, Observer { item ->  // Perform an action with the latest item data  })  }  }  class ListFragment : Fragment() {  // Using the activityViewModels() Kotlin property delegate from the  // fragment-ktx artifact to retrieve the ViewModel in the activity scope  private val viewModel: ItemViewModel by activityViewModels()  // Called when the item is clicked  fun onItemClicked(item: Item) {  // Set a new item  viewModel.selectItem(item)  }  } | </end> |
| <hitle> | Share/pass data between fragments ViewModel | <chare> | 4 | <pext> | Two or more fragments in the same activity can share a ViewModel  using their activity scope to handle this communication.  By sharing the ViewModel in this way,  the fragments do not need to know about each other,  and the activity does not need to do anything  to facilitate the communication.  class ListViewModel : ViewModel() {  val filters = MutableLiveData<Set<Filter>>()  private val originalList: LiveData<List<Item>>() = ...  val filteredList: LiveData<List<Item>> = ...  fun addFilter(filter: Filter) { ... }  fun removeFilter(filter: Filter) { ... }  }  class ListFragment : Fragment() {  // Using the activityViewModels() Kotlin property delegate from the  // fragment-ktx artifact to retrieve the ViewModel in the activity scope  private val viewModel: ListViewModel by activityViewModels()  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  viewModel.filteredList.observe(viewLifecycleOwner, Observer { list ->  // Update the list UI  }  }  }  class FilterFragment : Fragment() {  private val viewModel: ListViewModel by activityViewModels()  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  viewModel.filters.observe(viewLifecycleOwner, Observer { set ->  // Update the selected filters UI  }  }  fun onFilterSelected(filter: Filter) = viewModel.addFilter(filter)  fun onFilterDeselected(filter: Filter) = viewModel.removeFilter(filter)  }  Fragments use their host activity as the scope for the ViewModelProvider.  Because the fragments use the same scope,  they receive the same instance of the ViewModel,  which enables them to communicate back and forth. | </end> |
| <hitle> | Share/pass data between a parent and child fragment ViewModel | <chare> | 4 | <pext> | To share data between the parent fragment and its child fragments,  we can use the parent fragment as the ViewModel scope.  class ListFragment: Fragment() {  // Using the viewModels() Kotlin property delegate from the fragment-ktx  // artifact to retrieve the ViewModel  private val viewModel: ListViewModel by viewModels()  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  viewModel.filteredList.observe(viewLifecycleOwner, Observer { list ->  // Update the list UI  }  }  }  class ChildFragment: Fragment() {  // Using the viewModels() Kotlin property delegate from the fragment-ktx  // artifact to retrieve the ViewModel using the parent fragment's scope  private val viewModel: ListViewModel by viewModels({requireParentFragment()})  ...  } | </end> |
| <hitle> | Scoping a ViewModel to the Navigation Graph | <chare> | 4 | <pext> | If we're using the Navigation library, we can also scope a ViewModel  to the lifecycle of a destination's NavBackStackEntry.  For example, the ViewModel could be scoped  to the NavBackStackEntry for the ListFragment.  class ListFragment: Fragment() {  // Using the navGraphViewModels() Kotlin property delegate from the fragment-ktx  // artifact to retrieve the ViewModel using the NavBackStackEntry scope  // R.id.list\_fragment == the destination id of the ListFragment destination  private val viewModel: ListViewModel by navGraphViewModels(R.id.list\_fragment)  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  viewModel.filteredList.observe(viewLifecycleOwner, Observer { item ->  // Update the list UI  }  }  } | </end> |
| <hitle> | pass data Fragment Result API | <chare> | 3 | <pext> | In some cases, we may want to pass a one-time value  between two fragments or between a fragment and its host activity.  For example, you might have a fragment that reads QR codes,  passing the data back to a previous fragment.  Each FragmentManager implements FragmentResultOwner.  This means that a FragmentManager can act as a central store  for fragment results.  This change allows components to communicate with each other  by setting fragment results and listening for those results  without requiring those components to have direct references to each other. | </end> |
| <hitle> | Pass data between fragments result | <chare> | 4 | <pext> | To pass data back to fragment A from fragment B,  first we set a result listener on fragment A,  the fragment that receives the result.  We call setFragmentResultListener() on fragment A's FragmentManager.  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  // Use the Kotlin extension in the fragment-ktx artifact  setFragmentResultListener("requestKey") { requestKey, bundle ->  // We use a String here, but any type that can be put in a Bundle is supported  val result = bundle.getString("bundleKey")  // Do something with the result  }  }  In fragment B, the fragment producing the result,  we must set the result on the same FragmentManager  by using the same requestKey.  We can do so by using the setFragmentResult() API.  button.setOnClickListener {  val result = "result"  // Use the Kotlin extension in the fragment-ktx artifact  setFragmentResult("requestKey", bundleOf("bundleKey" to result))  }  Fragment A then receives the result and executes the listener callback  once the fragment is STARTED.  We can have only a single listener and result for a given key.  If we call setFragmentResult() more than once for the same key,  and if the listener is not STARTED, the system replaces  any pending results with your updated result.  If we set a result without a corresponding listener to receive it,  the result is stored in the FragmentManager  until you set a listener with the same key.  Once a listener receives a result  and fires the onFragmentResult() callback, the result is cleared.  This behavior has two major implications:  1. Fragments on the back stack do not receive results  until they have been popped and are STARTED.  2. If a fragment listening for a result is STARTED when the result is set,  the listener's callback is then fired immediately. | </end> |
| <hitle> | Test fragment pass data results | <chare> | 4 | <pext> | We can use FragmentScenario to test calls to setFragmentResult()  and setFragmentResultListener().  We need to create a scenario for the fragment under test  by using launchFragmentInContainer or launchFragment,  and then manually call the method that is not being tested.  To test setFragmentResultListener(), we need to create a scenario  with the fragment that makes the call to setFragmentResultListener().  Next, we can call setFragmentResult() directly, and verify the result.  @Test  fun testFragmentResultListener() {  val scenario = launchFragmentInContainer<ResultListenerFragment>()  scenario.onFragment { fragment ->  val expectedResult = "result"  fragment.parentFragmentManager.setFragmentResult("requestKey", bundleOf("bundleKey" to expectedResult))  assertThat(fragment.result).isEqualTo(expectedResult)  }  }  class ResultListenerFragment : Fragment() {  var result : String? = null  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  // Use the Kotlin extension in the fragment-ktx artifact  setFragmentResultListener("requestKey") { requestKey, bundle ->  result = bundle.getString("bundleKey")  }  }  }  To test setFragmentResult(), we can create a scenario with the fragment  that makes the call to setFragmentResult().  Next, we can call setFragmentResultListener() directly, and verify the result.  @Test  fun testFragmentResult() {  val scenario = launchFragmentInContainer<ResultFragment>()  lateinit var actualResult: String?  scenario.onFragment { fragment ->  fragment.parentFragmentManager  .setFragmentResultListener("requestKey") { requestKey, bundle ->  actualResult = bundle.getString("bundleKey")  }  }  onView(withId(R.id.result\_button)).perform(click())  assertThat(actualResult).isEqualTo("result")  }  class ResultFragment : Fragment(R.layout.fragment\_result) {  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  view.findViewById(R.id.result\_button).setOnClickListener {  val result = "result"  // Use the Kotlin extension in the fragment-ktx artifact  setFragmentResult("requestKey", bundleOf("bundleKey" to result))  }  }  } | </end> |
| <hitle> | Pass data result between parent and child fragments | <chare> | 4 | <pext> | To pass a result from a child fragment to a parent,  the parent fragment should use getChildFragmentManager()  instead of getParentFragmentManager()  when calling setFragmentResultListener().  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  // We set the listener on the child fragmentManager  childFragmentManager.setFragmentResultListener("requestKey") { key, bundle ->  val result = bundle.getString("bundleKey")  // Do something with the result  }  }  The child fragment sets the result on its FragmentManager.  The parent then receives the result once the fragment is STARTED:  button.setOnClickListener {  val result = "result"  // Use the Kotlin extension in the fragment-ktx artifact  setFragmentResult("requestKey", bundleOf("bundleKey" to result))  } | </end> |
| <hitle> | Pass data Receive fragment results host activity | <chare> | 4 | <pext> | To receive a fragment result in the host activity,  we can set a result listener on the fragment manager  using getSupportFragmentManager().  class MainActivity : AppCompatActivity() {  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  supportFragmentManager  .setFragmentResultListener("requestKey", this) { requestKey, bundle ->  // We use a String here, but any type that can be put in a Bundle is supported  val result = bundle.getString("bundleKey")  // Do something with the result  }  }  } | </end> |
| <hitle> | UpdatingRecords | <chare> | 3 | <pext> | UpdateingRecord  UpdateingRecord  UpdateingRecord  UpdateingRecord  UpdateingRecord | </end> |