|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | View Binding | <chare> | 1 | <pext> | View binding is a feature that allows you to more easily write  code that interacts with views.  Once view binding is enabled in a module, it generates a binding class  for each XML layout file present in that module. An instance of a binding class contains direct references to all views  that have an ID in the corresponding layout. | </end> |
| <hitle> | View Binding gradle | <chare> | 2 | <pext> | View binding is enabled on a module by module basis.  To enable view binding in a module,  set the viewBinding build option to true in the module-level build.gradle file.  android {  ...  buildFeatures {  viewBinding true  }  } | </end> |
| <hitle> | ignore View Binding | <chare> | 2 | <pext> | If you want a layout file to be ignored while generating binding classes,  add the tools:viewBindingIgnore="true" attribute  to the root view of that layout file.  <LinearLayout  ...  tools:viewBindingIgnore="true" >  ...  </LinearLayout> | </end> |
| <hitle> | howto view binding | <chare> | 2 | <pext> | If view binding is enabled for a module, a binding class is generated  for each XML layout file that the module contains.  Each binding class contains references to the root view  and all views that have an ID.  The name of the binding class is generated by converting  the name of the XML file to Pascal case and adding the word "Binding"  to the end.  <LinearLayout ... >  <TextView android:id="@+id/name" />  <ImageView android:cropToPadding="true" />  <Button android:id="@+id/button"  android:background="@drawable/rounded\_button" /> </LinearLayout>  Let's say we have TestLayout.xml and the generated binding class is called  TestLayoutBinding. If a button inside the layout has android:id property,  the binding class has a field with the assigned id name to refer it.  If not, there's no reference to it in the binding class. Every binding class also includes a getRoot() method,  providing a direct reference for the root view of the corresponding layout file.  The getRoot() method in the the binding class returns the root layout view  like LinearLayout, ConstraintLayout whatever we have in the root in xml. | </end> |
| <hitle> | view binding in activities | <chare> | 3 | <pext> | <starting>view binding in activities</heading> To set up an instance of the binding class for use with an activity,  We need to do something in the activity's onCreate() method: 1. Call the static inflate() method included in the generated binding class.  This creates an instance of the binding class for the activity to use. 2. Get a reference to the root view by either calling the getRoot() method  or using Kotlin property syntax. 3. Pass the root view to setContentView() to make it the active view  on the screen.  private lateinit var binding: ResultProfileBinding override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  binding = ResultProfileBinding.inflate(layoutInflater)  val view = binding.root  setContentView(view) } (\*We can now use the instance of the binding class to reference  any of the views.\*)  binding.name.text = viewModel.name binding.button.setOnClickListener { viewModel.userClicked() } </ending> | </end> |
| <hitle> | view binding in fragments | <chare> | 3 | <pext> | <starting>view binding in fragments</heading> To set up an instance of the binding class for use with a fragment,  We need to do something in the fragment's onCreateView() method: 1. Call the static inflate() method included in the generated binding class.  This creates an instance of the binding class for the fragment to use. 2. Get a reference to the root view by either calling the getRoot() method  or using Kotlin property syntax. 3. Return the root view from the onCreateView() method  to make it the active view on the screen. There's onething we have to know is, Fragments outlive their views. So, we need to make sure you clean up any references  to the binding class instance in the fragment's onDestroyView() method.  private var \_binding: ResultProfileBinding? = null // This property is only valid between onCreateView and // onDestroyView. private val binding get() = \_binding!! override fun onCreateView(  inflater: LayoutInflater,  container: ViewGroup?,  savedInstanceState: Bundle? ): View? {  \_binding = ResultProfileBinding.inflate(inflater, container, false)  val view = binding.root  return view } override fun onDestroyView() {  super.onDestroyView()  \_binding = null } (\*We can now use the instance of the binding class to reference  any of the views.\*)  binding.name.text = viewModel.name binding.button.setOnClickListener { viewModel.userClicked() } </ending> | </end> |
| <hitle> | tools:viewBindingType for different configurations | <chare> | 3 | <pext> | <starting>tools:viewBindingType for different configurations</heading> When we declare views across multiple configurations,  occasionally it makes sense to use a different view type  depending on the particular layout.  # in res/layout/example.xml <TextView android:id="@+id/user\_bio" /> # in res/layout-land/example.xml <EditText android:id="@+id/user\_bio" />  View binding supports a tools:viewBindingType attribute,  allowing you to tell the compiler what type to use in the generated code.  # in res/layout/example.xml (unchanged) <TextView android:id="@+id/user\_bio" /> # in res/layout-land/example.xml <EditText android:id="@+id/user\_bio" tools:viewBindingType="TextView" />  For example, suppose we have two layouts  where one contains a BottomNavigationView and  another contains a NavigationRailView.  Both classes extend NavigationBarView,  which contains most of the implementation details.  If your code doesn't need to know exactly which subclass is present  in the current layout, you can use tools:viewBindingType to set  the generated type to NavigationBarView in both layouts:  # in res/layout/navigation\_example.xml <BottomNavigationView android:id="@+id/navigation" tools:viewBindingType="NavigationBarView" /> # in res/layout-w720/navigation\_example.xml <NavigationRailView android:id="@+id/navigation" tools:viewBindingType="NavigationBarView" /> </ending> | </end> |
| <hitle> | view binding pros, view-binding vs fidnViewById() | <chare> | 2 | <pext> | <starting>view binding pros, view-binding vs fidnViewById()</heading> 1. Null safety: Since view binding creates direct references to views,  there's no risk of a null pointer exception due to an invalid view ID. Additionally, when a view is only present in some configurations of a layout,  the field containing its reference in the binding class is marked with @Nullable. 2. Type safety: The fields in each binding class have types  matching the views they reference in the XML file.  This means that there's no risk of a class cast exception.  Using view binding, incompatibilities between the layout and the code  will result in the build failing at compile time rather than at runtime. </ending> | </end> |
| <hitle> | view-binding vs data-binding | <chare> | 2 | <pext> | <starting>view-binding vs data-binding</heading> 1. Faster compilation: View binding requires no annotation processing,  so compile times are faster. 2. Ease of use: View binding does not require specially-tagged XML layout files,  so it is faster to adopt in your apps.  Once you enable view binding in a module, it applies  to all of that module's layouts automatically. 3. View binding doesn't support layout variables or layout expressions,  so it can't be used to declare dynamic UI content straight  from XML layout files. 4. View binding doesn't support two-way data binding. </ending> | </end> |
| <hitle> | view-binding validate | <chare> | 2 | <pext> | View binding is unable to validate the value of this attribute  when generating code.  To avoid compile time and runtime errors,  1. The value must be a class that inherits from android.view.View. 2. The value must be a superclass of the tag it is placed on. \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* <TextView tools:viewBindingType="ImageView" />  <!-- ImageView is not related to TextView -->  <TextView tools:viewBindingType="Button" />  <!-- Button is not a superclass of TextView --> \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. The final type must resolve consistently across all configurations. | </end> |
| <hitle> | Migrate/convert view binding | <chare> | 2 | <pext> | <starting> Migrate/convert view binding </heading> 1. Set the viewBinding build option to true  in the module-level build.gradle file  android {  ...  buildFeatures {  viewBinding true  } }  If the app does not use Parcelize features, remove the line  that enables Kotlin Android Extensions.  plugins {  id 'kotlin-android-extensions' }  If the app uses Parcelize features, we have to switch to using  the standalone kotlin-parcelize Gradle plugin.  plugins {  id 'kotlin-parcelize' }  2. Ane then, we need to update activity and fragment classes. 2-1. Remove all imports from kotlinx.android.synthetic. 2-2. Inflate an instance of the generated binding class  for the activity or fragment to use. 2-2-1. For activities, update the activity's onCreate() method  to inflate a view-binding instance.  2-2-2. For fragments, update the fragment's  onCreateView() method to inflate a view-binding instance. 2-3. Change all view references to use the binding class instance  instead of synthetic properties:  // Reference to "name" TextView using synthetic properties. name.text = viewModel.nameString // Reference to "name" TextView using the binding class instance. binding.name.text = viewModel.nameString </ending> | </end> |
| <hitle> | data-binding | <chare> | 1 | <pext> | The Data Binding Library is a support library that allows to bind  UI components in the layouts to data sources in your app using  a declarative format rather than programmatically. | </end> |
| <hitle> | data-binding pros | <chare> | 2 | <pext> | Binding components in the layout file lets you remove  many UI framework calls in your activities, making them  simpler and easier to maintain.  This can also improve your app's performance  and help prevent memory leaks and null pointer exceptions. | </end> |
| <hitle> | data-binding gradle | <chare> | 2 | <pext> | To configure your app to use data binding,  enable the dataBinding build option in your build.gradle file  in the app module.  android {  ...  buildFeatures {  dataBinding true  } } | </end> |
| <hitle> | Howto data-binding | <chare> | 2 | <pext> | Layouts are often defined in activities with code  that calls UI framework methods, like findViewById()  to find a TextView widget and bind it to the userName property  of the viewModel variable.  findViewById<TextView>(R.id.sample\_text).apply {  text = viewModel.userName  }  By Data Binding Library, we can use of @{} syntax  in the assignment expression to find a widget and bind it  to the property of the viewModel variable.  <TextView  android:text="@{viewmodel.userName}" /> | </end> |
| <hitle> | data-binding layout | <chare> | 2 | <pext> | Data binding layout files are slightly different  and start with a root tag of layout followed by a data element  and a view root element.  This view element is what the root would be in a non-binding layout file.  <?xml version="1.0" encoding="utf-8"?>  <layout xmlns:android="http://schemas.android.com/apk/res/android">  <data>  <variable name="user" type="com.example.User"/>  </data>  <LinearLayout  android:orientation="vertical"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent">  <TextView android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="@{user.firstName}"/>  <TextView android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="@{user.lastName}"/>  </LinearLayout>  </layout>  Any variable, like <variable name="user">, within data describes a property  that may be used within this layout. | </end> |
| <hitle> | howto data-binding data | <chare> | 2 | <pext> | A binding class is generated for each layout file.  By default, the name of the class is based on the name of the layout file,  converting it to Pascal case and adding the Binding suffix to it.  For the layout filename activity\_main.xml will be ActivityMainBinding.  This class holds all the bindings from the layout properties,  like the user variable, to the layout's views and knows how to assign values  for the binding expressions.  The recommended method to create the bindings is to do it  while inflating the layout.  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  val binding: ActivityMainBinding = DataBindingUtil.setContentView(  this, R.layout.activity\_main)  binding.user = User("Test", "User")  }  At runtime, the app displays the Test user in the UI.  Alternatively, we can get the view using a LayoutInflater.  val binding: ActivityMainBinding = ActivityMainBinding.inflate(getLayoutInflater())  If we are using data binding items inside a Fragment, ListView,  or RecyclerView adapter, we may need to use the inflate() methods  of the bindings classes or the DataBindingUtil class.  val listItemBinding = ListItemBinding.inflate(layoutInflater, viewGroup, false)  // or  val listItemBinding = DataBindingUtil.inflate(layoutInflater, R.layout.list\_item, viewGroup, false) | </end> |
| <hitle> | data-binding expression | <chare> | 2 | <pext> | Expressions within the layout are written in the attribute properties  using the "@{}" syntax.  <variable name="user" type="com.example.User" />  Now we can use property names of the user variable,  like the TextView text can be set to a property of the user variable.  <TextView android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="@{user.firstName}" />  1. Common expression Mathematical + - / \* % String concatenation + Logical && || Binary & | ^ Unary + - ! ~ Shift >> >>> << Comparison == > < >= <= (Note that < needs to be escaped as &lt;) instanceof Grouping () Literals - character, String, numeric, null Cast Method calls Field access Array access [] Ternary operator ?:  android:text="@{String.valueOf(index + 1)}" android:visibility="@{age > 13 ? View.GONE : View.VISIBLE}" android:transitionName='@{"image\_" + id}'  2. We can’t use this super new Explicit generic invocation  3. Null coalescing operator  The null coalescing operator (??) chooses the left operand if it isn't null  or the right if the former is null.  android:text="@{user.displayName ?? user.lastName}"  This is functionally equivalent to:  android:text="@{user.displayName != null ? user.displayName : user.lastName}"  4. Property references  An expression can reference a property in a class,  which is the same for fields, getters, and ObservableField objects:  android:text="@{user.lastName}"  5. Avoiding null pointer exceptions  Generated data binding code automatically checks for null values and  avoid null pointer exceptions.  For example, in the expression @{user.name}, if user is null,  user.name is assigned its default value of null.  If you reference user.age, where age is of type int, then data binding  uses the default value of 0.  6. View references  We can reference other views in the layout by ID like,  android:text="@{exampleText.text}"  The binding class converts IDs to camel case.  Let’s see an example.  <EditText  android:id="@+id/example\_text"  android:layout\_height="wrap\_content"  android:layout\_width="match\_parent"/>  <TextView  android:id="@+id/example\_output"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="@{exampleText.text}"/>  The TextView view references an EditText view in the same layout.  7. Collections  Common collections, such as arrays, lists, sparse lists, and maps,  can be accessed using the [] operator for convenience.  <data>  <import type="android.util.SparseArray"/>  <import type="java.util.Map"/>  <import type="java.util.List"/>  <variable name="list" type="List&lt;String>"/>  <variable name="sparse" type="SparseArray&lt;String>"/>  <variable name="map" type="Map&lt;String, String>"/>  <variable name="index" type="int"/>  <variable name="key" type="String"/>  </data>  …  android:text="@{list[index]}"  …  android:text="@{sparse[index]}"  …  android:text="@{map[key]}"  Note: For the XML to be syntactically correct,  we have to escape the < characters.  For example: instead of List<String> you have to write List&lt;String>.  We can also refer to a value in the map using the object.key notation.  For example, @{map[key]} can be replaced with @{map.key}.  8. String literals  We can use single quotes to surround the attribute value, which allows  to use double quotes in the expression  android:text='@{map["firstName"]}'  It is also possible to use double quotes to surround the attribute value.  When doing so, string literals should be surrounded with back quotes `.  android:text="@{map[`firstName`]}"  9. Resources  We can reference app resources.  android:padding="@{large? @dimen/largePadding : @dimen/smallPadding}"  We can evaluate format strings and plurals by providing parameters:  android:text="@{@string/nameFormat(firstName, lastName)}"  android:text="@{@plurals/banana(bananaCount)}"  We can pass property references and view references as resource parameters:  android:text="@{@string/example\_resource(user.lastName, exampleText.text)}"  When a plural takes multiple parameters, you must pass all parameters:  Have an orange  Have %d oranges  android:text="@{@plurals/orange(orangeCount, orangeCount)}"  Some resources require explicit type evaluation.  String[] - @array -@stringArray  int[] - @array - @intArray  TypedArray - @array - @typedArray  Animator - @animator - @animator  StateListAnimator - @animator - @stateListAnimator  color int - @color - @color  ColorStateList - @color - @colorStateList | </end> |
| <hitle> | data-binding expression event handling | <chare> | 3 | <pext> | Data binding allows you to write expression handling events  that are dispatched from the views like, the onClick() method.  Attribute names are determined by the name of the listener method  with a few exceptions like, View.OnClickListener has a method onClick(),  so the attribute for this event is android:onClick. | </end> |
| <hitle> | data-binding specialized event handler | <chare> | 4 | <pext> | There are some specialized event handlers for the click event  that need an attribute other than android:onClick to avoid a conflict.  1. SearchView  Listener setter --- setOnSearchClickListener(View.OnClickListener)  Attribute --- android:onSearchClick  2. ZoomControls  Listener setter --- setOnZoomInClickListener(View.OnClickListener)  Attribute --- android:onZoomIn  3. ZoomControls  Listener setter --- setOnZoomOutClickListener(View.OnClickListener)  Attribute --- android:onZoomOut | </end> |
| <hitle> | data-binding event handler mechanism | <chare> | 4 | <pext> | We can use two mechanisms to handle an event.  1. Method references: In the expressions, we can reference methods  that conform to the signature of the listener method.  When an expression evaluates to a method reference, Data binding wraps  the method reference and owner object in a listener, and sets  that listener on the target view. If the expression evaluates to null,  Data binding doesn't create a listener and sets a null listener instead.  2. Listener bindings: These are lambda expressions that are evaluated  when the event happens. Data binding always creates a listener,  which it sets on the view. When the event is dispatched,  the listener evaluates the lambda expression. | </end> |
| <hitle> | Method references vs Listener binding | <chare> | 4 | <pext> | The actual listener implementation is created when the data is bound,  not when the event is triggered.  If we need to evaluate the expression when the event happens,  s we need to use listener binding.  In method references, the parameters of the method must match  the parameters of the event listener.  In listener bindings, only your return value must match  the expected return value of the listener (unless it is expecting void). | </end> |
| <hitle> | Method references mechanism | <chare> | 4 | <pext> | Events can be bound to handler methods directly, similar to the way  android:onClick can be assigned to a method in an activity.  To assign an event to its handler, we can use a normal binding expression,  with the value being the method name to call.  class MyHandlers {  fun onClickFriend(view: View) { ... }  }  The binding expression can assign the click listener for a view  to the onClickFriend() method.  <?xml version="1.0" encoding="utf-8"?>  <layout xmlns:android="http://schemas.android.com/apk/res/android">  <data>  <variable name="handlers" type="com.example.MyHandlers"/>  <variable name="user" type="com.example.User"/>  </data>  <LinearLayout  android:orientation="vertical"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent">  <TextView android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="@{user.firstName}"  android:onClick="@{handlers::onClickFriend}"/>  </LinearLayout>  </layout> | </end> |
| <hitle> | Method references mechanism pros | <chare> | 4 | <pext> | One major advantage compared to the View onClick attribute is that  the expression is processed at compile time, so if the method doesn't exist  or its signature is incorrect, you receive a compile time error. | </end> |
| <hitle> | Listener bindings mechanism | <chare> | 4 | <pext> | Listener bindings are binding expressions that run when an event happens.  They are similar to method references,  but they run arbitrary data binding expressions.  In method references, the parameters of the method must match  the parameters of the event listener.  In listener bindings, only return value must match the expected return value  of the listener (unless it is expecting void).  Let’s see an example with Presenter class and onSaveClick() method:  class Presenter {  fun onSaveClick(task: Task){}  }  We can bind the click event to the onSaveClick() method like this,  <?xml version="1.0" encoding="utf-8"?>  <layout xmlns:android="http://schemas.android.com/apk/res/android">  <data>  <variable name="task" type="com.android.example.Task" />  <variable name="presenter" type="com.android.example.Presenter" />  </data>  <LinearLayout android:layout\_width="match\_parent" android:layout\_height="match\_parent">  <Button android:layout\_width="wrap\_content" android:layout\_height="wrap\_content"  android:onClick="@{() -> presenter.onSaveClick(task)}" />  </LinearLayout>  </layout>  When a callback is used in an expression, data binding automatically creates  the necessary listener and registers it for the event.  When the view fires the event, data binding evaluates the given expression.  As in regular binding expressions, you still get null and thread safety  of data binding while these listener expressions are being evaluated.  Listener bindings provide two choices for listener parameters:  1. We can either ignore all parameters to the method or name all of them.  2. If we need the name the parameters, we can use them in the expression.  android:onClick="@{(view) -> presenter.onSaveClick(task)}"  Or if we want to use the parameter in the expression,  class Presenter {  fun onSaveClick(view: View, task: Task){}  }  android:onClick="@{(theView) -> presenter.onSaveClick(theView, task)}"  We can use a lambda expression with more than one parameter:  class Presenter {  fun onCompletedChanged(task: Task, completed: Boolean){}  }  <CheckBox android:layout\_width="wrap\_content" android:layout\_height="wrap\_content"  android:onCheckedChanged="@{(cb, isChecked) -> presenter.completeChanged(task, isChecked)}" />  If the event returns a value whose type isn't void, the expressions  must return the same type of value as well. For example, if the long click event, the expression should return a boolean.  class Presenter {  fun onLongClick(view: View, task: Task): Boolean { }  }  android:onLongClick="@{(theView) -> presenter.onLongClick(theView, task)}"  If the expression cannot be evaluated due to null objects, data binding  returns the default value for that type.  For example, null for reference types, 0 for int, false for boolean, etc.  If we need to use an expression with a predicate (for example, ternary),  we can use void as a symbol.  android:onClick="@{(v) -> v.isVisible() ? doSomething() : void}" | </end> |
| <hitle> | Avoid complex listeners | <chare> | 4 | <pext> | Listener expressions are very powerful and can make the code  very easy to read.  On the other hand, listeners containing complex expressions  make the layouts hard to read and maintain.  These expressions should be as simple as passing available data  from your UI to your callback method.  We need to implement any business logic inside the callback method  that we invoked from the listener expression. | </end> |
| <hitle> | Data Binding expression import | <chare> | 3 | <pext> | Imports allow us to easily reference classes inside the layout file,  just like in managed code.  Zero or more import elements may be used inside the data element.  <data>  <import type="android.view.View"/>  </data>  Importing the View class allows to reference it from the binding expressions.  For example to reference the VISIBLE and GONE constants of the View class,  We can do like this.  <TextView  android:text="@{user.lastName}"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:visibility="@{user.isAdult ? View.VISIBLE : View.GONE}"/> | </end> |
| <hitle> | Data Binding expression Type aliases | <chare> | 3 | <pext> | When there are class name conflicts, one of the classes may be renamed  to an alias.  <import type="android.view.View"/>  <import type="com.example.real.estate.View"  alias="Vista"/>  We can use Vista to reference the com.example.real.estate.View  and View may be used to reference android.view.View  within the layout file. | </end> |
| <hitle> | Data Binding expression import other class | <chare> | 3 | <pext> | Imported types can be used as type references in variables and expressions.  <data>  <import type="com.example.User"/>  <import type="java.util.List"/>  <variable name="user" type="User"/>  <variable name="userList" type="List&lt;User>"/>  </data>  (\*For example, User and List used as the type of a variable.\*)  We can also use the imported types to cast part of an expression.  <TextView  android:text="@{((User)(user.connection)).lastName}"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"/>  (It casts the connection property to a type of User.)  Imported types may also be used when referencing static fields  and methods in expressions.  <data>  <import type="com.example.MyStringUtils"/>  <variable name="user" type="com.example.User"/>  </data>  …  <TextView  android:text="@{MyStringUtils.capitalize(user.lastName)}"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"/>  (\*It imports the MyStringUtils class and references its capitalize method.\*)  Just as in managed code, java.lang.\* is imported automatically. | </end> |
| <hitle> | Data Binding expression Variables | <chare> | 3 | <pext> | We can use multiple variable elements inside the data element.  Each variable element describes a property that may be set on the layout  to be used in binding expressions within the layout file.  <data>  <import type="android.graphics.drawable.Drawable"/>  <variable name="user" type="com.example.User"/>  <variable name="image" type="Drawable"/>  <variable name="note" type="String"/>  </data>  (\*It declares the user, image, and note variables.\*)  The variable types are inspected at compile time, so if a variable implements  Observable or is an observable collection, that should be reflected  in the type. If the variable is a base class or interface that doesn't implement  the Observable interface, the variables are not observed.  When there are different layout files for various configurations  (for example, landscape or portrait), the variables are combined.  There must not be conflicting variable definitions between these layout files.  The generated binding class has a setter and getter for each  of the described variables.  The variables take the default managed code values  until the setter is called—null for reference types, 0 for int, false for boolean, etc.  A special variable named context is generated for use in binding expressions  as needed. The value for context is the Context object from the root View's  getContext() method. The context variable is overridden  by an explicit variable declaration with that name. | </end> |
| <hitle> | Data Binding expression Includes | <chare> | 3 | <pext> | Variables may be passed into an included layout's binding  from the containing layout by using the app namespace  and the variable name in an attribute.  <?xml version="1.0" encoding="utf-8"?>  <layout xmlns:android="http://schemas.android.com/apk/res/android"  xmlns:bind="http://schemas.android.com/apk/res-auto">  <data>  <variable name="user" type="com.example.User"/>  </data>  <LinearLayout  android:orientation="vertical"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent">  <include layout="@layout/name"  bind:user="@{user}"/>  <include layout="@layout/contact"  bind:user="@{user}"/>  </LinearLayout>  </layout>  (Included user variables from the name.xml and contact.xml layout files.)  Data binding doesn't support include as a direct child of a merge element.  <?xml version="1.0" encoding="utf-8"?>  <layout xmlns:android="http://schemas.android.com/apk/res/android"  xmlns:bind="http://schemas.android.com/apk/res-auto">  <data>  <variable name="user" type="com.example.User"/>  </data>  <merge><!-- Doesn't work -->  <include layout="@layout/name"  bind:user="@{user}"/>  <include layout="@layout/contact"  bind:user="@{user}"/>  </merge>  </layout>  (The layout isn't supported.) | </end> |
| <hitle> | Data Binding observable data objects | <chare> | 2 | <pext> | Observability refers to the capability of an object to notify others  about changes in its data.  The Data Binding Library allows to make objects, fields, or collections  observable.  Any plain-old object can be used for data binding,  but modifying the object doesn't automatically cause the UI to update.  Data binding can be used to give data objects the ability to notify  other objects, known as listeners, when its data changes.  There are three different types of observable classes: objects, fields,  and collections.  When one of these observable data objects is bound to the UI and  a property of the data object changes, the UI is updated automatically. | </end> |
| <hitle> | Observable fields | <chare> | 3 | <pext> | Some work is involved in creating classes that implement  the Observable interface, which could not be worth the effort if the classes  only have a few properties.  In this case, we can use the generic Observable class and those are:  ObservableBoolean  ObservableByte  ObservableChar  ObservableShort  ObservableInt  ObservableLong  ObservableFloat  ObservableDouble  ObservableParcelable  Observable fields are self-contained observable objects  that have a single field.  The primitive versions avoid boxing and unboxing during access operations.  To use this mechanism, We need to create a public final property  in the Java or a read-only property in Kotlin.  class User {  val firstName = ObservableField<String>()  val lastName = ObservableField<String>()  val age = ObservableInt()  }  To access the field value, we can use the set() and get() accessor methods  or use Kotlin property syntax.  user.firstName = "Google"  val age = user.age | </end> |
| <hitle> | Observable collections | <chare> | 3 | <pext> | Observable collections allow access to these structures by using a key.  The ObservableArrayMap class is useful when the key is a reference type,  such as String.  ObservableArrayMap<String, Any>().apply {  put("firstName", "Google")  put("lastName", "Inc.")  put("age", 17)  }  (In the layout, the map can be found using the string keys.)  <data>  <import type="android.databinding.ObservableMap"/>  <variable name="user" type="ObservableMap&lt;String, Object&gt;"/>  </data>  …  <TextView  android:text="@{user.lastName}"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"/>  <TextView  android:text="@{String.valueOf(1 + (Integer)user.age)}"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"/>  The ObservableArrayList class is useful when the key is an integer.  ObservableArrayList<Any>().apply {  add("Google")  add("Inc.")  add(17)  }  (In the layout, the list can be accessed through the indexes.)  <data>  <import type="android.databinding.ObservableList"/>  <import type="com.example.my.app.Fields"/>  <variable name="user" type="ObservableList&lt;Object&gt;"/>  </data>  …  <TextView  android:text='@{user[Fields.LAST\_NAME]}'  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"/>  <TextView  android:text='@{String.valueOf(1 + (Integer)user[Fields.AGE])}'  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"/> | </end> |
| <hitle> | Observable objects | <chare> | 3 | <pext> | A class that implements the Observable interface allows the registration  of listeners that want to be notified of property changes  from the observable object.  The Observable interface has a mechanism to add and remove listeners,  but we must decide when notifications are sent.  To make development easier,  the Data Binding Library provides the BaseObservable class,  which implements the listener registration mechanism.  The data class that implements BaseObservable is responsible for notifying  when the properties change.  This is done by assigning a Bindable annotation to the getter  and calling the notifyPropertyChanged() method in the setter.  class User : BaseObservable() {  @get:Bindable  var firstName: String = ""  set(value) {  field = value  notifyPropertyChanged(BR.firstName)  }  @get:Bindable  var lastName: String = ""  set(value) {  field = value  notifyPropertyChanged(BR.lastName)  }  }  Data binding generates a class named BR in the module package  which contains the IDs of the resources used for data binding.  The Bindable annotation generates an entry in the BR class file  during compilation.  If the base class for data classes cannot be changed,  the Observable interface can be implemented using  a PropertyChangeRegistry object to register and notify listeners efficiently. | </end> |
| <hitle> | Lifecycle-aware objects | <chare> | 3 | <pext> | The layouts can also bind to data binding sources that automatically  notify the UI about changes in the data.  That way, the bindings are lifecycle-aware and are only triggered  when the UI is visible on the screen.  Data binding currently supports StateFlow and LiveData. | </end> |
| <hitle> | Howto StateFlow | <chare> | 4 | <pext> | If the app uses Kotlin with coroutines, we can use StateFlow objects  as the data binding source.  To use a StateFlow object with the binding class, we must specify  a lifecycle owner to define the scope of the StateFlow object.  class ViewModelActivity : AppCompatActivity() {  override fun onCreate(savedInstanceState: Bundle?) {  // Inflate view and obtain an instance of the binding class.  val binding: UserBinding = DataBindingUtil.setContentView(this, R.layout.user)  // Specify the current activity as the lifecycle owner.  binding.lifecycleOwner = this  }  }  (It specifies the activity as the lifecycle owner after the binding class is instantiated.)  Data binding works seamlessly with ViewModel objects.  We can use StateFlow and ViewModel together.  class ScheduleViewModel : ViewModel() {  private val \_username = MutableStateFlow<String>("")  val username: StateFlow<String> = \_username  init {  viewModelScope.launch {  \_username.value = Repository.loadUserName()  }  }  }  In the layout, we need to assign the properties and methods  of the ViewModel object to the corresponding views using binding expressions.  <TextView  android:id="@+id/name"  android:text="@{viewmodel.username}" />  The UI is automatically updated whenever the user's name value changes. | </end> |
| <hitle> | Disable StateFlow | <chare> | 4 | <pext> | For apps that use Kotlin and AndroidX, StateFlow support is  automatically included with data binding.  This means that the coroutines dependency is automatically included  in the app if the dependency is not already available.  We can option out of this functionality by changing build.gradle file.  android {  ...  dataBinding {  addKtx = false  }  }  Alternatively, We can disable it globally by changing gradle.properties file.  android.defaults.databinding.addKtx = false | </end> |
| <hitle> | Generated binding classes | <chare> | 2 | <pext> | The Data Binding Library generates binding classes that are used to access  the layout's variables and views.  The generated binding class links the layout variables with the views  within the layout.  The name and package of the binding class can be customized.  All generated binding classes inherit from the ViewDataBinding class.  A binding class is generated for each layout file.  By default, the name of the class is based on the name of the layout file,  converting it to Pascal case and adding the Binding suffix to it.  If activity\_main.xml, so the corresponding generated class is  ActivityMainBinding.  This class holds all the bindings from the layout properties  to the layout's views and knows how to assign values  for the binding expressions. | </end> |
| <hitle> | Create a binding object | <chare> | 3 | <pext> | The binding object is created immediately after inflating the layout  to ensure that the view hierarchy isn't modified  before it binds to the views with expressions within the layout.  The most common method to bind the object to a layout is to use  the static methods on the binding class.  We can inflate the view hierarchy and bind the object to it by using  the inflate() method of the binding class.  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  val binding: MyLayoutBinding = MyLayoutBinding.inflate(layoutInflater)  setContentView(binding.root)  }  There is an alternate version of the inflate() method that takes  a ViewGroup object in addition to the LayoutInflater object.  val binding: MyLayoutBinding = MyLayoutBinding.inflate(getLayoutInflater(), viewGroup, false)  If the layout was inflated using a different mechanism,  it can be bound separately.  val binding: MyLayoutBinding = MyLayoutBinding.bind(viewRoot)  Sometimes the binding type cannot be known in advance.  In such cases, the binding can be created using the DataBindingUtil class.  val viewRoot = LayoutInflater.from(this).inflate(layoutId, parent, attachToParent)  val binding: ViewDataBinding? = DataBindingUtil.bind(viewRoot)  If we are using data binding items inside a Fragment, ListView,  or RecyclerView adapter, we may need to use the inflate() methods  of the bindings classes or the DataBindingUtil class.  val listItemBinding = ListItemBinding.inflate(layoutInflater, viewGroup, false)  // or  val listItemBinding = DataBindingUtil.inflate(layoutInflater, R.layout.list\_item, viewGroup, false) | </end> |
| <hitle> | Views with IDs | <chare> | 3 | <pext> | The Data Binding Library creates a immutable field in the binding class  for each view that has an ID in the layout.  <layout xmlns:android="http://schemas.android.com/apk/res/android">  <data>  <variable name="user" type="com.example.User"/>  </data>  <LinearLayout  android:orientation="vertical"  android:layout\_width="match\_parent"  android:layout\_height="match\_parent">  <TextView android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="@{user.firstName}"  android:id="@+id/firstName"/>  <TextView android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  android:text="@{user.lastName}"  android:id="@+id/lastName"/>  </LinearLayout>  </layout>  (The Data Binding Library creates the firstName and lastName fields of type TextView.)  The library extracts the views including the IDs from the view hierarchy  in a single pass.  This mechanism can be faster than calling the findViewById() method  for every view in the layout.  IDs aren't as necessary as they are without data binding,  but there are still some instances  where access to views is still necessary from code. | </end> |
| <hitle> | Variables | <chare> | 3 | <pext> | The Data Binding Library generates accessor methods for each variable  declared in the layout.  <data>  <import type="android.graphics.drawable.Drawable"/>  <variable name="user" type="com.example.User"/>  <variable name="image" type="Drawable"/>  <variable name="note" type="String"/>  </data>  (The layout generates setter and getter methods in the binding class for the user, image, and note variables) | </end> |
| <hitle> | ViewStubs | <chare> | 3 | <pext> | Unlike normal views, ViewStub objects start off as an invisible view.  When they either are made visible or are explicitly told to inflate,  they replace themselves in the layout by inflating another layout.  Because the ViewStub essentially disappears from the view hierarchy,  the view in the binding object must also disappear to allow to be claimed  by garbage collection.  Because the views are final, a ViewStubProxy object takes the place  of the ViewStub in the generated binding class, giving you access  to the ViewStub when it exists and also access to the inflated view hierarchy  when the ViewStub has been inflated.  When inflating another layout, a binding must be established  for the new layout. Therefore, the ViewStubProxy must listen  to the ViewStub OnInflateListener and establish the binding when required.  Since only one listener can exist at a given time, the ViewStubProxy allows  you to set an OnInflateListener, which it calls after establishing the binding. | </end> |
| <hitle> | Immediate Binding | <chare> | 3 | <pext> | When a variable or observable object changes, the binding is scheduled  to change before the next frame.  There are times, however, when binding must be executed immediately.  To force execution, we can use the executePendingBindings() method. | </end> |
| <hitle> | Dynamic Variables | <chare> | 3 | <pext> | The specific binding class isn't known.  For example, a RecyclerView.Adapter operating against arbitrary layouts  doesn't know the specific binding class.  It still must assign the binding value during the call to the  onBindViewHolder() method.  All layouts that the RecyclerView binds to have an item variable.  The BindingHolder object has a getBinding() method returning  the ViewDataBinding base class.  override fun onBindViewHolder(holder: BindingHolder, position: Int) {  item: T = items.get(position)  holder.binding.setVariable(BR.item, item);  holder.binding.executePendingBindings();  } | </end> |
| <hitle> | Data binding background thread | <chare> | 3 | <pext> | We can change the data model in a background thread  as long as it isn't a collection.  Data binding localizes each variable / field during evaluation  to avoid any concurrency issues. | </end> |
| <hitle> | Custom binding class names | <chare> | 3 | <pext> | By default, a binding class is generated based on the name of the layout file,  starting with an uppercase letter, removing underscores ( \_ ),  capitalizing the following letter, and suffixing the word Binding.  The class is placed in a databinding package under the module package.  For example, the layout file contact\_item.xml generates  the ContactItemBinding class.  If the module package is com.example.my.app, then the binding class  is placed in the com.example.my.app.databinding package.  Binding classes may be renamed or placed in different packages  by adjusting the class attribute of the data element.  <data class="ContactItem">  …  </data>  (It generates the ContactItem binding class in the databinding package in the current module.)  We can generate the binding class in a different package  by prefixing the class name with a period.  <data class=".ContactItem">  …  </data>  (It generates the binding class in the module package.)  We can also use the full package name where you want the binding class  to be generated.  <data class="com.example.ContactItem">  …  </data>  (It creates the ContactItem binding class in the com.example package.) | </end> |
| <hitle> | Data Binding adapters | <chare> | 2 | <pext> | Binding adapters are responsible for making  the appropriate framework calls to set values.  For example, setting a property value like calling the setText() method. Another example is setting an event listener  like calling the setOnClickListener() method.  The Data Binding Library allows to specify the method called to set a value,  provide our own binding logic, and specify the type of the returned object  by using adapters. | </end> |
| <hitle> | Setting attribute values | <chare> | 3 | <pext> | Whenever a bound value changes, the generated binding class must call  a setter method on the view with the binding expression.  We can allow the Data Binding Library to automatically determine  the method, explicitly declare the method, or provide custom logic  to select a method. | </end> |
| <hitle> | Automatic method selection | <chare> | 4 | <pext> | For an attribute named example, the library automatically tries  to find the method setExample(arg) that accepts compatible types  as the argument. The namespace of the attribute isn't considered,  only the attribute name and type are used when searching for a method.  For example, given the android:text="@{user.name}" expression,  the library looks for a setText(arg) method that accepts the type  returned by user.getName(). If the return type of user.getName() is String,  the library looks for a setText() method that accepts a String argument.  If the expression returns an int instead, the library searches for  a setText() method that accepts an int argument.  The expression must return the correct type, we can cast the return value  if necessary.  Data binding works even if no attribute exists with the given name.  We can then create attributes for any setter by using data binding.  For example, the support class DrawerLayout doesn't have any attributes,  but plenty of setters.  <android.support.v4.widget.DrawerLayout  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"  app:scrimColor="@{@color/scrim}"  app:drawerListener="@{fragment.drawerListener}">  (It automatically uses the setScrimColor(int) and  setDrawerListener(DrawerListener) methods as the setter  for the app:scrimColor and app:drawerListener attributes, respectively.) | </end> |
| <hitle> | Specify a custom method name | <chare> | 4 | <pext> | Some attributes have setters that don't match by name.  In these situations, an attribute may be associated with the setter using  the BindingMethods annotation.  The annotation is used with a class and can contain multiple  BindingMethod annotations, one for each renamed method.  Binding methods are annotations that can be added to any class in your app.  @BindingMethods(value = [  BindingMethod(  type = android.widget.ImageView::class,  attribute = "android:tint",  method = "setImageTintList")])  (The android:tint attribute is associated with the  setImageTintList(ColorStateList) method, not with the setTint() method.)  Most of the time, we don't need to rename setters  in Android framework classes.  The attributes already have implemented using the name convention  for to automatically find matching methods. | </end> |
| <hitle> | attributes custom binding logic | <chare> | 2 | <pext> | Some attributes need custom binding logic.  For example, there is no associated setter  for the android:paddingLeft attribute.  Instead, the setPadding(left, top, right, bottom) method is provided.  A static binding adapter method with the BindingAdapter annotation  allows to customize how a setter for an attribute is called.  The attributes of the Android framework classes already have BindingAdapter annotations created.  @BindingAdapter("android:paddingLeft")  fun setPaddingLeft(view: View, padding: Int) {  view.setPadding(padding,  view.getPaddingTop(),  view.getPaddingRight(),  view.getPaddingBottom())  }  (It shows the binding adapter for the paddingLeft attribute.)  The parameter types are important.  The first parameter determines the type of the view that is associated  with the attribute.  The second parameter determines the type  accepted in the binding expression for the given attribute.  Binding adapters are useful for other types of customization.  For example, a custom loader can be called from a worker thread  to load an image.  The binding adapters that you define override the default adapters  provided by the Android framework when there is a conflict.  We can also have adapters that receive multiple attributes.  @BindingAdapter("imageUrl", "error")  fun loadImage(view: ImageView, url: String, error: Drawable) {  Picasso.get().load(url).error(error).into(view)  }  We can use the adapter in the layout.  Surrounding the resource with @{} makes it a valid binding expression.  The Data Binding Library ignores custom namespaces  for matching purposes.  <ImageView app:imageUrl="@{venue.imageUrl}" app:error="@{@drawable/venueError}" />  (@drawable/venueError refers to a resource in your app.  The adapter is called if both imageUrl and error are used  for an ImageView object and imageUrl is a string and error is a Drawable.  If you want the adapter to be called when any of the attributes is set, you can set the optional requireAll flag of the adapter to false.)  @BindingAdapter(value = ["imageUrl", "placeholder"], requireAll = false)  fun setImageUrl(imageView: ImageView, url: String?, placeHolder: Drawable?) {  if (url == null) {  imageView.setImageDrawable(placeholder);  } else {  MyImageLoader.loadInto(imageView, url, placeholder);  }  }  The binding adapters override the default data binding adapters  when there is a conflict.  Binding adapter methods may optionally take the old values  in their handlers.  A method taking old and new values should declare all old values  for the attributes first, followed by the new values.  @BindingAdapter("android:paddingLeft")  fun setPaddingLeft(view: View, oldPadding: Int, newPadding: Int) {  if (oldPadding != newPadding) {  view.setPadding(newPadding,  view.getPaddingTop(),  view.getPaddingRight(),  view.getPaddingBottom())  }  }  Event handlers may only be used with interfaces or abstract classes  with one abstract method.  @BindingAdapter("android:onLayoutChange")  fun setOnLayoutChangeListener(  view: View,  oldValue: View.OnLayoutChangeListener?,  newValue: View.OnLayoutChangeListener?  ) {  if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.HONEYCOMB) {  if (oldValue != null) {  view.removeOnLayoutChangeListener(oldValue)  }  if (newValue != null) {  view.addOnLayoutChangeListener(newValue)  }  }  }  We can use this event handler in the layout.  <View android:onLayoutChange="@{() -> handler.layoutChanged()}"/>  When a listener has multiple methods, it must be split into multiple listeners.  For example, View.OnAttachStateChangeListener has two methods:  onViewAttachedToWindow(View) and  onViewDetachedFromWindow(View).  The library provides two interfaces to differentiate the attributes and  handlers for them:  // Translation from provided interfaces in Java:  @TargetApi(Build.VERSION\_CODES.HONEYCOMB\_MR1)  interface OnViewDetachedFromWindow {  fun onViewDetachedFromWindow(v: View)  }  @TargetApi(Build.VERSION\_CODES.HONEYCOMB\_MR1)  interface OnViewAttachedToWindow {  fun onViewAttachedToWindow(v: View)  }  Because changing one listener can also affect the other,  we need an adapter that works for either attribute or for both.  We can set requireAll to false in the annotation to specify  that not every attribute must be assigned a binding expression.  @BindingAdapter(  "android:onViewDetachedFromWindow",  "android:onViewAttachedToWindow",  requireAll = false  )  fun setListener(view: View, detach: OnViewDetachedFromWindow?, attach: OnViewAttachedToWindow?) {  if (Build.VERSION.SDK\_INT >= Build.VERSION\_CODES.HONEYCOMB\_MR1) {  val newListener: View.OnAttachStateChangeListener?  newListener = if (detach == null && attach == null) {  null  } else {  object : View.OnAttachStateChangeListener {  override fun onViewAttachedToWindow(v: View) {  attach.onViewAttachedToWindow(v)  }  override fun onViewDetachedFromWindow(v: View) {  detach.onViewDetachedFromWindow(v)  }  }  }  val oldListener: View.OnAttachStateChangeListener? =  ListenerUtil.trackListener(view, newListener, R.id.onAttachStateChangeListener)  if (oldListener != null) {  view.removeOnAttachStateChangeListener(oldListener)  }  if (newListener != null) {  view.addOnAttachStateChangeListener(newListener)  }  }  }  (This example is slightly more complicated than normal  because the View class uses the addOnAttachStateChangeListener() and  removeOnAttachStateChangeListener() methods  instead of a setter method for OnAttachStateChangeListener.)    The android.databinding.adapters.ListenerUtil class helps keep track  of the previous listeners so that they may be removed  in the binding adapter.  By annotating the interfaces OnViewDetachedFromWindow and  OnViewAttachedToWindow with @TargetApi(VERSION\_CODES.HONEYCOMB\_MR1),  the data binding code generator knows  that the listener should only be generated  when running on Android 3.1 (API level 12) and higher,  the same version supported by  the addOnAttachStateChangeListener() method. | </end> |
| <hitle> | Automatic object conversion | <chare> | 2 | <pext> | When an Object is returned from a binding expression,  the library chooses the method used to set the value of the property.  The Object is cast to a parameter type of the chosen method.  This behavior is convenient in apps using the ObservableMap class  to store data.  <TextView  android:text='@{userMap["lastName"]}'  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content" />  (We can also refer to a value in the map using the object.key notation.  For example, @{userMap["lastName"]} can be replaced with  @{userMap.lastName}.  The userMap object in the expression returns a value, which is  automatically cast to the parameter type  found in the setText(CharSequence) method used to set the value  of the android:text attribute.)    If the parameter type is ambiguous, we must cast the return type  in the expression. | </end> |
| <hitle> | Custom object conversions | <chare> | 2 | <pext> | In some situations, a custom conversion is required between specific types.  For example, the android:background attribute of a view  expects a Drawable, but the color value specified is an integer.  <View  android:background="@{isError ? @color/red : @color/white}"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"/>  (It shows an attribute that expects a Drawable,  but an integer is provided instead.  Whenever a Drawable is expected and an integer is returned,  the int should be converted to a ColorDrawable.)    The conversion can be done using a static method  with a BindingConversion annotation.  @BindingConversion  fun convertColorToDrawable(color: Int) = ColorDrawable(color)  However, the value types provided in the binding expression  must be consistent.  <View  android:background="@{isError ? @drawable/error : @color/white}"  android:layout\_width="wrap\_content"  android:layout\_height="wrap\_content"/>  (We cannot use different types in the same expression.) | </end> |
| <hitle> | Data bind layout views to Architecture Components | <chare> | 1 | <pext> | The AndroidX library includes the Architecture Components,  which we can use to design robust, testable, and maintainable apps.  The Data Binding Library works seamlessly  with the Architecture Components to further simplify the development  of the UI.  The layouts in the app can bind to the data in the Architecture Components,  which already help us to manage the UI controllers lifecycle  and notify about changes in the data. | </end> |
| <hitle> | LiveData to notify the UI about data changes | <chare> | 2 | <pext> | We can use LiveData objects as the data binding source  to automatically notify the UI about changes in the data.  Unlike objects that implement Observable, such as observable fields,  LiveData objects know about the lifecycle of the observers  subscribed to the data changes.  In Android Studio version 3.1 and higher, we can replace observable fields  with LiveData objects in the data binding code.  To use a LiveData object with the binding class, we need to specify  a lifecycle owner to define the scope of the LiveData object.  class ViewModelActivity : AppCompatActivity() {  override fun onCreate(savedInstanceState: Bundle?) {  // Inflate view and obtain an instance of the binding class.  val binding: UserBinding = DataBindingUtil.setContentView(this, R.layout.user)  // Specify the current activity as the lifecycle owner.  binding.setLifecycleOwner(this)  }  }  (It shows the activity as the lifecycle owner after the binding class  has been instantiated.)  We can use a ViewModel component to bind the data to the layout.  In the ViewModel component, we can use the LiveData object  to transform the data or merge multiple data sources.  class ScheduleViewModel : ViewModel() {  val userName: LiveData  init {  val result = Repository.userName  userName = Transformations.map(result) { result -> result.value }  }  }  (It shows how to transform the data in the ViewModel.) | </end> |
| <hitle> | ViewModel to manage UI-related data | <chare> | 2 | <pext> | The Data Binding Library works seamlessly with ViewModel components,  which expose the data that the layout observes and reacts to its changes.  Using ViewModel components with the Data Binding Library allows  to move UI logic out of the layouts and into the components,  which are easier to test.  The Data Binding Library ensures that the views are bound and unbound  from the data source when needed.  Most of the remaining work consists in making sure that we're exposing  the correct data.  To use the ViewModel component with the Data Binding Library,  we must instantiate the component,  which inherits from the ViewModel class,  obtain an instance of the binding class,  and assign your ViewModel component to a property in the binding class.  class ViewModelActivity : AppCompatActivity() {  override fun onCreate(savedInstanceState: Bundle?) {  // Obtain the ViewModel component.  val userModel: UserModel by viewModels()  // Inflate view and obtain an instance of the binding class.  val binding: UserBinding = DataBindingUtil.setContentView(this, R.layout.user)  // Assign the component to a property in the binding class.  binding.viewmodel = userModel  }  }  (It shows how to use the component with the library.)  In the layout, we need to assign the properties and methods  of ViewModel component to the corresponding views  using binding expressions.  <CheckBox  android:id="@+id/rememberMeCheckBox"  android:checked="@{viewmodel.rememberMe}"  android:onCheckedChanged="@{() -> viewmodel.rememberMeChanged()}" /> | </end> |
| <hitle> | Observable ViewModel | <chare> | 2 | <pext> | We can use a ViewModel component that implements the Observable  to notify other app components about changes in the data.  There are situations where we need to use a ViewModel component  that implements the Observable interface over using LiveData objects,  even if we lose the lifecycle management capabilities of LiveData.  Using a ViewModel component that implements Observable gives us  more control over the binding adapters in the app.  For example, Using a ViewModel component that implements Observable  gives us more control over the notifications when data changes,  it also allows to specify a custom method to set the value of an attribute  in two-way data binding.  To implement an observable ViewModel component, we must create  a class that inherits from the ViewModel class and implements  the Observable interface.  We can provide custom logic  when an observer subscribes or unsubscribes to notifications  using the addOnPropertyChangedCallback() and  removeOnPropertyChangedCallback() methods.  We can also provide custom logic that runs when properties change  in the notifyPropertyChanged() method.  /\*\*  \* A ViewModel that is also an Observable,  \* to be used with the Data Binding Library.  \*/  open class ObservableViewModel : ViewModel(), Observable {  private val callbacks: PropertyChangeRegistry = PropertyChangeRegistry()  override fun addOnPropertyChangedCallback(  callback: Observable.OnPropertyChangedCallback) {  callbacks.add(callback)  }  override fun removeOnPropertyChangedCallback(  callback: Observable.OnPropertyChangedCallback) {  callbacks.remove(callback)  }  /\*\*  \* Notifies observers that all properties of this instance have changed.  \*/  fun notifyChange() {  callbacks.notifyCallbacks(this, 0, null)  }  /\*\*  \* Notifies observers that a specific property has changed. The getter for the  \* property that changes should be marked with the @Bindable annotation to  \* generate a field in the BR class to be used as the fieldId parameter.  \*  \* @param fieldId The generated BR id for the Bindable field.  \*/  fun notifyPropertyChanged(fieldId: Int) {  callbacks.notifyCallbacks(this, fieldId, null)  }  }  (It shows how to implement an observable ViewModel.) | </end> |
| <hitle> | Two-way data binding | <chare> | 1 | <pext> | Using one-way data binding, we can set a value on an attribute  and set a listener that reacts to a change in that attribute.  <CheckBox  android:id="@+id/rememberMeCheckBox"  android:checked="@{viewmodel.rememberMe}"  android:onCheckedChanged="@{viewmodel.rememberMeChanged}"  />  Two-way Data Binding is a technique of binding objects to the XML layouts  so that the layout can send data to the binding object.  This is compared to a “traditional” or “one-way” Data Binding setup,  where data would only move from the binding object to the layout.  <CheckBox  android:id="@+id/rememberMeCheckBox"  android:checked="@={viewmodel.rememberMe}"  />  The @={} notation, which importantly includes the "=" sign,  receives data changes to the property and listen to user updates  at the same time.  In order to react to changes in the backing data,  we can make the layout variable an implementation of Observable,  usually BaseObservable, and use a @Bindable annotation.  class LoginViewModel : BaseObservable {  // val data = ...  @Bindable  fun getRememberMe(): Boolean {  return data.rememberMe  }  fun setRememberMe(value: Boolean) {  // Avoids infinite loops.  if (data.rememberMe != value) {  data.rememberMe = value  // React to the change.  saveData()  // Notify observers of a new value.  notifyPropertyChanged(BR.remember\_me)  }  }  }  (Because the bindable property's getter method is called getRememberMe(),  the property's corresponding setter method automatically uses  the name setRememberMe().) | </end> |
| <hitle> | Two-way data binding using custom attributes | <chare> | 2 | <pext> | There are two-way data binding implementations  for the most common two-way attributes and change listeners.  If we want to use two-way data binding with custom attributes,  we need to work with the @InverseBindingAdapter and  @InverseBindingMethod annotations.  For example, if we want to enable two-way data binding  on a "time" attribute in a custom view called MyView,  we need to complete two steps.  1. We need to annotate the method that sets the initial value  and updates when the value changes using @BindingAdapter.  @BindingAdapter("time")  @JvmStatic fun setTime(view: MyView, newValue: Time) {  // Important to break potential infinite loops.  if (view.time != newValue) {  view.time = newValue  }  }  2. We need to annotate the method that reads the value from the view  using @InverseBindingAdapter.  @InverseBindingAdapter("time")  @JvmStatic fun getTime(view: MyView) : Time {  return view.getTime()  }  At this point, data binding knows what to do when the data changes  (it calls the method annotated with @BindingAdapter)  and what to call when the view attribute changes  (it calls the InverseBindingListener).  However, it doesn't know when or how the attribute changes.  For that, we need to set a listener on the view.  It can be a custom listener associated with the custom view,  or it can be a generic event, such as a loss of focus or a text change.  We need to add the @BindingAdapter annotation to the method  that sets the listener for changes on the property.  @BindingAdapter("app:timeAttrChanged")  @JvmStatic fun setListeners(  view: MyView,  attrChange: InverseBindingListener  ) {  // Set a listener for click, focus, touch, etc.  }  The listener includes an InverseBindingListener as a parameter.  We use the InverseBindingListener to tell the data binding system  that the attribute has changed.  The system can then start calling the method annotated using  @InverseBindingAdapter, and so on.  In practice, this listener includes some non-trivial logic, including listeners  for one-way data binding. For an example, we can see the adapter  for the text attribute change, TextViewBindingAdapter.  Every two-way binding generates a synthetic event attribute.  This attribute has the same name as the base attribute, but it includes  the suffix "AttrChanged".  The synthetic event attribute allows the library to create a method  annotated using @BindingAdapter to associate the event listener  to the appropriate instance of View. | </end> |
| <hitle> | Two-way data binding Converters | <chare> | 2 | <pext> | If the variable that's bound to a View object needs to be formatted,  translated, or changed somehow before being displayed,  it's possible to use a Converter object.  For example, we can see EditText object that shows a date.  <EditText  android:id="@+id/birth\_date"  android:text="@={Converter.dateToString(viewmodel.birthDate)}"  />  (The viewmodel.birthDate attribute contains a value of type Long,  so it needs to be formatted by using a converter.)  Because a two-way expression is being used, there also needs to be  an inverse converter to let the library know how to convert  the user-provided string back to the backing data type, in this case Long.  This process is done by adding the @InverseMethod annotation  to one of the converters and have this annotation reference the inverse converter.  object Converter {  @InverseMethod("stringToDate")  @JvmStatic fun dateToString(  view: EditText, oldValue: Long,  value: Long  ): String {  // Converts long to String.  }  @JvmStatic fun stringToDate(  view: EditText, oldValue: String,  value: String  ): Long {  // Converts String to long.  }  } | </end> |
| <hitle> | two-way data binding Infinite loops | <chare> | 2 | <pext> | We need to be careful not to introduce infinite loops  when using two-way data binding.  When the user changes an attribute, the method annotated  using @InverseBindingAdapter is called, and the value is assigned  to the backing property.  This, in turn, would call the method annotated using @BindingAdapter,  which would trigger another call to the method annotated using  @InverseBindingAdapter, and so on.  For this reason, it's important to break possible infinite loops by comparing  new and old values in the methods annotated using @BindingAdapter. | </end> |
| <hitle> | Two-way attributes | <chare> | 2 | <pext> | The platform provides built-in support for two-way data binding  when we use some attributes.  Class Attribute(s) Binding adapter  1. AdapterView  Attribute: android:selectedItemPosition, android:selection  Binding adapter: AdapterViewBindingAdapter  2. CalendarView  Attribute: android:date,  Binding adapter: CalendarViewBindingAdapter  3. CompoundButton  Attribute: android:checked  Binding adapter: CompoundButtonBindingAdapter  4. DatePicker  Attribute: android:year, android:month, android:day  Binding adapter: DatePickerBindingAdapter  5. NumberPicker  Attribute: android:value  Binding adapter: NumberPickerBindingAdapter  6. RadioButton  Attribute: android:checkedButton  Binding adapter: RadioGroupBindingAdapter  7. RatingBar  Attribute: android:rating  Binding adapter: RatingBarBindingAdapter  8. SeekBar  Attribute: android:progress  Binding adapter: SeekBarBindingAdapter  9. TabHost  Attribute: android:currentTab  Binding adapter: TabHostBindingAdapter  10. TextView  Attribute: android:text  Binding adapter: TextViewBindingAdapter  11. TimePicker  Attribute: android:hour, android:minute  Binding adapter: TimePickerBindingAdapter | </end> |
| <hitle> | Lifecycle-Aware Components | <chare> | 1 | <pext> | Lifecycle-aware components perform actions in response to a change  in the lifecycle status of another component, such as activities and  fragments. These components help us produce better-organized,  and often lighter-weight code, that is easier to maintain.  There are few lifecycle aware android architecture components:  1. ViewModel: Helps to create, store and retrieve data and communicates  with other components belonging to the same lifecycle.  2. Lifecycle Owner: It’s an interface implemented by activity and fragment,  to observe changes to the lifecycle of owners.  3. LiveData: Allows observing changes in data across diff components  in the same lifecycle.  The androidx.lifecycle package provides classes and interfaces that let us  build lifecycle-aware components which are components that can  automatically adjust their behavior based on the current lifecycle state  of an activity or fragment. | </end> |
| <hitle> | Lifecycle-Aware Components Lifecycle | <chare> | 2 | <pext> | Lifecycle is a class that holds the information about the lifecycle state  of a component (like an activity or a fragment) and allows other objects  to observe this state.  Lifecycle uses two main enumerations to track the lifecycle status  for its associated component:  1. Event  The lifecycle events that are dispatched from the framework  and the Lifecycle class.  These events map to the callback events in activities and fragments.  2. State  The current state of the component tracked by the Lifecycle object.  <https://developer.android.com/static/images/topic/libraries/architecture/lifecycle-states.svg> | </end> |
| <hitle> | Lifecycle-Aware Components Lifecycle Event | <chare> | 3 | <pext> | The lifecycle events that are dispatched from the framework  and the Lifecycle class.  These events map to the callback events in activities and fragments. | </end> |
| <hitle> | Lifecycle-Aware Components Lifecycle State | <chare> | 3 | <pext> | The current state of the component tracked by the Lifecycle object. | </end> |
| <hitle> | monitor lifecycle status/State | <chare> | 3 | <pext> | A class can monitor the component's lifecycle status by implementing  DefaultLifecycleObserver and overriding corresponding methods  such as onCreate, onStart, etc.  Then we can add an observer by calling the addObserver() method  of the Lifecycle class and passing an instance of the observer.  class MyObserver : DefaultLifecycleObserver {  override fun onResume(owner: LifecycleOwner) {  connect()  }  override fun onPause(owner: LifecycleOwner) {  disconnect()  }  }  myLifecycleOwner.getLifecycle().addObserver(MyObserver())  (myLifecycleOwner object implements the LifecycleOwner interface,  which is explained in the following section.) | </end> |
| <hitle> | Lifecycle-Aware Components LifecycleOwner | <chare> | 2 | <pext> | LifecycleOwner is a single method interface  that denotes that the class has a Lifecycle.  It has one method, getLifecycle(), which must be implemented by the class.  This interface abstracts the ownership of a Lifecycle from individual classes,  such as Fragment and AppCompatActivity, and allows writing components  that work with them.  Any custom application class can implement the LifecycleOwner interface.  Components that implement DefaultLifecycleObserver work seamlessly  with components that implement LifecycleOwner because an owner  can provide a lifecycle, which an observer can register to watch.  class MyActivity : AppCompatActivity() {  private lateinit var myLocationListener: MyLocationListener  override fun onCreate(...) {  myLocationListener = MyLocationListener(this, lifecycle) { location ->  // update UI  }  Util.checkUserStatus { result ->  if (result) {  myLocationListener.enable()  }  }  }  }  (We can make the MyLocationListener class implement  DefaultLifecycleObserver and then initialize it with the activity's Lifecycle  in the onCreate() method. This allows the MyLocationListener class to be  self-sufficient, meaning that the logic to react to changes in lifecycle status  is declared in MyLocationListener instead of the activity.  Having the individual components store their own logic makes  the activities and fragments logic easier to manage.)  A common use case is to avoid invoking certain callbacks if the Lifecycle  isn't in a good state right now. For example, if the callback runs  a fragment transaction after the activity state is saved,  it would trigger a crash, so we would never want to invoke that callback.  The Lifecycle class allows other objects to query the current state.  internal class MyLocationListener(  private val context: Context,  private val lifecycle: Lifecycle,  private val callback: (Location) -> Unit  ): DefaultLifecycleObserver {  private var enabled = false  override fun onStart(owner: LifecycleOwner) {  if (enabled) {  // connect  }  }  fun enable() {  enabled = true  if (lifecycle.currentState.isAtLeast(Lifecycle.State.STARTED)) {  // connect if not connected  }  }  override fun onStop(owner: LifecycleOwner) {  // disconnect if connected  }  }  (With this implementation, our LocationListener class is completely lifecycle-aware.  If we need to use our LocationListener from another activity or fragment,  we just need to initialize it.  All of the setup and teardown operations are managed by the class itself.) | </end> |
| <hitle> | custom LifecycleOwner | <chare> | 2 | <pext> | Fragments and Activities in latest Support Library already implement  the LifecycleOwner interface.  If we have a custom class that need to make a LifecycleOwner, we can use  the LifecycleRegistry class, but we need to forward events into that class.  class MyActivity : Activity(), LifecycleOwner {  private lateinit var lifecycleRegistry: LifecycleRegistry  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  lifecycleRegistry = LifecycleRegistry(this)  lifecycleRegistry.markState(Lifecycle.State.CREATED)  }  public override fun onStart() {  super.onStart()  lifecycleRegistry.markState(Lifecycle.State.STARTED)  }  override fun getLifecycle(): Lifecycle {  return lifecycleRegistry  }  } | </end> |
| <hitle> | Best practices for lifecycle-aware components | <chare> | 2 | <pext> | 1. We need to keep UI controllers (activities and fragments)  as lean as possible.  They should not try to acquire their own data; instead,  use a ViewModel to do that, and observe a LiveData object  to reflect the changes back to the views.  2. We need to try to write data-driven UIs  where your UI controller’s responsibility is to update the views  as data changes, or notify user actions back to the ViewModel.  3. We need to put data logic in ViewModel class.  ViewModel should serve as the connector between UI controller  and the rest of the app. We need to be careful though,  it isn't ViewModel's responsibility to fetch data.  Instead, ViewModel should call the appropriate component  to fetch the data, then provide the result back to the UI controller.  4. We need to use Data Binding to maintain a clean interface  between views and the UI controller.  This allows to make views more declarative and minimize  the update code we need to write in activities and fragments.  If we need to do this in the Java programming language,  we need to use a library like Butter Knife to avoid boilerplate code  and have a better abstraction.  5. If UI is complex, we need to consider creating a presenter class  to handle UI modifications.  This might be a laborious task, but it can make UI components  easier to test.  6. We need to avoid referencing a View or Activity context  in your ViewModel.  If the ViewModel outlives the activity  (in case of configuration changes), the activity leaks  and isn't properly disposed by the garbage collector.  7. Use Kotlin coroutines to manage long-running tasks  and other operations that can run asynchronously. | </end> |
| <hitle> | Whentouse lifecycle-aware components | <chare> | 2 | <pext> | 1. Switching between coarse and fine-grained location updates.  We can use lifecycle-aware components to enable  fine-grained location updates while the location app is visible  and switch to coarse-grained updates  when the app is in the background.  LiveData, a lifecycle-aware component, allows the app  to automatically update the UI when your user changes locations.  2. Stopping and starting video buffering.  We can use lifecycle-aware components to start video buffering  as soon as possible, but defer playback until app is fully started.  We can also use lifecycle-aware components to terminate buffering  when your app is destroyed.  3. Starting and stopping network connectivity.  We can use lifecycle-aware components to enable live updating (streaming)  of network data while an app is in the foreground  and also to automatically pause when the app goes into the background.  4. Pausing and resuming animated drawables.  We can use lifecycle-aware components to handle pausing animated drawables  when while app is in the background and resume drawables  after the app is in the foreground. | </end> |
| <hitle> | Lifecycle stop events problem | <chare> | 2 | <pext> | When a Lifecycle belongs to an AppCompatActivity or Fragment,  the Lifecycle's state changes to CREATED and the ON\_STOP event is dispatched  when the AppCompatActivity or Fragment's onSaveInstanceState() is called.  When a Fragment or AppCompatActivity's state is saved via onSaveInstanceState(),  it's UI is considered immutable until ON\_START is called.  Trying to modify the UI after the state is saved is likely to cause inconsistencies  in the navigation state of the application which is why FragmentManager  throws an exception if the app runs a FragmentTransaction after state is saved.  LiveData prevents this edge case out of the box by refraining from calling its observer if the observer's associated Lifecycle isn't at least STARTED.  Behind the scenes, it calls isAtLeast() before deciding to invoke its observer.  Unfortunately, AppCompatActivity's onStop() method is called after  onSaveInstanceState(), which leaves a gap where UI state changes are not allowed  but the Lifecycle has not yet been moved to the CREATED state.  To prevent this issue, the Lifecycle class in version beta2 and lower mark the state  as CREATED without dispatching the event so that any code that checks  the current state gets the real value even though the event isn't dispatched  until onStop() is called by the system.  Unfortunately, this solution has two major problems:  1. On API level 23 and lower, the Android system actually saves the state  of an activity even if it is partially covered by another activity.  In other words, the Android system calls onSaveInstanceState()  but it doesn't necessarily call onStop().  This creates a potentially long interval where the observer still thinks that  the lifecycle is active even though its UI state can't be modified.  2. Any class that wants to expose a similar behavior to the LiveData class  has to implement the workaround provided by Lifecycle version beta 2 and lower. | </end> |
| <hitle> | ViewModel | <chare> | 2 | <pext> | ViewModel is one of the most critical classes  of the Android Jetpack Architecture Component that support data  for UI components.  Its purpose is to hold and manage the UI-related data.  Moreover, its main function is to maintain the integrity  and allows data to be serviced during configuration changes  like screen rotations.  Any kind of configuration change in Android devices tends to recreate  the whole activity of the application.  It means the data will be lost if it has not been saved and restored properly  from the activity which was destroyed.  To avoid these issues, it is recommended to store all UI data in the ViewModel  instead of an activity. | </end> |
| <hitle> | ViewModel pros | <chare> | 3 | <pext> | 1. It allows you to persist UI state.  ViewModel allows the survival through both the state that a ViewModel holds,  and operations that a ViewModel trigger.  This caching means that you don’t have to fetch data again  through common configuration changes, such as a screen rotation.  2. It provides access to business logic. | </end> |
| <hitle> | ViewModelProvider.Factory | <chare> | 3 | <pext> | Implementations of ViewModelProviders.Factory interface  are responsible to instantiate ViewModels.  That means write your own implementation  for creating an instance of ViewModel.  We pass our ViewModel arguments  to the ViewModelProvider.Factory through constructor  or any other pattern you prefer (Singleton, FactoryPattern etc.).  And it is because we cannot call ViewModel constructor  in Activity or Fragment when we initializing ViewModel  and you want to set ViewModel constructor’s argument value  so that’s why you need to pass argument value  to ViewModelProvider.Factory and it will create the ViewModel.  ViewModelProvider.Factory is an interface which have create method.  The create method is responsible for creating our VeiwModel’s instance.  So Why we need ViewModelProvider.Factory?  Here some questions comes on mind, I will not passing value to constructor of ViewModel, I will create method to set my values and it will work fine so why need this ViewModelProvider.Factory. In somehow you are right but some cases you need ViewModelProver.Factory.  When to use ViewModelProvider.Factory?  If your ViewModel have dependencies then you should pass this dependencies through the constructor (It is the best way to pass your dependencies), so you can mock that dependencies and test your ViewModel.  When not to use ViewModelProvider.Factory  If your ViewModel have no dependencies then you will not require to create your own ViewModelProvider.Factory. The default implementation is enough to create ViewModel for you. | </end> |
| <hitle> | ViewModel Persistence | <chare> | 3 | <pext> | ViewModel allows the survival through both the state that a ViewModel holds,  and operations that a ViewModel trigger.  This caching means that you don’t have to fetch data again  through common configuration changes, such as a screen rotation. | </end> |
| <hitle> | ViewModel lifecycle | <chare> | 3 | <pext> | The lifecycle of a ViewModel is tied directly to its scope.  A ViewModel remains in memory until the ViewModelStoreOwner  to which it is scoped disappears.  A ViewModel remains in memory:  1. In the case of an activity, when it finishes.  2. In the case of a fragment, when it detaches.  3. In the case of a Navigation entry, when it's removed from the back stack.  This makes ViewModels a great solution for storing data that survives  configuration changes.  We usually request a ViewModel the first time  the system calls an activity object's onCreate() method.  The system may call onCreate() several times throughout the existence  of an activity, such as when a device screen is rotated.  The ViewModel exists from when we first request a ViewModel  until the activity is finished and destroyed. | </end> |
| <hitle> | ViewModel business logic | <chare> | 3 | <pext> | Even though the vast majority of business logic is present in the data layer,  the UI layer can also contain business logic.  This can be the case when combining data from multiple repositories  to create the screen UI state, or when a particular type of data  doesn't require a data layer.  ViewModel is the right place to handle business logic in the UI layer.  The ViewModel is also in charge of handling events and delegating them  to other layers of the hierarchy when business logic needs to be applied  to modify application data. | </end> |
| <hitle> | ViewModel by Jetpack Compose | <chare> | 3 | <pext> | When using Jetpack Compose, ViewModel is the primary means of  exposing screen UI state to the composables.  In a hybrid app, activities and fragments simply host the composable functions.  This is a shift from past approaches, where it wasn't that simple and intuitive  to create reusable pieces of UI with activities and fragments,  which caused them to be much more active as UI controllers.  The most important thing to keep in mind when using ViewModel with Compose  is that we cannot scope a ViewModel to a composable.  This is because a composable is not a ViewModelStoreOwner.  Two instances of the same composable in the Composition,  or two different composables accessing the same ViewModel type  under the same ViewModelStoreOwner would receive  the same instance of the ViewModel, which often is not the expected behavior.  To get the benefits of ViewModel in Compose, we need to host each screen  in a Fragment or Activity, or use Compose Navigation and use ViewModels  in composable functions as close as possible to the Navigation destination.  That is because we can scope a ViewModel to Navigation destinations,  Navigation graphs, Activities, and Fragments. | </end> |
| <hitle> | howto ViewModel | <chare> | 3 | <pext> | data class DiceUiState(  val firstDieValue: Int? = null,  val secondDieValue: Int? = null,  val numberOfRolls: Int = 0,  )  class DiceRollViewModel : ViewModel() {  // Expose screen UI state  private val \_uiState = MutableStateFlow(DiceUiState())  val uiState: StateFlow<DiceUiState> = \_uiState.asStateFlow()  // Handle business logic  fun rollDice() {  \_uiState.update { currentState ->  currentState.copy(  firstDieValue = Random.nextInt(from = 1, until = 7),  secondDieValue = Random.nextInt(from = 1, until = 7),  numberOfRolls = currentState.numberOfRolls + 1,  )  }  }  }  (It is displaying a list of users.)  import androidx.activity.viewModels  class DiceRollActivity : AppCompatActivity() {  override fun onCreate(savedInstanceState: Bundle?) {  // Create a ViewModel the first time the system calls an activity's onCreate() method.  // Re-created activities receive the same DiceRollViewModel instance created by the first activity.  // Use the 'by viewModels()' Kotlin property delegate  // from the activity-ktx artifact  val viewModel: DiceRollViewModel by viewModels()  lifecycleScope.launch {  repeatOnLifecycle(Lifecycle.State.STARTED) {  viewModel.uiState.collect {  // Update UI elements  }  }  }  }  }  (We can then access the list from an activity.) | </end> |
| <hitle> | View Model Best practices | <chare> | 3 | <pext> | 1. Because of their scoping, we may use ViewModels as implementation details  of a screen level state holder.  We should not use them as state holders of reusable UI components  such as chip groups or forms.  Otherwise, we would get the same ViewModel instance in different usages  of the same UI component under the same ViewModelStoreOwner.  2. ViewModels shouldn't know about the UI implementation details.  We need to keep the names of the methods the ViewModel API exposes  and those of the UI state fields as generic as possible.  In this way, the ViewModel can accommodate any type of UI: a mobile phone,  foldable, tablet, or even a Chromebook!  3. As Viewmodel can potentially live longer than the ViewModelStoreOwner,  ViewModels shouldn't hold any references of lifecycle-related APIs  such as the Context or Resources to prevent memory leaks.  4. We should not pass ViewModels to other classes, functions  or other UI components.  Because the platform manages them, we should keep them as close to it  as we can.  We need to close to the Activity, fragment, or screen level composable function.  This prevents lower level components from accessing more data and logic  than they need. | </end> |
| <hitle> | ViewModel dependency injection | <chare> | 3 | <pext> | ViewModels can take dependencies as parameters  in their constructor.  These are mostly of types from the domain or data layers.  Because the framework provides the ViewModels,  a special mechanism is required to create instances of them.  That mechanism is the ViewModelProvider.Factory interface.  Only implementations of this interface can instantiate ViewModels  in the right scope.  If the ViewModel takes no dependencies or just the SavedStateHandle type  as a dependency, we don't need to provide a factory for the framework  to instantiate instances of that ViewModel type.  When injecting ViewModels using Hilt as a dependency injection solution,  we don't have to define a ViewModel factory manually.  Hilt generates a factory that knows how to create all ViewModels annotated  with @HiltViewModel for you at compile time.  Classes annotated with @AndroidEntryPoint can directly access  the Hilt generated factory when calling the regular ViewModel APIs. | </end> |
| <hitle> | ViewModels with CreationExtras | <chare> | 4 | <pext> | If a ViewModel class receives dependencies in its constructor,  we need to provide a factory that implements  the ViewModelProvider.Factory interface.  And we need to override the create(Class<T>, CreationExtras) function  to provide a new instance of the ViewModel.  CreationExtras allows to access relevant information  that helps instantiate a ViewModel.  There's several keys that can be accessed from extras:  1. ViewModelProvider.NewInstanceFactory.VIEW\_MODEL\_KEY  It provides access to the custom key you passed to ViewModelProvider.get().  2. ViewModelProvider.AndroidViewModelFactory.APPLICATION\_KEY  It provides access to the instance of the Application class.  3. SavedStateHandleSupport.DEFAULT\_ARGS\_KEY  It provides access to the Bundle of arguments you should use to construct SavedStateHandle.  4. SavedStateHandleSupport.SAVED\_STATE\_REGISTRY\_OWNER\_KEY  It provides access to the SavedStateRegistryOwner that is being used to construct the ViewModel.  5. SavedStateHandleSupport.VIEW\_MODEL\_STORE\_OWNER\_KEY  It provides access to the ViewModelStoreOwner that is being used to construct the ViewModel.  To create a new instance of SavedStateHandle, we can use  the CreationExtras.createSavedStateHandle().createSavedStateHandle()) function  and pass it to the ViewModel.  import androidx.lifecycle.SavedStateHandle  import androidx.lifecycle.ViewModel  import androidx.lifecycle.ViewModelProvider  import androidx.lifecycle.ViewModelProvider.AndroidViewModelFactory.Companion.APPLICATION\_KEY  import androidx.lifecycle.createSavedStateHandle  import androidx.lifecycle.viewmodel.CreationExtras  class MyViewModel(  private val myRepository: MyRepository,  private val savedStateHandle: SavedStateHandle  ) : ViewModel() {  // ViewModel logic  // ...  // Define ViewModel factory in a companion object  companion object {  val Factory: ViewModelProvider.Factory = object : ViewModelProvider.Factory {  @Suppress("UNCHECKED\_CAST")  override fun <T : ViewModel> create(  modelClass: Class<T>,  extras: CreationExtras  ): T {  // Get the Application object from extras  val application = checkNotNull(extras[APPLICATION\_KEY])  // Create a SavedStateHandle for this ViewModel from extras  val savedStateHandle = extras.createSavedStateHandle()  return MyViewModel(  (application as MyApplication).myRepository,  savedStateHandle  ) as T  }  }  }  }  (It shos how to provide an instance of a ViewModel that takes a repository scoped to the Application class and SavedStateHandle as dependencies.)  import androidx.activity.viewModels  class MyActivity : AppCompatActivity() {  private val viewModel: MyViewModel by viewModels { MyViewModel.Factory }  // Rest of Activity code  }  (Then, we can use this factory when retrieving an instance of the ViewModel.)  import androidx.lifecycle.SavedStateHandle  import androidx.lifecycle.ViewModel  import androidx.lifecycle.ViewModelProvider  import androidx.lifecycle.ViewModelProvider.AndroidViewModelFactory.Companion.APPLICATION\_KEY  import androidx.lifecycle.createSavedStateHandle  import androidx.lifecycle.viewmodel.initializer  import androidx.lifecycle.viewmodel.viewModelFactory  class MyViewModel(  private val myRepository: MyRepository,  private val savedStateHandle: SavedStateHandle  ) : ViewModel() {  // ViewModel logic  // Define ViewModel factory in a companion object  companion object {  val Factory: ViewModelProvider.Factory = viewModelFactory {  initializer {  val savedStateHandle = createSavedStateHandle()  val myRepository = (this[APPLICATION\_KEY] as MyApplication).myRepository  MyViewModel(  myRepository = myRepository,  savedStateHandle = savedStateHandle  )  }  }  }  }  (Alternatively, we can use the ViewModel factory DSL to create factories using a more idiomatic Kotlin API.) | </end> |
| <hitle> | ViewModel Scope | <chare> | 3 | <pext> | When we instantiate a ViewModel, we pass it an object that implements  the ViewModelStoreOwner interface.  This may be a Navigation destination, Navigation graph, activity,  fragment, or any other type that implements the interface.  The ViewModel is then scoped to the Lifecycle  of the ViewModelStoreOwner.  It remains in memory until its ViewModelStoreOwner  goes away permanently.  A range of classes are either direct or indirect subclasses of  the ViewModelStoreOwner interface.  The direct subclasses are ComponentActivity, Fragment,  and NavBackStackEntry.  When the fragment or activity to which the ViewModel is scoped  is destroyed, asynchronous work continues in the ViewModel  that is scoped to it.  This is the key to persistence.  Scope is key to using ViewModels effectively.  Each ViewModel is scoped to an object that implements  the ViewModelStoreOwner interface.  There are several APIs that allow to more easily manage  the scope of ViewModels.  The ViewModelProvider.get() method let us obtain  an instance of a ViewModel scoped to any ViewModelStoreOwner. | </end> |
| <hitle> | ViewModels scoped to the closest ViewModelStoreOwner | <chare> | 4 | <pext> | We can scope a ViewModel to an Activity, Fragment,  or destination of a Navigation graph.  The viewModels() extension functions provided by the Activity,  Fragment and Navigation libraries,  and the viewModel() function in Compose allows to get an instance  of the ViewModel scoped to the closest ViewModelStoreOwner.  import androidx.activity.viewModels  class MyActivity : AppCompatActivity() {  // ViewModel API available in activity.activity-ktx  // The ViewModel is scoped to `this` Activity  val viewModel: MyViewModel by viewModels()  }  import androidx.fragment.app.viewModels  class MyFragment : Fragment() {  // ViewModel API available in fragment.fragment-ktx  // The ViewModel is scoped to `this` Fragment  val viewModel: MyViewModel by viewModels()  } | </end> |
| <hitle> | ViewModels scoped to any ViewModelStoreOwner | <chare> | 4 | <pext> | The ComponentActivity.viewModels()  and Fragment.viewModels() functions in the View system  and the viewModel() function in Compose  take an optional ownerProducer parameter that we can use to specify  to which ViewModelStoreOwner the instance of the ViewModel  is scoped to.  import androidx.fragment.app.viewModels  class MyFragment : Fragment() {  // ViewModel API available in fragment.fragment-ktx  // The ViewModel is scoped to the parent of `this` Fragment  val viewModel: SharedViewModel by viewModels(  ownerProducer = { requireParentFragment() }  )  }  (It shows how to get an instance of a ViewModel scoped to the parent fragment.)  When getting an Activity-scoped ViewModel from a Fragment,  we can use the activityViewModels() Views extension function.  If we're not using Views and Kotlin, we can use the same APIs  as above and by passing the right owner.  import androidx.fragment.app.activityViewModels  class MyFragment : Fragment() {  // ViewModel API available in fragment.fragment-ktx  // The ViewModel is scoped to the host Activity  val viewModel: SharedViewModel by activityViewModels()  } | </end> |
| <hitle> | ViewModels scoped to the Navigation graph | <chare> | 4 | <pext> | Navigation graphs are also ViewModel store owners.  If we're using Navigation Fragment or Navigation Compose,  we can get an instance of a ViewModel scoped to a Navigation graph  with the navGraphViewModels(graphId) Views extension function.  import androidx.navigation.navGraphViewModels  class MyFragment : Fragment() {  // ViewModel API available in navigation.navigation-fragment  // The ViewModel is scoped to the `nav\_graph` Navigation graph  val viewModel: SharedViewModel by navGraphViewModels(R.id.nav\_graph)  // Equivalent navGraphViewModels code using the viewModels API  val viewModel: SharedViewModel by viewModels(  { findNavController().getBackStackEntry(R.id.nav\_graph) }  )  }  If we're using Hilt in addition to Jetpack Navigation,  we can use the hiltNavGraphViewModels(graphId) API.  import androidx.hilt.navigation.fragment.hiltNavGraphViewModels  class MyFragment : Fragment() {  // ViewModel API available in hilt.hilt-navigation-fragment  // The ViewModel is scoped to the `nav\_graph` Navigation graph  // and is provided using the Hilt-generated ViewModel factory  val viewModel: SharedViewModel by hiltNavGraphViewModels(R.id.nav\_graph)  } | </end> |
| <hitle> | Saved State module for ViewModel | <chare> | 3 | <pext> | ViewModel objects can handle configuration changes,  so we don't need to worry about state in rotations or other cases.  However, if we need to handle system-initiated process death,  we might want to use the SavedStateHandle API as backup.  UI state is usually stored or referenced in ViewModel objects  and not activities, so using onSaveInstanceState()  or rememberSaveable requires some boilerplate  that the saved state module can handle.  When using this module, ViewModel objects receive  a SavedStateHandle object through its constructor.  This object is a key-value map that we can write and retrieve objects  to and from the saved state.  These values persist after the process is killed by the system  and remain available through the same object.  Saved state is tied to the task stack. If the task stack goes away,  saved state also goes away.  This can occur when force stopping an app,  removing the app from the recents menu,  or rebooting the device. In such cases, the task stack disappears  and we can't restore the information in saved state.  In User-initiated UI state dismissal scenarios,  saved state isn't restored. In system-initiated scenarios, it is. | </end> |
| <hitle> | Setup SavedStateHandle | <chare> | 4 | <pext> | We can accept a SavedStateHandle as a constructor argument  to the ViewModel.  class SavedStateViewModel(private val state: SavedStateHandle) : ViewModel() { ... }  We can then retrieve an instance of the ViewModel  without any additional configuration.  The default ViewModel factory provides  the appropriate SavedStateHandle to the ViewModel.  class MainFragment : Fragment() {  val vm: SavedStateViewModel by viewModels()  ...  }  When providing a custom ViewModelProvider.Factory instance,  we can enable usage of SavedStateHandle by extending  AbstractSavedStateViewModelFactory. | </end> |
| <hitle> | Howto SavedStateHandle | <chare> | 4 | <pext> | The SavedStateHandle class is a key-value map that allows  to write and retrieve data to and from the saved state  through the set() and get() methods.  By using SavedStateHandle, the query value is retained  across process death, ensuring that the user  sees the same set of filtered data before and after recreation  without the activity or fragment needing to manually save, restore,  and forward that value back to the ViewModel.  Caution: SavedStateHandle only saves data written to it  when the Activity is stopped.  Writes to SavedStateHandle while the Activity is stopped  aren't saved unless the Activity receives onStart  followed by onStop again.  SavedStateHandle also has other methods we may expect  when interacting with a key-value map:  1. contains(String key) - Checks if there is a value for the given key.  2. remove(String key) - Removes the value for the given key.  3. keys() - Returns all keys contained within the SavedStateHandle.  Additionally, we can retrieve values from SavedStateHandle  using an observable data holder.  Supported types are:  1. LiveData.  2. StateFlow.  3. Compose's State APIs. | </end> |
| <hitle> | SavedStateHandle by LiveData | <chare> | 5 | <pext> | We can retrieve values from SavedStateHandle that are wrapped  in a LiveData observable using getLiveData().  When the key's value is updated, the LiveData receives the new value.  Most often, the value is set due to user interactions,  such as entering a query to filter a list of data.  This updated value can then be used to transform LiveData.  class SavedStateViewModel(private val savedStateHandle: SavedStateHandle) : ViewModel() {  val filteredData: LiveData<List<String>> =  savedStateHandle.getLiveData<String>("query").switchMap { query ->  repository.getFilteredData(query)  }  fun setQuery(query: String) {  savedStateHandle["query"] = query  }  } | </end> |
| <hitle> | SavedStateHandle by StateFlow | <chare> | 5 | <pext> | We can retrieve values from SavedStateHandle that are wrapped  in a StateFlow observable using getStateFlow().  When we update the key's value, the StateFlow receives the new value.  Most often, the value is set due to user interactions,  such as entering a query to filter a list of data.  We can then transform this updated value using other Flow operators.  class SavedStateViewModel(private val savedStateHandle: SavedStateHandle) : ViewModel() {  val filteredData: StateFlow<List<String>> =  savedStateHandle.getStateFlow<String>("query")  .flatMapLatest { query ->  repository.getFilteredData(query)  }  fun setQuery(query: String) {  savedStateHandle["query"] = query  }  } | </end> |
| <hitle> | SavedStateHandle supported types | <chare> | 4 | <pext> | Data kept within a SavedStateHandle is saved and restored as a Bundle,  along with the rest of the savedInstanceState for the activity or fragment.  By default, we can call set() and get() on a SavedStateHandle  for the same data types as a Bundle.  If the class does not extend the default data types, we need to consider  making the class parcelable by adding the @Parcelize Kotlin annotation  or implementing Parcelable directly.  If a class does not implement Parcelable or Serializable  and cannot be modified to implement one of those interfaces,  then it is not possible to directly save an instance of that class  into a SavedStateHandle.  In recent Lifecycle library, SavedStateHandle allows to save any object  by providing it’s own logic for saving and restoring the object  as a Bundle using the setSavedStateProvider() method.  SavedStateRegistry.SavedStateProvider is an interface that defines  a single saveState() method that returns a Bundle containing  the state we want to save.  When SavedStateHandle is ready to save its state, it calls saveState()  to retrieve the Bundle from the SavedStateProvider  and saves the Bundle for the associated key.  class TempFileViewModel : ViewModel() {  private var tempFile: File? = null  fun createOrGetTempFile(): File {  return tempFile ?: File.createTempFile("temp", null).also {  tempFile = it  }  }  }  (Let’s see an example of an app that requests an image from the camera app  via the ACTION\_IMAGE\_CAPTURE intent,  passing in a temporary file for where the camera should store the image.  The TempFileViewModel encapsulates the logic for creating that temporary file.)  private fun File.saveTempFile() = bundleOf("path", absolutePath)  class TempFileViewModel(savedStateHandle: SavedStateHandle) : ViewModel() {  private var tempFile: File? = null  init {  savedStateHandle.setSavedStateProvider("temp\_file") { // saveState()  if (tempFile != null) {  tempFile.saveTempFile()  } else {  Bundle()  }  }  }  fun createOrGetTempFile(): File {  return tempFile ?: File.createTempFile("temp", null).also {  tempFile = it  }  }  }  (To ensure the temporary file is not lost if the activity's process is killed and later restored, TempFileViewModel can use the SavedStateHandle to persist its data.  To allow TempFileViewModel to save its data, implement SavedStateProvider and set it  as a provider on the SavedStateHandle of the ViewModel.)  private fun File.saveTempFile() = bundleOf("path", absolutePath)  private fun Bundle.restoreTempFile() = if (containsKey("path")) {  File(getString("path"))  } else {  null  }  class TempFileViewModel(savedStateHandle: SavedStateHandle) : ViewModel() {  private var tempFile: File? = null  init {  val tempFileBundle = savedStateHandle.get<Bundle>("temp\_file")  if (tempFileBundle != null) {  tempFile = tempFileBundle.restoreTempFile()  }  savedStateHandle.setSavedStateProvider("temp\_file") { // saveState()  if (tempFile != null) {  tempFile.saveTempFile()  } else {  Bundle()  }  }  }  fun createOrGetTempFile(): File {  return tempFile ?: File.createTempFile("temp", null).also {  tempFile = it  }  }  }  (To restore the File data when the user returns, retrieve the temp\_file Bundle from  the SavedStateHandle. This is the same Bundle provided by saveTempFile() that contains  the absolute path. The absolute path can then be used to instantiate a new File.) | </end> |
| <hitle> | SavedStateHandle in tests | <chare> | 4 | <pext> | To test a ViewModel that takes a SavedStateHandle as a dependency,  we can create a new instance of SavedStateHandle  with the test values it requires and pass it to the ViewModel instance  that we are testing.  class MyViewModelTest {  private lateinit var viewModel: MyViewModel  @Before  fun setup() {  val savedState = SavedStateHandle(mapOf("someIdArg" to testId))  viewModel = MyViewModel(savedState = savedState)  }  } | </end> |
| <hitle> | LiveData | <chare> | 2 | <pext> | LiveData component is an observable data holder class  like, the contained value can be observed.  LiveData is a lifecycle-aware component and thus it performs its functions  according to the lifecycle state of other application components.  Further, if the observer’s lifecycle state is active  like, either STARTED or RESUMED,  only then LiveData updates the app component.  LiveData always checks the observer’s state  before making any update to ensure  that the observer must be active to receive it.  If the observer’s lifecycle state is destroyed, LiveData is capable of removing it,  and thus it avoids memory leaks.  It makes the task of data synchronization easier. | </end> |
| <hitle> | LiveData pros | <chare> | 3 | <pext> | 1. It ensures the UI matches data state.  LiveData follows the observer pattern.  LiveData notifies Observer objects when underlying data changes.  We can consolidate the code to update the UI in these Observer objects.  That way, we don't need to update the UI every time the app data changes  because the observer does it for you.  2. Using LiveData, there’s no memory leaks.  Observers are bound to Lifecycle objects and clean up after themselves  when their associated lifecycle is destroyed.  3. Using LiveData, there’s no crashes due to stopped activities.  If the observer's lifecycle is inactive,  such as in the case of an activity in the back stack,  then it doesn’t receive any LiveData events.  4. Using LiveData, there’s no more manual lifecycle handling.  UI components just observe relevant data and don’t stop  or resume observation.  LiveData automatically manages all of this  since it’s aware of the relevant lifecycle status changes while observing.  5. LiveData let us keep always up to date data.  If a lifecycle becomes inactive, it receives the latest data  upon becoming active again.  For example, an activity that was in the background  receives the latest data right after it returns to the foreground.  6. Using LiveData, we can make proper configuration changes.  If an activity or fragment is recreated due to a configuration change,  like device rotation, it immediately receives the latest available data.  7. Sharing resources is powerful when using LiveData  We can extend a LiveData object using the singleton pattern  to wrap system services so that they can be shared in your app.  The LiveData object connects to the system service once,  and then any observer that needs the resource  can just watch the LiveData object. | </end> |
| <hitle> | Howto LiveData | <chare> | 3 | <pext> | 1. Create an instance of LiveData to hold a certain type of data.  This is usually done within the ViewModel class.  2. Create an Observer object that defines the onChanged() method,  which controls what happens  when the LiveData object's held data changes.  We usually create an Observer object in a UI controller,  such as an activity or fragment.  3. Attach the Observer object to the LiveData object  using the observe() method.  The observe() method takes a LifecycleOwner object.  This subscribes the Observer object to the LiveData object  so that it is notified of changes.  We usually attach the Observer object in a UI controller,  such as an activity or fragment.  Note: We can register an observer  without an associated LifecycleOwner object  using the observeForever(Observer) method.  In this case, the observer is considered to be always active  and is therefore always notified about modifications.  We can remove these observers calling the removeObserver(Observer) method.  When we update the value stored in the LiveData object,  it triggers all registered observers as long as the attached LifecycleOwner  is in the active state.  LiveData allows UI controller observers to subscribe to updates.  When the data held by the LiveData object changes,  the UI automatically updates in response. | </end> |
| <hitle> | Create LiveData | <chare> | 3 | <pext> | LiveData is a wrapper that can be used with any data,  including objects that implement Collections, such as List.  A LiveData object is usually stored within a ViewModel object  and is accessed via a getter method.  class NameViewModel : ViewModel() {  // Create a LiveData with a String  val currentName: MutableLiveData<String> by lazy {  MutableLiveData<String>()  }  // Rest of the ViewModel...  }  Initially, the data in a LiveData object is not set.  When creating LiveData, we need to store LiveData objects that update  the UI in ViewModel objects, as opposed to an activity or fragmentnt  to avoid bloated activities and fragments,  and to decouple LiveData instances from specific activity  or fragment instances and allow LiveData objects  to survive configuration changes. | </end> |
| <hitle> | Observe LiveData | <chare> | 3 | <pext> | In most cases, an app component’s onCreate() method is the right place  to begin observing a LiveData object.  1. The reason is, we need to ensure the system doesn’t make redundant calls  from an activity or fragment’s onResume() method.  2. The second reason is, we need to ensure that the activity or fragment  has data that it can display as soon as it becomes active.  As soon as an app component is in the STARTED state,  it receives the most recent value from the LiveData objects it’s observing.  This only occurs if the LiveData object to be observed has been set.  Generally, LiveData delivers updates only when data changes,  and only to active observers.  An exception to this behavior is that observers also receive an update  when they change from an inactive to an active state.  Furthermore, if the observer changes from inactive to active a second time,  it only receives an update if the value has changed  since the last time it became active.  class NameActivity : AppCompatActivity() {  // Use the 'by viewModels()' Kotlin property delegate  // from the activity-ktx artifact  private val model: NameViewModel by viewModels()  override fun onCreate(savedInstanceState: Bundle?) {  super.onCreate(savedInstanceState)  // Other code to setup the activity...  // Create the observer which updates the UI.  val nameObserver = Observer<String> { newName ->  // Update the UI, in this case, a TextView.  nameTextView.text = newName  }  // Observe the LiveData, passing in this activity as the LifecycleOwner and the observer.  model.currentName.observe(this, nameObserver)  }  }  (It shows how to start observing a LiveData object.  After observe() is called with nameObserver passed as parameter, onChanged() is immediately invoked providing the most recent value stored in mCurrentName. If the LiveData object hasn't set a value in mCurrentName, onChanged() is not called.) | </end> |
| <hitle> | Update LiveData | <chare> | 3 | <pext> | LiveData has no publicly available methods to update the stored data.  The MutableLiveData class exposes  the setValue(T) and postValue(T) methods publicly  and we must use these if we need to edit the value stored in a LiveData object.  Usually MutableLiveData is used in the ViewModel  and then the ViewModel only exposes immutable LiveData objects  to the observers.  button.setOnClickListener {  val anotherName = "John Doe"  model.currentName.setValue(anotherName)  }  (After set up the observer relationship,  we can then update the value of the LiveData object,  which triggers all observers when the user taps a button.  Calling setValue(T) results in the observers calling their onChanged() methods  with the value John Doe. The example shows a button press, but setValue() or postValue() could be called to update mName for a variety of reasons, including in response to a network request or a database load completing; in all cases, the call to setValue() or postValue() triggers observers and updates the UI.) | </end> |
| <hitle> | setValue() vs postValue() LiveData | <chare> | 3 | <pext> | 1. setValue() sets the value.  If there are active observers, the value will be dispatched to them.  This method must be called from the main thread.  2. postValue() posts a task to a main thread to set the given value.  If we called this method multiple times  before a main thread executed a posted task,  only the last value would be dispatched.  So the key difference is, setValue() method must be called  from the main thread.  But if we need set a value from a background thread,  postValue() should be used. | </end> |
| <hitle> | LiveData with Room | <chare> | 3 | <pext> | The Room persistence library supports observable queries,  which return LiveData objects.  Observable queries are written as part of a Database Access Object (DAO).  Room generates all the necessary code to update the LiveData object  when a database is updated.  The generated code runs the query asynchronously  on a background thread when needed.  This pattern is useful for keeping the data displayed in a UI in sync  with the data stored in a database. | </end> |
| <hitle> | LiveData in app architecture | <chare> | 3 | <pext> | LiveData is lifecycle-aware, following the lifecycle of entities  such as activities and fragments.  We can use LiveData to communicate between these lifecycle owners  and other objects with a different lifespan, such as ViewModel objects.  The main responsibility of the ViewModel is to load and manage  UI-related data, which makes it a great candidate  for holding LiveData objects.  We can create LiveData objects in the ViewModel and use them  to expose state to the UI layer.  Activities and fragments should not hold LiveData instances  because their role is to display data, not hold state.  Also, keeping activities and fragments free from holding data  makes it easier to write unit tests.  It may be tempting to work LiveData objects in your data layer class,  but LiveData is not designed to handle asynchronous streams of data.  Even though we can use LiveData transformations and MediatorLiveData  to achieve this, this approach has drawbacks  which the capability to combine streams of data is very limited  and all LiveData objects are observed on the main thread.  class UserRepository {  // DON'T DO THIS! LiveData objects should not live in the repository.  fun getUsers(): LiveData<List<User>> {  ...  }  fun getNewPremiumUsers(): LiveData<List<User>> {  return TransformationsLiveData.map(getUsers()) { users ->  // This is an expensive call being made on the main thread and may  // cause noticeable jank in the UI!  users  .filter { user ->  user.isPremium  }  .filter { user ->  val lastSyncedTime = dao.getLastSyncedTime()  user.timeCreated > lastSyncedTime  }  }  }  (\*It shos how holding a LiveData in the Repository  can block the main thread.\*)  If we need to use streams of data in other layers of your app,  we need to consider using Kotlin Flows and then converting them  to LiveData in the ViewModel using asLiveData(). | </end> |
| <hitle> | Extend LiveData | <chare> | 3 | <pext> | LiveData considers an observer to be in an active state  if the observer's lifecycle is in either the STARTED or RESUMED states.  class StockLiveData(symbol: String) : LiveData<BigDecimal>() {  private val stockManager = StockManager(symbol)  private val listener = { price: BigDecimal ->  value = price  }  override fun onActive() {  stockManager.requestPriceUpdates(listener)  }  override fun onInactive() {  stockManager.removeUpdates(listener)  }  }  (\*It shos how to extend the LiveData class.  The implementation of the price listener includes  the following important methods:  1. The onActive() method is called when the LiveData object  has an active observer. This means we need to start observing  the stock price updates from this method.  2. The onInactive() method is called when the LiveData object  doesn't have any active observers.  Since no observers are listening, there is no reason to stay  connected to the StockManager service.  3. The setValue(T) method updates the value of the LiveData instance  and notifies any active observers about the change.\*)  public class MyFragment : Fragment() {  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  super.onViewCreated(view, savedInstanceState)  val myPriceListener: LiveData<BigDecimal> = ...  myPriceListener.observe(viewLifecycleOwner, Observer<BigDecimal> { price: BigDecimal? ->  // Update the UI.  })  }  }  (\* We can use the StockLiveData class.  The observe() method passes the LifecycleOwner  associated with the fragment's view as the first argument.  Doing so it denotes that this observer is bound to the Lifecycle object  associated with the owner, meaning:  If the Lifecycle object is not in an active state, then the observer  isn't called even if the value changes.  After the Lifecycle object is destroyed,  the observer is automatically removed.  The fact that LiveData objects are lifecycle-aware means  that we can share them between multiple activities,  fragments, and services. \*)  class StockLiveData(symbol: String) : LiveData<BigDecimal>() {  private val stockManager: StockManager = StockManager(symbol)  private val listener = { price: BigDecimal ->  value = price  }  override fun onActive() {  stockManager.requestPriceUpdates(listener)  }  override fun onInactive() {  stockManager.removeUpdates(listener)  }  companion object {  private lateinit var sInstance: StockLiveData  @MainThread  fun get(symbol: String): StockLiveData {  sInstance = if (::sInstance.isInitialized) sInstance else StockLiveData(symbol)  return sInstance  }  }  }  (\*To make simple, we can implement the LiveData class as a singleton.\*)  class MyFragment : Fragment() {  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  super.onViewCreated(view, savedInstanceState)  StockLiveData.get(symbol).observe(viewLifecycleOwner, Observer<BigDecimal> { price: BigDecimal? ->  // Update the UI.  })  }  (\*And we can use it in the fragment.  Multiple fragments and activities can observe  the MyPriceListener instance.  LiveData only connects to the system service  if one or more of them is visible and active.\*) | </end> |
| <hitle> | Transform LiveData | <chare> | 3 | <pext> | We may want to make changes to the value stored in a LiveData object  before dispatching it to the observers, or you may need to return  a different LiveData instance based on the value of another one.  The Lifecycle package provides the Transformations class  which includes helper methods that support these scenarios.  1. Transformations.map()  It applies a function on the value stored in the LiveData object,  and propagates the result downstream.  val userLiveData: LiveData<User> = UserLiveData()  val userName: LiveData<String> = Transformations.map(userLiveData) {  user -> "${user.name} ${user.lastName}"  }  2. Transformations.switchMap()  It is similar to map(), it applies a function to the value stored  in the LiveData object and unwraps and dispatches the result downstream.  The function passed to switchMap() must return a LiveData object.  private fun getUser(id: String): LiveData<User> {  ...  }  val userId: LiveData<String> = ...  val user = Transformations.switchMap(userId) { id -> getUser(id) }  We can use transformation methods to carry information  across the observer's lifecycle.  The transformations aren't calculated unless an observer is watching  the returned LiveData object.  Because the transformations are calculated lazily,  lifecycle-related behavior is implicitly passed down  without requiring additional explicit calls or dependencies.  If we need a Lifecycle object inside a ViewModel object,  a transformation is probably a better solution.  For example, say we have a UI component that accepts an address  and returns the postal code for that address.  We can implement the naive ViewModel for this component.  class MyViewModel(private val repository: PostalCodeRepository) : ViewModel() {  private fun getPostalCode(address: String): LiveData<String> {  // DON'T DO THIS  return repository.getPostCode(address)  }  }  (\*The UI component then needs to unregister  from the previous LiveData object and register to the new instance  each time it calls getPostalCode().  In addition, if the UI component is recreated, it triggers another call  to the repository.getPostCode() method  instead of using the previous call’s result.\*)  class MyViewModel(private val repository: PostalCodeRepository) : ViewModel() {  private val addressInput = MutableLiveData<String>()  val postalCode: LiveData<String> = Transformations.switchMap(addressInput) {  address -> repository.getPostCode(address) }  private fun setInput(address: String) {  addressInput.value = address  }  }  (\*Instead, we can implement the postal code lookup  as a transformation of the address input.  In this case, the postalCode field is defined  as a transformation of the addressInput.  As long as the app has an active observer associated  with the postalCode field, the field's value is recalculated  and retrieved whenever addressInput changes.\*)  This mechanism allows lower levels of the app to create LiveData objects  that are lazily calculated on demand.  A ViewModel object can easily obtain references to LiveData objects  and then define transformation rules on top of them. | </end> |
| <hitle> | Create new transformations | <chare> | 3 | <pext> | There are a dozen different specific transformation  that may be useful in your app, but they aren’t provided by default.  To implement own transformation we can use the MediatorLiveData class,  which listens to other LiveData objects and processes events  emitted by them.  MediatorLiveData correctly propagates its state  to the source LiveData object. | </end> |
| <hitle> | Merge multiple LiveData | <chare> | 3 | <pext> | MediatorLiveData is a subclass of LiveData  that allows you to merge multiple LiveData sources.  Observers of MediatorLiveData objects are then triggered  whenever any of the original LiveData source objects change.  For example, if we have a LiveData object in UI that can be updated  from a local database or a network, then you can add  LiveData sources to the MediatorLiveData object.  Those LiveData sources are:  1. A LiveData object associated with the data stored in the database.  2. A LiveData object associated with the data accessed from the network.  Then, The activity only needs to observe the MediatorLiveData object  to receive updates from both sources. | </end> |
| <hitle> | Save UI states | <chare> | 2 | <pext> | Preserving and restoring an activity's UI state in a timely fashion  across system-initiated activity or application destruction  is a crucial part of the user experience.  In these cases the user expects the UI state to remain the same,  but the system destroys the activity and any state stored in it.  To bridge the gap between user expectation and system behavior,  we can use a combination of ViewModel objects, saved instance state  that includes the onSaveInstanceState() API in the View system,  rememberSaveable in Jetpack Compose  and SavedStateHandle in ViewModels, and local storage  to persist the UI state across such application  and Activity instance transitions.  Deciding how to combine these options depends on the complexity  of UI data, use cases for the app, and consideration of  how fast the data is available versus memory usage.  Regardless of which approach we take, we should ensure  the app meets users expectations with respect to their UI state,  and provides a smooth, snappy UI  like avoids lag time in loading data into the UI,  especially after frequently occurring configuration changes, like rotation.  In most cases we should use a combination  of the different approaches to get the best user experience in your app. | </end> |
| <hitle> | Options for save UI state | <chare> | 3 | <pext> | 1. Storage location  1-1 for ViewModel, it is saved in memory.  1-2 for Saved instance state, it is saved in memory.  1-3 for Persistent storage, it is saved on disk or network.  2. Survives configuration change  We can use ViewModel, Saved instance sate, Persistent storage.  3. Survives system-initiated process death  ViewModel can’t be used.  But Saved instance sate, Persistent storage can be used.  4. Survives user complete activity dismissal/onFinish()  ViewModel and Saved instance state can’t be used.  Persistent storate can be used.  5. Data limitations  Using ViewModel, complex objects are fine,  but space is limited by available memory.  Using Saved instance state, only for primitive types and simple,  small objects such as String can be saved.  Using Persistent storage, the limitation is disk space or cost,  and time of retrieval from the network resource.  6. Read/write time  ViewModel is quick because memory access only  Saved instance state is slow  because it requires serialization/deserialization and disk access.  Persistent storage is slow  because it requires disk access or network transaction. | </end> |
| <hitle> | MVVM ViewModel to handle configuration changes | <chare> | 3 | <pext> | ViewModel is ideal for storing and managing UI-related data  while the user is actively using the application.  It allows quick access to UI data and helps us  avoid refetching data from network or disk across rotation,  window resizing, and other commonly occurring configuration changes.  ViewModel retains the data in memory, which means it is cheaper  to retrieve than data from the disk or the network.  A ViewModel is associated with an activity or some other lifecycle owner  and it stays in memory during a configuration change  and the system automatically associates the ViewModel  with the new activity instance that results from the configuration change.  ViewModels are automatically destroyed by the system  when your user backs out of your activity or fragment  or if you call finish(), which means the state is cleared  as the user expects in these scenarios.  Unlike saved instance state, ViewModels are destroyed  during a system-initiated process death.  To reload data after a system-initiated process death in a ViewModel,  we need to use the SavedStateHandle API.  Alternatively, if the data is related to the UI and doesn't need  to be held in the ViewModel, we can use onSaveInstanceState()  in the View system or rememberSaveable in Jetpack Compose.  If the data is application data, then it might be better to persist it to disk.  If we already have an in-memory solution in place for storing  UI state across configuration changes,  we may not need to use ViewModel.  ------------ViewModelStore------------  ViewModelStore is an object which contains a HashMap of String  and ViewModel.key is the default key and value is viewmodel.  When user rotates screen, the activity/fragment’s onCreate will get called,  we will be initializing viewmodel with viewmodelprovider in onCreate only.  Every time it’s creating a new ViewModelProvider,  but the first argument is ViewModelStore.  This viewmodelstore is maintained by activity.  So here if configuration changes happened,  if already a viewmodelstore is available then it will return that instance  else a new viewmodelstore is created.  This is how viewmodel persists on screen rotation.  And when finish is called it activity will call viewmodel.clear(). | </end> |
| <hitle> | Saved instance state for system-initiated process death | <chare> | 3 | <pext> | The onSaveInstanceState() callback in the View system,  rememberSaveable in Jetpack Compose, and SavedStateHandle  in ViewModels store data needed to reload the state of a UI controller,  such as an activity or a fragment, if the system destroys  and later recreates that controller.  Saved instance state bundles persist through  both configuration changes and process death  but they are limited by storage and speed,  because the different APIs serialize data to disk.  Serialization can consume a lot of memory  if the objects being serialized are complicated.  Because this process happens on the main thread  during a configuration change, long-running serialization  can cause dropped frames and visual stutter.  Key Point: Saved instance state APIs only save data written to it  when the Activity is stopped.  Writing into it in between this lifecycle state defers the save operation  till the next stopped lifecycle event.  We don't use saved instance state to store large amounts of data,  such as bitmaps, nor complex data structures  that require lengthy serialization or deserialization.  Instead, store only primitive types and simple, small objects such as String.  As such, we use Saved instance state to store  a minimal amount of data necessary, such as an ID, to re-create  the data necessary to restore the UI back to its previous state.    We might not need to use saved instance state at all.  For example, a browser might take the user  back to the exact webpage they were looking at  before they exited the browser.  In this case, we can forego using saved instance state  and instead persist everything locally.  Key Point: Usually, data stored in saved instance state is transient state  that depends on user input or navigation.  For example, the scroll position of a list,  the id of the item the user wants more detail about,  the in-progress selection of user preferences, or input in text fields.  Additionally, when we open an activity from an intent,  the bundle of extras is delivered to the activity  both when the configuration changes and  when the system restores the activity.  If a piece of UI state data, such as a search query, were passed in  as an intent extra when the activity was launched,  we could use the extras bundle instead of the saved instance state bundle.    In either of these scenarios, we should still use a ViewModel  to avoid wasting cycles reloading data from the database  during a configuration change.  In cases where the UI data to preserve is simple and lightweight,  we might use saved instance state APIs alone to preserve state data. | </end> |
| <hitle> | Hook into saved state using SavedStateRegistry | <chare> | 3 | <pext> | Some of classes such as Activity or Fragment,  implement SavedStateRegistryOwner  and provide a SavedStateRegistry that is bound to that controller.  SavedStateRegistry allows components to hook into  UI controller's saved state to consume or contribute to it.  For example, the Saved State module for ViewModel  uses SavedStateRegistry to create a SavedStateHandle  and provide it to ViewModel objects.  We can retrieve the SavedStateRegistry from within UI controller  by calling getSavedStateRegistry().  Components that contribute to saved state must implement  SavedStateRegistry.SavedStateProvider,  which defines a single method called saveState().  The saveState() method allows component to return a Bundle  containing any state that should be saved from that component.  SavedStateRegistry calls this method during the saving state phase  of the UI controller's lifecycle.  Important: The SavedStateHandle only saves data written to it  when the Activity is stopped.  Writes to SavedStateHandle while the Activity is stopped  aren't saved unless the Activity receives onStart  followed by onStop again.  class SearchManager : SavedStateRegistry.SavedStateProvider {  companion object {  private const val QUERY = "query"  }  private val query: String? = null  ...  override fun saveState(): Bundle {  return bundleOf(QUERY to query)  }  }  To register a SavedStateProvider, call registerSavedStateProvider()  on the SavedStateRegistry, passing a key to associate  with the provider's data as well as the provider.  The previously saved data for the provider can be retrieved  from the saved state by calling consumeRestoredStateForKey()  on the SavedStateRegistry, passing in the key associated  with the provider's data.  Within an Activity or Fragment,  we can register a SavedStateProvider in onCreate()  after calling super.onCreate().  Alternatively, we can set a LifecycleObserver  on a SavedStateRegistryOwner, which implements LifecycleOwner,  and register the SavedStateProvider  once the ON\_CREATE event occurs.  By using a LifecycleObserver, we can decouple the registration  and retrieval of the previously saved state  from the SavedStateRegistryOwner itself.  class SearchManager(registryOwner: SavedStateRegistryOwner) : SavedStateRegistry.SavedStateProvider {  companion object {  private const val PROVIDER = "search\_manager"  private const val QUERY = "query"  }  private val query: String? = null  init {  // Register a LifecycleObserver for when the Lifecycle hits ON\_CREATE  registryOwner.lifecycle.addObserver(LifecycleEventObserver { \_, event ->  if (event == Lifecycle.Event.ON\_CREATE) {  val registry = registryOwner.savedStateRegistry  // Register this object for future calls to saveState()  registry.registerSavedStateProvider(PROVIDER, this)  // Get the previously saved state and restore it  val state = registry.consumeRestoredStateForKey(PROVIDER)  // Apply the previously saved state  query = state?.getString(QUERY)  }  }  }  override fun saveState(): Bundle {  return bundleOf(QUERY to query)  }  ...  }  class SearchFragment : Fragment() {  private var searchManager = SearchManager(this)  ...  } | </end> |
| <hitle> | local persistence to handle process death for complex or large data | <chare> | 3 | <pext> | Persistent local storage, such as a database or shared preferences,  will survive for as long as the application is installed  on the user's device (unless the user clears the data for your app).  While such local storage survives system-initiated activity  and application process death, it can be expensive to retrieve  because it will have to be read from local storage in to memory.  Often this persistent local storage may already  be a part of the application architecture to store all data  we don't want to lose if we open and close the activity.  Neither ViewModel nor saved instance state are long-term storage solutions  and thus are not replacements for local storage, such as a database.  Instead we need to use these mechanisms for temporarily storing  transient UI state only and use persistent storage for other app data. | </end> |
| <hitle> | Manage UI state: divide and conquer | <chare> | 3 | <pext> | we can efficiently save and restore UI state by dividing the work  among the various types of persistence mechanisms.  In most cases, each of these mechanisms should store  a different type of data used in the activity,  based on the tradeoffs of data complexity, access speed, and lifetime:  1. Local persistence: Stores all the application data we don't want to lose  if we open and close the activity.  For example: A collection of song objects,  which could include audio files and metadata.  2. ViewModel: Stores in memory all the data needed to display  the associated the screen UI state.  For example: The song objects of the most recent search  and the most recent search query.  3. Saved instance state: Stores a small amount of data  needed to easily reload UI state if the system stops  and then recreates the UI.  Instead of storing complex objects here, persist the complex objects  in local storage and store a unique ID for these objects  in the saved instance state APIs.  For example: Storing the most recent search query.  As an example, let’s say an activity that allows to search through  the library of songs.  We can see how different events should be handled.  When the user adds a song, the ViewModel immediately  delegates persisting this data locally.  If this newly added song should be shown in the UI,  we should also update the data in the ViewModel object  to reflect the addition of the song.  We have to remember to do all database inserts off of the main thread.  When the user searches for a song, whatever complex song data  we load from the database, it should be immediately stored  in the ViewModel object as part of the screen UI state.  When the activity goes into the background and the system calls  the saved instance state APIs, the search query should be stored  in saved instance state, in case the process recreates.  Since the information is necessary to load application data  persisted in this, store the search query  in the ViewModel SavedStateHandle.  This is all the information we need to load the data  and get the UI back into its current state. | </end> |
| <hitle> | Restoring complex states | <chare> | 3 | <pext> | When it is time for the user to return to the activity,  there are two possible scenarios for recreating the activity:  1. The activity is recreated after having been stopped by the system.  The system has the query saved in an saved instance state bundle,  and the UI should pass the query to the ViewModel  if SavedStateHandle is not used.  The ViewModel sees that it has no search results cached  and delegates loading the search results using the given search query.  2. The activity is created after a configuration change.  Since the ViewModel instance hasn't been destroyed,  the ViewModel has all the information cached in memory  and it doesn't need to re-query the database.  Note: When an activity is initially created, the saved instance state bundle  contains no data, and the ViewModel object is empty.  When we create the ViewModel object, you pass an empty query,  which tells the ViewModel object that there's no data to load yet.  Therefore, the activity starts in an empty state. | </end> |
| <hitle> | Lifecycle-aware coroutine scopes | <chare> | 2 | <pext> | There are two Lifecycle-aware coroutine scope components.  1. ViewModelScope  A ViewModelScope is defined for each ViewModel in your app.  Any coroutine launched in this scope is automatically canceled  if the ViewModel is cleared.  Coroutines are useful here for when we have work  that needs to be done only if the ViewModel is active.  2. LifecycleScope  A LifecycleScope is defined for each Lifecycle object.  Any coroutine launched in this scope is canceled  when the Lifecycle is destroyed.  We can access the CoroutineScope of the Lifecycle  either via lifecycle.coroutineScope or lifecycleOwner.lifecycleScope properties. | </end> |
| <hitle> | Lifecycle-aware coroutine LifecycleScope | <chare> | 3 | <pext> | A LifecycleScope is defined for each Lifecycle object.  Any coroutine launched in this scope is canceled  when the Lifecycle is destroyed.  We can access the CoroutineScope of the Lifecycle  either via lifecycle.coroutineScope or lifecycleOwner.lifecycleScope properties.  class MyFragment: Fragment() {  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  super.onViewCreated(view, savedInstanceState)  viewLifecycleOwner.lifecycleScope.launch {  val params = TextViewCompat.getTextMetricsParams(textView)  val precomputedText = withContext(Dispatchers.Default) {  PrecomputedTextCompat.create(longTextContent, params)  }  TextViewCompat.setPrecomputedText(textView, precomputedText)  }  }  }  (\*It shows how to use lifecycleOwner.lifecycleScope to create precomputed text asynchronously.\*) | </end> |
| <hitle> | Lifecycle-aware coroutine ViewModelScope | <chare> | 3 | <pext> | A ViewModelScope is defined for each ViewModel in your app.  Any coroutine launched in this scope is automatically canceled  if the ViewModel is cleared.  Coroutines are useful here for when we have work  that needs to be done only if the ViewModel is active.  For example, if we are computing some data for a layout,  we should scope the work to the ViewModel so that  if the ViewModel is cleared, the work is canceled automatically  to avoid consuming resources.  We can access the CoroutineScope of a ViewModel  through the viewModelScope property of the ViewModel.  class MyViewModel: ViewModel() {  init {  viewModelScope.launch {  // Coroutine that will be canceled when the ViewModel is cleared.  }  }  } | </end> |
| <hitle> | Restartable Lifecycle-aware coroutines | <chare> | 2 | <pext> | Even though the lifecycleScope provides a proper way to cancel  long-running operations automatically when the Lifecycle is DESTROYED,  we might have other cases where we want to start the execution  of a code block when the Lifecycle is in a certain state,  and cancel when it is in another state.  For example, we might want to collect a flow  when the Lifecycle is STARTED and cancel the collection  when it's STOPPED.  This approach processes the flow emissions  only when the UI is visible on the screen,  saving resources and potentially avoiding app crashes.  For these cases, Lifecycle and LifecycleOwner provide  the suspend repeatOnLifecycle API that does exactly that.  class MyFragment : Fragment() {  val viewModel: MyViewModel by viewModel()  override fun onViewCreated(view: View, savedInstanceState: Bundle?) {  super.onViewCreated(view, savedInstanceState)  // Create a new coroutine in the lifecycleScope  viewLifecycleOwner.lifecycleScope.launch {  // repeatOnLifecycle launches the block in a new coroutine every time the  // lifecycle is in the STARTED state (or above) and cancels it when it's STOPPED.  viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {  // Trigger the flow and start listening for values.  // This happens when lifecycle is STARTED and stops  // collecting when the lifecycle is STOPPED  viewModel.someDataFlow.collect {  // Process item  }  }  }  }  }  (It shows a code block that runs every time the associated Lifecycle is at least in the STARTED state and cancels when the Lifecycle is STOPPED.) | </end> |
| <hitle> | Lifecycle-aware flow collection | <chare> | 3 | <pext> | If we only need to perform lifecycle-aware collection on a single flow,  we can use the Flow.flowWithLifecycle() method.  viewLifecycleOwner.lifecycleScope.launch {  exampleProvider.exampleFlow()  .flowWithLifecycle(viewLifecycleOwner.lifecycle, Lifecycle.State.STARTED)  .collect {  // Process the value.  }  }  However, if we need to perform lifecycle-aware collection  on multiple flows in parallel, then we must collect each flow  in different coroutines.  In that case, it's more efficient to use repeatOnLifecycle() directly.  viewLifecycleOwner.lifecycleScope.launch {  viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED) {  // Because collect is a suspend function, if you want to  // collect multiple flows in parallel, you need to do so in  // different coroutines.  launch {  flow1.collect { /\* Process the value. \*/ }  }  launch {  flow2.collect { /\* Process the value. \*/ }  }  }  } | </end> |
| <hitle> | Suspend Lifecycle-aware coroutines | <chare> | 2 | <pext> | Even though the CoroutineScope provides a proper way to cancel  long-running operations automatically, we might have other cases  where we want to suspend execution of a code block  unless the Lifecycle is in a certain state.  For example, to run a FragmentTransaction, we must wait  until the Lifecycle is at least STARTED.  For these cases, Lifecycle provides additional methods:  lifecycle.whenCreated, lifecycle.whenStarted, and lifecycle.whenResumed.  Any coroutine run inside these blocks is suspended  if the Lifecycle isn't at least in the minimal desired state.  class MyFragment: Fragment {  init { // Notice that we can safely launch in the constructor of the Fragment.  lifecycleScope.launch {  whenStarted {  // The block inside will run only when Lifecycle is at least STARTED.  // It will start executing when fragment is started and  // can call other suspend methods.  loadingView.visibility = View.VISIBLE  val canAccess = withContext(Dispatchers.IO) {  checkUserAccess()  }  // When checkUserAccess returns, the next line is automatically  // suspended if the Lifecycle is not \*at least\* STARTED.  // We could safely run fragment transactions because we know the  // code won't run unless the lifecycle is at least STARTED.  loadingView.visibility = View.GONE  if (canAccess == false) {  findNavController().popBackStack()  } else {  showContent()  }  }  // This line runs only after the whenStarted block above has completed.  }  }  }  (\*It shows a code block that runs only when the associated Lifecycle is at least in the STARTED state.)  If the Lifecycle is destroyed while a coroutine is active  via one of the when methods, the coroutine is automatically canceled.  class MyFragment: Fragment {  init {  lifecycleScope.launchWhenStarted {  try {  // Call some suspend functions.  } finally {  // This line might execute after Lifecycle is DESTROYED.  if (lifecycle.state >= STARTED) {  // Here, since we've checked, it is safe to run any  // Fragment transactions.  }  }  }  }  }  (\*In the example below, the finally block runs once the Lifecycle state is DESTROYED.\*)  Note: Even though these methods provide convenience  when working with Lifecycle, we should use them  only when the information is valid within the scope of the Lifecycle  (precomputed text, for example).  If the activity restarts, the coroutine is not restarted.  Warning: It it good to use collecting flows using the repeatOnLifecycle API  instead of collecting inside the launchWhenX APIs.  As the latter APIs suspend the coroutine instead of cancelling it  when the Lifecycle is STOPPED, upstream flows are kept active  in the background, potentially emitting new items and wasting resources. | </end> |
| <hitle> | coroutines with LiveData | <chare> | 2 | <pext> | When using LiveData, we might need to calculate values asynchronously.  For example, we might want to retrieve a user's preferences  and serve them to your UI.  In these cases, we can use the liveData builder function  to call a suspend function, serving the result as a LiveData object.  val user: LiveData<User> = liveData {  val data = database.loadUser() // loadUser is a suspend function.  emit(data)  }  (\*loadUser() is a suspend function declared elsewhere. Use the liveData builder function to call loadUser() asynchronously, and then use emit() to emit the result.\*)  The liveData building block serves as a structured concurrency primitive  between coroutines and LiveData.  The code block starts executing when LiveData becomes active  and is automatically canceled after a configurable timeout  when the LiveData becomes inactive.  If it is canceled before completion, it is restarted  if the LiveData becomes active again.  If it completed successfully in a previous run, it doesn't restart.  Note: It is restarted only if canceled automatically.  If the block is canceled for any other reason  like throwing a CancellationException, it is not restarted.  We can also emit multiple values from the block.  Each emit() call suspends the execution of the block  until the LiveData value is set on the main thread.  val user: LiveData<Result> = liveData {  emit(Result.loading())  try {  emit(Result.success(fetchUser()))  } catch(ioException: Exception) {  emit(Result.error(ioException))  }  }  You can also combine liveData with Transformations.  class MyViewModel: ViewModel() {  private val userId: LiveData<String> = MutableLiveData()  val user = userId.switchMap { id ->  liveData(context = viewModelScope.coroutineContext + Dispatchers.IO) {  emit(database.loadUserById(id))  }  }  }  You can emit multiple values from a LiveData  by calling the emitSource() function  whenever you want to emit a new value.  Note: Each call to emit() or emitSource() removes the previously-added source.  class UserDao: Dao {  @Query("SELECT \* FROM User WHERE id = :id")  fun getUser(id: String): LiveData<User>  }  class MyRepository {  fun getUser(id: String) = liveData<User> {  val disposable = emitSource(  userDao.getUser(id).map {  Result.loading(it)  }  )  try {  val user = webservice.fetchUser(id)  // Stop the previous emission to avoid dispatching the updated user  // as `loading`.  disposable.dispose()  // Update the database.  userDao.insert(user)  // Re-establish the emission with success type.  emitSource(  userDao.getUser(id).map {  Result.success(it)  }  )  } catch(exception: IOException) {  // Any call to `emit` disposes the previous one automatically so we don't  // need to dispose it here as we didn't get an updated value.  emitSource(  userDao.getUser(id).map {  Result.error(exception, it)  }  )  }  }  } | </end> |
| <hitle> | Paging | <chare> | 1 | <pext> | The Paging library let us load and display pages of data  from a larger dataset from local storage or over network.  This approach allows the app to use both network bandwidth  and system resources more efficiently.  The components of the Paging library are designed to fit  into the recommended Android app architecture,  integrate cleanly with other Jetpack components,  and provide first-class Kotlin support. | </end> |
| <hitle> | Paging pros | <chare> | 2 | <pext> | 1. In-memory caching for your paged data.  This ensures that your app uses system resources efficiently  while working with paged data.  2. Built-in request deduplication, ensuring that the app  uses network bandwidth and system resources efficiently.  3. Configurable RecyclerView adapters that automatically request data  as the user scrolls toward the end of the loaded data.  4. First-class support for Kotlin coroutines and Flow,  as well as LiveData and RxJava.  Built-in support for error handling, including refresh and retry capabilities. | </end> |
| <hitle> | Paging Library architecture | <chare> | 2 | <pext> | The Paging library integrates directly  into the recommended Android app architecture.  The library's components operate in three layers of the app:  1. The repository layer  2. The ViewModel layer  3. The UI layer | </end> |
| <hitle> | Paging architecture repository layer | <chare> | 3 | <pext> | The primary Paging library component in the repository layer  is PagingSource.  Each PagingSource object defines a source of data  and how to retrieve data from that source.  A PagingSource object can load data from any single source,  including network sources and local databases.  Another Paging library component that we might use  is RemoteMediator.  A RemoteMediator object handles paging from a layered data source,  such as a network data source with a local database cache. | </end> |
| <hitle> | Paging architecture ViewModel layer | <chare> | 3 | <pext> | The Pager component provides a public API  for constructing instances of PagingData  that are exposed in reactive streams,  based on a PagingSource object and a PagingConfig configuration object.  The component that connects the ViewModel layer  to the UI is PagingData.  A PagingData object is a container for a snapshot of paginated data.  It queries a PagingSource object and stores the result. | </end> |
| <hitle> | Paging architecture UI layer | <chare> | 3 | <pext> | The primary Paging library component in the UI layer  is PagingDataAdapter, a RecyclerView adapter  that handles paginated data.  Alternatively, we can use the included AsyncPagingDataDiffer component  to build your own custom adapter.  Note: If the app uses Compose for UI, we can use  the androidx.paging:paging-compose artifact to integrate Paging  with UI layer instead. | </end> |
| <hitle> | Load and display paged data | <chare> | 2 | <pext> | 1. Define a data source  The first step is to define a PagingSource implementation  to identify the data source.  The PagingSource API class includes the load() method,  which we must override to indicate how to retrieve paged data  from the corresponding data source.  2. Next, we need a stream of paged data  from the PagingSource implementation.  Typically, we should set up the data stream in your ViewModel.  The Pager class provides methods that expose a reactive stream  of PagingData objects from a PagingSource.  The Paging library supports using several stream types,  including Flow, LiveData, and the Flowable and Observable types from RxJava.  3. We also need to set up an adapter to receive the data  into RecyclerView list.  The Paging library provides the PagingDataAdapter class for this purpose.  4. Finally we have defined a PagingSource, created a way for the app  to generate a stream of PagingData,  and defined a PagingDataAdapter,  we need to connect these elements together and display paged data  in your activity. | </end> |
| <hitle> | Define a data source | <chare> | 3 | <pext> | To implement Paging, the first step is to define  a PagingSource implementation to identify the data source.  The PagingSource API class includes the load() method,  which we must override to indicate how to retrieve paged data  from the corresponding data source.  We can use the PagingSource class directly to use Kotlin coroutines  for async loading.  The Paging library also provides classes to support other async frameworks:  1. To use RxJava, we can implement RxPagingSource instead.  2. To use ListenableFuture from Guava,  we can implement ListenableFuturePagingSource instead.  How to select key and value types?  PagingSource<Key, Value> has two type parameters: Key and Value.  The key defines the identifier used to load the data,  and the value is the type of the data itself.  For example, if we load pages of User objects from the network  by passing Int page numbers to Retrofit,  we would select Int as the Key type and User as the Value type.  How to Define the PagingSource?  class ExamplePagingSource(  val backend: ExampleBackendService,  val query: String  ) : PagingSource<Int, User>() {  override suspend fun load(  params: LoadParams<Int>  ): LoadResult<Int, User> {  try {  // Start refresh at page 1 if undefined.  val nextPageNumber = params.key ?: 1  val response = backend.searchUsers(query, nextPageNumber)  return LoadResult.Page(  data = response.users,  prevKey = null, // Only paging forward.  nextKey = response.nextPageNumber  )  } catch (e: Exception) {  // Handle errors in this block and return LoadResult.Error if it is an  // expected error (such as a network failure).  }  }  override fun getRefreshKey(state: PagingState<Int, User>): Int? {  // Try to find the page key of the closest page to anchorPosition, from  // either the prevKey or the nextKey, but you need to handle nullability  // here:  // \* prevKey == null -> anchorPage is the first page.  // \* nextKey == null -> anchorPage is the last page.  // \* both prevKey and nextKey null -> anchorPage is the initial page, so  // just return null.  return state.anchorPosition?.let { anchorPosition ->  val anchorPage = state.closestPageToPosition(anchorPosition)  anchorPage?.prevKey?.plus(1) ?: anchorPage?.nextKey?.minus(1)  }  }  }  (\*It shows implementing a PagingSource that loads pages of items by page number. The Key type is Int and the Value type is User.\*)  A typical PagingSource implementation passes parameters  provided in its constructor to the load() method  to load appropriate data for a query.  For example, those parameters can be:  1. backend: An instance of the backend service that provides the data.  2. query: The search query to send to the service indicated by backend.  The LoadParams object contains information about the load operation  to be performed.  This includes the key to be loaded and the number of items to be loaded.  The LoadResult object contains the result of the load operation.  LoadResult is a sealed class that takes one of two forms,  depending on whether the load() call succeeded:  1. If the load is successful, return a LoadResult.Page object.  2. If the load is not successful, return a LoadResult.Error object.  The PagingSource implementation must also implement  a getRefreshKey() method which takes a PagingState object as a parameter  and returns the key to pass into the load() method  when the data is refreshed or invalidated after the initial load.  The Paging Library calls this method automatically  on subsequent refreshes of the data.  How to handle errors?  Requests to load data can fail for a number of reasons,  especially when loading over network.  We can report errors encountered during loading by returning  a LoadResult.Error object from the load() method.  catch (e: IOException) {  // IOException for network failures.  return LoadResult.Error(e)  } catch (e: HttpException) {  // HttpException for any non-2xx HTTP status codes.  return LoadResult.Error(e)  }  PagingSource collects and delivers LoadResult.Error objects to the UI  so that you can act upon them. | </end> |
| <hitle> | Set up stream of PagingData | <chare> | 3 | <pext> | We need a stream of paged data from  the PagingSource implementation.  Typically, we should set up the data stream in your ViewModel.  The Pager class provides methods that expose a reactive stream  of PagingData objects from a PagingSource.  The Paging library supports using several stream types,  including Flow, LiveData, and the Flowable and  Observable types from RxJava.  When we create a Pager instance to set up your reactive stream,  we must provide the instance with a PagingConfig configuration object  and a function that tells Pager how to get  an instance of PagingSource implementation:  val flow = Pager(  // Configure how data is loaded by passing additional properties to  // PagingConfig, such as prefetchDistance.  PagingConfig(pageSize = 20)  ) {  ExamplePagingSource(backend, query)  }.flow  .cachedIn(viewModelScope)  (\*The cachedIn() operator makes the data stream shareable  and caches the loaded data with the provided CoroutineScope.  It uses the viewModelScope.\*)  The Pager object calls the load() method from the PagingSource object,  providing it with the LoadParams object  and receiving the LoadResult object in return. | </end> |
| <hitle> | Define RecyclerView adapter | <chare> | 3 | <pext> | We need to set up an adapter to receive the data  into RecyclerView list.  The Paging library provides the PagingDataAdapter class  for this purpose.  We need to define a class that extends PagingDataAdapter.  class UserAdapter(diffCallback: DiffUtil.ItemCallback<User>) :  PagingDataAdapter<User, UserViewHolder>(diffCallback) {  override fun onCreateViewHolder(  parent: ViewGroup,  viewType: Int  ): UserViewHolder {  return UserViewHolder(parent)  }  override fun onBindViewHolder(holder: UserViewHolder, position: Int) {  val item = getItem(position)  // Note that item may be null. ViewHolder must support binding a  // null item as a placeholder.  holder.bind(item)  }  }  (\*UserAdapter extends PagingDataAdapter to provide  a RecyclerView adapter for list items of type User  and using UserViewHolder as a view holder.\*)  The adapter must also define the onCreateViewHolder()  and onBindViewHolder() methods and specify a DiffUtil.ItemCallback.  This works the same as it normally does  when defining RecyclerView list adapters.  object UserComparator : DiffUtil.ItemCallback<User>() {  override fun areItemsTheSame(oldItem: User, newItem: User): Boolean {  // Id is unique.  return oldItem.id == newItem.id  }  override fun areContentsTheSame(oldItem: User, newItem: User): Boolean {  return oldItem == newItem  }  } | </end> |
| <hitle> | Display paged data | <chare> | 3 | <pext> | We have defined a PagingSource, created a way  to generate a stream of PagingData, and defined a PagingDataAdapter,  we can connect these elements together and display paged data.  We need to do some steps in activity's onCreate  or fragment's onViewCreated method.  1. Create an instance of PagingDataAdapter class.  2. Pass the PagingDataAdapter instance to the RecyclerView list  that you want to display your paged data.  3. Observe the PagingData stream, and pass each generated value  to your adapter's submitData() method.  val viewModel by viewModels<ExampleViewModel>()  val pagingAdapter = UserAdapter(UserComparator)  val recyclerView = findViewById<RecyclerView>(R.id.recycler\_view)  recyclerView.adapter = pagingAdapter  // Activities can use lifecycleScope directly, but Fragments should instead use  // viewLifecycleOwner.lifecycleScope.  lifecycleScope.launch {  viewModel.flow.collectLatest { pagingData ->  pagingAdapter.submitData(pagingData)  }  }  The RecyclerView list now displays the paged data from the data source  and automatically loads another page when necessary. | </end> |
| <hitle> | DataStore | <chare> | 1 | <pext> | Jetpack DataStore is a data storage solution that allows to store  key-value pairs or typed objects with protocol buffers.  DataStore uses Kotlin coroutines and Flow to store data asynchronously,  consistently, and transactionally.  If we're currently using SharedPreferences to store data,  we need to consider migrating to DataStore instead.  Note: If we need to support large or complex datasets, partial updates,  or referential integrity, we need to consider using Room  instead of DataStore.  DataStore is ideal for small, simple datasets and does not support  partial updates or referential integrity. | </end> |
| <hitle> | DataStore Rules | <chare> | 2 | <pext> | 1. Never create more than one instance of DataStore for a given file  in the same process.  Doing so can break all DataStore functionality.  If there are multiple DataStores active for a given file in the same process,  DataStore will throw IllegalStateException  when reading or updating data.  2. The generic type of the DataStore must be immutable.  Mutating a type used in DataStore invalidates any guarantees  that DataStore provides and creates potentially serious,  hard-to-catch bugs.  It is strongly recommended that we use protocol buffers  which provide immutability guarantees,  a simple API and efficient serialization.  3. Never mix usages of SingleProcessDataStore  and MultiProcessDataStore for the same file.  If we intend to access the DataStore from more than one process  always use MultiProcessDataStore. | </end> |
| <hitle> | Preferences DataStore vs Proto DataStore | <chare> | 2 | <pext> | 1. Preferences DataStore stores and accesses data using keys.  This implementation does not require a predefined schema,  and it does not provide type safety.  2. Proto DataStore stores data as instances of a custom data type.  This implementation requires to define a schema using protocol buffers,  but it provides type safety. | </end> |
| <hitle> | DataStore gradle | <chare> | 2 | <pext> | 1. Preferences DataStore  // Preferences DataStore (SharedPreferences like APIs)  dependencies {  implementation "androidx.datastore:datastore-preferences:1.0.0"  // optional - RxJava2 support  implementation "androidx.datastore:datastore-preferences-rxjava2:1.0.0"  // optional - RxJava3 support  implementation "androidx.datastore:datastore-preferences-rxjava3:1.0.0"  }  // Alternatively - use the following artifact without an Android dependency.  dependencies {  implementation "androidx.datastore:datastore-preferences-core:1.0.0"  }  2. Proto DataStore  // Typed DataStore (Typed API surface, such as Proto)  dependencies {  implementation "androidx.datastore:datastore:1.0.0"  // optional - RxJava2 support  implementation "androidx.datastore:datastore-rxjava2:1.0.0"  // optional - RxJava3 support  implementation "androidx.datastore:datastore-rxjava3:1.0.0"  }  // Alternatively - use the following artifact without an Android dependency.  dependencies {  implementation "androidx.datastore:datastore-core:1.0.0"  } | </end> |
| <hitle> | Preferences DataStore | <chare> | 2 | <pext> | Preferences DataStore stores and accesses data using keys.  This implementation does not require a predefined schema,  and it does not provide type safety.  The Preferences DataStore implementation uses  the DataStore and Preferences classes to persist  simple key-value pairs to disk. | </end> |
| <hitle> | Create Preferences DataStore | <chare> | 3 | <pext> | We can use the property delegate created by preferencesDataStore  to create an instance of Datastore<Preferences>.  We can call it once at the top level of kotlin file,  and access it through this property throughout the rest of application.  This makes it easier to keep DataStore as a singleton.  Alternatively, we can use RxPreferenceDataStoreBuilder  if we're using RxJava.  The mandatory name parameter  is the name of the Preferences DataStore.  // At the top level of kotlin file:  val Context.dataStore: DataStore<Preferences> by preferencesDataStore(name = "settings") | </end> |
| <hitle> | Read from Preferences DataStore | <chare> | 3 | <pext> | Because Preferences DataStore does not use a predefined schema,  we must use the corresponding key type function to define a key  for each value that we need to store  in the DataStore<Preferences> instance.  For example, to define a key for an int value,  we can use intPreferencesKey().  Then, we can use the DataStore.data property to expose  the appropriate stored value using a Flow.  val EXAMPLE\_COUNTER = intPreferencesKey("example\_counter")  val exampleCounterFlow: Flow<Int> = context.dataStore.data  .map { preferences ->  // No type safety.  preferences[EXAMPLE\_COUNTER] ?: 0  } | </end> |
| <hitle> | Write to Preferences DataStore | <chare> | 3 | <pext> | Preferences DataStore provides an edit() function  that transactionally updates the data in a DataStore.  The function's transform parameter accepts a block of code  where we can update the values as needed.  All of the code in the transform block is treated as a single transaction.  suspend fun incrementCounter() {  context.dataStore.edit { settings ->  val currentCounterValue = settings[EXAMPLE\_COUNTER] ?: 0  settings[EXAMPLE\_COUNTER] = currentCounterValue + 1  }  } | </end> |
| <hitle> | Proto DataStore | <chare> | 2 | <pext> | Proto DataStore stores data as instances of a custom data type.  This implementation requires you to define a schema using protocol buffers,  but it provides type safety.  The Proto DataStore implementation uses DataStore and protocol buffers  to persist typed objects to disk. | </end> |
| <hitle> | Define a schema Proto DataStore | <chare> | 3 | <pext> | Proto DataStore requires a predefined schema in a proto file  in the app/src/main/proto/ directory.  This schema defines the type for the objects  that we persist in your Proto DataStore.  More about defining a proto schema,  we need to check the protobuf language guide.  https://developers.google.com/protocol-buffers/docs/proto3  syntax = "proto3";  option java\_package = "com.example.application";  option java\_multiple\_files = true;  message Settings {  int32 example\_counter = 1;  }  Note: The class for stored objects is generated at compile time  from the message defined in the proto file.  That means we need rebuild project. | </end> |
| <hitle> | Create Proto DataStore | <chare> | 3 | <pext> | There are two steps involved in creating a Proto DataStore  to store your typed objects:  1. We need to define a class that implements Serializer<T>,  where T is the type defined in the proto file.  This serializer class tells DataStore how to read and write your data type.  We need to include a default value for the serializer to be used  if there is no file created yet.  2. We need to use the property delegate created by dataStore  to create an instance of DataStore<T>,  where T is the type defined in the proto file.  We can call this once at the top level of the kotlin file and access it  through this property delegate throughout the rest of the app.  The filename parameter tells DataStore which file to use to store the data,  and the serializer parameter tells DataStore the name  of the serializer class we defined already in previous step.  object SettingsSerializer : Serializer<Settings> {  override val defaultValue: Settings = Settings.getDefaultInstance()  override suspend fun readFrom(input: InputStream): Settings {  try {  return Settings.parseFrom(input)  } catch (exception: InvalidProtocolBufferException) {  throw CorruptionException("Cannot read proto.", exception)  }  }  override suspend fun writeTo(  t: Settings,  output: OutputStream) = t.writeTo(output)  }  val Context.settingsDataStore: DataStore<Settings> by dataStore(  fileName = "settings.pb",  serializer = SettingsSerializer  ) | </end> |
| <hitle> | Read from Proto DataStore | <chare> | 3 | <pext> | We can use DataStore.data to expose a Flow  of the appropriate property from stored object.  val exampleCounterFlow: Flow<Int> = context.settingsDataStore.data  .map { settings ->  // The exampleCounter property is generated from the proto schema.  settings.exampleCounter  } | </end> |
| <hitle> | Write Proto DataStore | <chare> | 3 | <pext> | Proto DataStore provides an updateData() function  that transactionally updates a stored object.  updateData() gives the current state of the data  as an instance of data type and updates the data transactionally  in an atomic read-write-modify operation.  suspend fun incrementCounter() {  context.settingsDataStore.updateData { currentSettings ->  currentSettings.toBuilder()  .setExampleCounter(currentSettings.exampleCounter + 1)  .build()  }  } | </end> |
| <hitle> | DataStore in synchronous | <chare> | 2 | <pext> | One of the primary benefits of DataStore is the asynchronous API,  but it may not always be feasible to change surrounding code  to be asynchronous.  This might be the case if we're working with an existing codebase  that uses synchronous disk I/O or if we have a dependency  that doesn't provide an asynchronous API.  Kotlin coroutines provide the runBlocking() coroutine builder  to help bridge the gap between synchronous and asynchronous code.  We can use runBlocking() to read data from DataStore synchronously.  RxJava offers blocking methods on Flowable.  val exampleData = runBlocking { context.dataStore.data.first() }  (\*It is calling thread until DataStore returns data.\*)  Performing synchronous I/O operations on the UI thread can  cause ANRs or UI jank.  We can mitigate these issues  by asynchronously preloading the data from DataStore:  override fun onCreate(savedInstanceState: Bundle?) {  lifecycleScope.launch {  context.dataStore.data.first()  // You should also handle IOExceptions here.  }  }  By doing that, DataStore asynchronously reads the data  and caches it in memory.  Later synchronous reads using runBlocking() may be faster  or may avoid a disk I/O operation altogether  if the initial read has completed. | </end> |
| <hitle> | DataStore in multi-process | <chare> | 2 | <pext> | We can configure DataStore to access the same data  across different processes with the same data consistency guarantees  as from within a single process.  In particular, DataStore guarantees:  1. DataStore guarantees Reads only return the data  that has been persisted to disk.  2. DataStore guarantees Read-after-write consistency.  3. DataStore guarantees Writes are serialized.  4. DataStore guarantees Reads are never blocked by writes.  Let’s say we have a sample application with a service and an activity:  1. The service is running in a separate process and periodically  updates the DataStore.  <service  android:name=".MyService"  android:process=":my\_process\_id" />  Important: To run the service in a different process,  we need to use the android:process attribute.  Note: the process id is prefixed with a colon (':').  This makes the service run in a new process,  private to the application.  override fun onStartCommand(intent: Intent?, flags: Int, startId: Int): Int {  scope.launch {  while(isActive) {  dataStore.updateData {  Settings(lastUpdate = System.currentTimeMillis())  }  delay(1000)  }  }  }  2. While the app would collect those changes and update its UI.  val settings: Settings by dataStore.data.collectAsState()  Text(  text = "Last updated: $${settings.timestamp}",  )  To be able to use DataStore across different processes,  we need to construct the DataStore object  using the MultiProcessDataStoreFactory.  val dataStore: DataStore<Settings> = MultiProcessDataStoreFactory.create(  serializer = SettingsSerializer(),  produceFile = {  File("${context.cacheDir.path}/myapp.preferences\_pb")  }  )  serializer tells DataStore how to read and write your data type.  We need to include a default value for the serializer to be used  if there is no file created yet.  @Serializable  data class Settings(  val lastUpdate: Long  )  @Singleton  class SettingsSerializer @Inject constructor() : Serializer<Settings> {  override val defaultValue = Settings(lastUpdate = 0)  override suspend fun readFrom(input: InputStream): Timer =  try {  Json.decodeFromString(  Settings.serializer(), input.readBytes().decodeToString()  )  } catch (serialization: SerializationException) {  throw CorruptionException("Unable to read Settings", serialization)  }  override suspend fun writeTo(t: Settings, output: OutputStream) {  output.write(  Json.encodeToString(Settings.serializer(), t)  .encodeToByteArray()  )  }  }  We can use Hilt dependency injection to make sure that DataStore instance  is unique per process:  @Provides  @Singleton  fun provideDataStore(@ApplicationContext context: Context): DataStore<Settings> =  MultiProcessDataStoreFactory.create(...) | </end> |
| <hitle> | WorkManager | <chare> | 1 | <pext> | WorkManager is the recommended solution for persistent work.  Work is persistent when it remains scheduled  through app restarts and system reboots.  Because most background processing is best accomplished  through persistent work,  WorkManager is the primary recommended API  for background processing. | </end> |
| <hitle> | Persistent work | <chare> | 2 | <pext> | Work is persistent when it remains scheduled through app restarts  and system reboots.  WorkManager is the recommended solution for persistent work.  Because most background processing is best accomplished  through persistent work,  WorkManager is therefore also the primary recommended API  for background processing in general. | </end> |
| <hitle> | Types of persistent work | <chare> | 2 | <pext> | 1. Immediate: Tasks that must begin immediately and complete soon.  It may be expedited. Periodicity is One time.  We can access by OneTimeWorkRequest and Worker.  For expedited work, we need to call setExpedited()  on OneTimeWorkRequest.  2. Long Running: Tasks which might run for longer,  potentially longer than 10 minutes.  Periodicity is One time or periodic.  We can access by any WorkRequest or Worker.  We need to call setForeground() in the Worker  to handle the notification.  3. Deferrable: Scheduled tasks that start at a later time  and can run periodically.  Periodicity is One time or periodic.  We can access by PeriodicWorkRequest and Worker. | </end> |
| <hitle> | WorkManager Features | <chare> | 2 | <pext> | 1. Work constraints  Declaratively define the optimal conditions for work  to run using work constraints.  For example, run only when the device is on an unmetered network,  when the device is idle, or when it has sufficient battery.  2. Robust scheduling  WorkManager allows to schedule work to run one-time or repeatedly  using flexible scheduling windows.  Work can be tagged and named as well, allowing you to schedule unique,  replaceable work and monitor or cancel groups of work together.  Scheduled work is stored in an internally managed SQLite database  and WorkManager takes care of ensuring that this work persists  and is rescheduled across device reboots.  In addition, WorkManager adheres to power-saving features  and best practices like Doze mode, so you don't have to worry about it.  3. Expedited work  We can use WorkManager to schedule immediate work  for execution in the background.  We should use Expedited work for tasks that are important to the user  and which complete within a few minutes.  4. Flexible retry policy  Sometimes work fails. WorkManager offers flexible retry policies,  including a configurable exponential backoff policy.  5. Work chaining  For complex related work, chain individual work tasks together  using an intuitive interface that allows to control which pieces  run sequentially and which run in parallel.  val continuation = WorkManager.getInstance(context)  .beginUniqueWork(  Constants.IMAGE\_MANIPULATION\_WORK\_NAME,  ExistingWorkPolicy.REPLACE,  OneTimeWorkRequest.from(CleanupWorker::class.java)  ).then(OneTimeWorkRequest.from(WaterColorFilterWorker::class.java))  .then(OneTimeWorkRequest.from(GrayScaleFilterWorker::class.java))  .then(OneTimeWorkRequest.from(BlurEffectFilterWorker::class.java))  .then(  if (save) {  workRequest<SaveImageToGalleryWorker>(tag = Constants.TAG\_OUTPUT)  } else /\* upload \*/ {  workRequest<UploadWorker>(tag = Constants.TAG\_OUTPUT)  }  )  For each work task, we can define input and output data for that work.  When chaining work together,  WorkManager automatically passes output data from one work task  to the next.  6. Built-In threading interoperability  WorkManager integrates seamlessly with Coroutines and RxJava  and provides the flexibility to plug in your own asynchronous APIs. | </end> |
| <hitle> | WorkManager for reliable work | <chare> | 2 | <pext> | WorkManager is intended for work that is required to run reliably  even if the user navigates off a screen, the app exits,  or the device restarts.  For example:  1. Sending logs or analytics to backend services.  2. Periodically syncing application data with a server.  WorkManager is not intended for in-process background work  that can safely be terminated if the app process goes away.  It is also not a general solution for all work that requires  immediate execution. | </end> |