|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | Jetpack | <chare> | 1 | <pext> | <starting>Jetpack</heading> Android Jetpack is a set of software components  that help you accelerate your app development.  These components provide a range of features,   including a new set of tools and libraries,  that make it easier to develop high-quality apps.  Android Jetpack is a set of libraries, tools and guidance to help  make it easier to write high-quality, robust, and maintainable apps. One way it helps to reduce boilerplate code is by providing libraries  that handle common tasks, such as navigation, lifecycle management,  and data binding.  This means that you don’t have to write as much code  to handle these tasks yourself,  which can help to reduce the amount of boilerplate in your app. </ending> | </end> |
| <hitle> | Jetpack pros | <chare> | 1 | <pext> | <starting>Jetpack pros</heading> 1. Forms a recommended way for app architecture through its components 2. Eliminate boilerplate code 3. Simplify complex task 4. Provide backward compatibility as libraries like support are unbundled  from Android API and are re-packaged to androidx.\* package 5. Inbuilt productivity feature of the Kotlin Integration </ending> | </end> |
| <hitle> | Jetpack- ViewModel | <chare> | 1 | <pext> | <starting>Jetpack- ViewModel</heading> 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.  </ending> | </end> |
| <hitle> | Jetpack- Room | <chare> | 1 | <pext> | <starting>Jetpack- Room</heading> The requirement of a database in Android is fulfilled by SQLite  from the very beginning.  However, it comes with some severe drawbacks like not checking the queries  at compile-time, it does not save POJOs.  Developers also need to write a lot of boilerplate code  to make the SQLite database.  The Room component comes into the picture  as an SQLite Object Mapping Library which overcomes  all the mentioned challenges.  Room converts queries directly into objects,  checks errors in queries at the compile-time,  and is also capable of persisting the Java POJOs.  Moreover, it produces LiveData results/observables  from the given query result.  Because of this versatile nature of the Room component,  Google officially supports and recommends developers to use it.  The Room consists of sub-components: 1. Entity: It is the annotated class for which the Room creates a table  within the database.  The field of the class represents columns in the table. 2. DAO(Data Access Object): It is responsible for defining the methods  to access the database and to perform operations. 3. Database: It is an abstract class that extends RoomDatabase class  and it serves as the main access point to the underlying app’s relational data. </ending> | </end> |
| <hitle> | Room Entity | <chare> | 1 | <pext> | <starting>Room Entity</heading> Entity: It is the annotated class for which the Room creates a table  within the database. </ending> | </end> |
| <hitle> | Room DAO | <chare> | 1 | <pext> | <starting>Room DAO</heading> DAO(Data Access Object): It is responsible for defining the methods  to access the database and to perform operations. </ending> | </end> |
| <hitle> | Room Database | <chare> | 1 | <pext> | <starting>Room Database</heading> Database: It is an abstract class that extends RoomDatabase class  and it serves as the main access point to the underlying app’s relational data. </ending> | </end> |
| <hitle> | Jetpack- Data Binding | <chare> | 1 | <pext> | <starting>Jetpack- Data Binding</heading> Data Binding library is a support library  that provides the feature of binding UI components in an activity/fragment  to the data sources of the application.  The library carries out this binding task in a declarative format  and not in a programmatic way.  For example, in the past to find a TextView widget and bind it to  the userName property of the ViewModel variable,  the findViewById() method was called: TextView textView = findViewById(R.id.sample\_text); textView.setText(viewModel.getUserName()); After using the Data Binding library, we can do it by using the assignment expression - @{}(At curly braces). <TextView  android:text=”@{viewmodel.userName}” /> \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Advantages of Data Binding Component: 1. Make code simpler and easy to maintain by removing UI frameworks  called in the activity. 2. Allows classes and methods to observe changes in data 3. Allows to make objects and fill which works as collection observables. </ending> | </end> |
| <hitle> | Jetpack- Data Binding pros | <chare> | 1 | <pext> | <starting>Jetpack- Data Binding pros</heading> 1. Make code simpler and easy to maintain by removing UI frameworks  called in the activity. 2. Allows classes and methods to observe changes in data 3. Allows to make objects and fill which works as collection observables. </ending> | </end> |
| <hitle> | Jetpack- LiveData | <chare> | 1 | <pext> | <starting>Jetpack- LiveData</heading> LiveData component is an observable data holder class  i.e, 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  i.e., 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. Advantages of LiveData component: 1. UI is updated as per the appropriate change in the data 2. It removes the stopped or destroyed activities  which reduce the chance of app crash 3. No memory leaks as LiveData is a lifecycle-aware component. </ending> | </end> |
| <hitle> | Jetpack- LiveData pros | <chare> | 1 | <pext> | <starting>Jetpack- LiveData pros</heading> 1. UI is updated as per the appropriate change in the data 2. It removes the stopped or destroyed activities  which reduce the chance of app crash 3. No memory leaks as LiveData is a lifecycle-aware component. </ending> | </end> |
| <hitle> | Jetpack- Dagger | <chare> | 1 | <pext> | <starting>Jetpack- Dagger</heading> Dagger is a Dependency Injection framework that will generate a lot of boilerplate code for you to achieve the goal of Dependency injection in Android development. Not that one must have a good understanding of Dependency Injection before answering this question.   There are a number of reasons why you might want to use dagger to inject dependencies instead of injecting them manually. First, dagger can help to reduce the amount of boilerplate code that you need to write in order to inject dependencies. Second, dagger can help to improve the performance of your application by caching injected objects and avoiding repeated object creation. Finally, dagger can help to make your code more modular and easier to test by allowing you to easily swap out different implementations of injected dependencies. </ending> | </end> |
| <hitle> | Jetpack- Architecture Components | <chare> | 1 | <pext> | <starting>Jetpack- Architecture Components</heading> 1. Room 2. WorkManager 3. Lifecycle 4. ViewModel 5. LiveData 6. Navigation 7. Paging 8. Data Binding </ending> | </end> |
| <hitle> | Navigation component Jetpack | <chare> | 1 | <pext> | <starting>Navigation component Jetpack</heading> The Navigation component is a Jetpack library  that helps you manage fragment transactions  and back stack management in your app.  It also provides a way to create a consistent navigation UI across your app.  The benefits of using the Navigation component include a simpler  and more robust way to manage fragment transactions,  as well as a more consistent navigation experience for users. </ending> | </end> |
| <hitle> | Lifecycles component Jetpack | <chare> | 1 | <pext> | <starting>Lifecycles component Jetpack</heading> The Lifecycles component is designed to help manage the lifecycle  of an Android app.  It provides a number of features that can help to automate tasks  and manage memory usage.  For example, you can use the Lifecycles component  to automatically start and stop services based on the lifecycle of your app. </ending> | </end> |
| <hitle> | Lifecycle Jetpack | <chare> | 1 | <pext> | <starting>Lifecycle Jetpack</heading> 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. </ending> | </end> |
| <hitle> | Jetpack- inter-fragment communication | <chare> | 1 | <pext> | <starting>Jetpack- inter-fragment communication</heading> There are two ways of implementing inter-fragment communication: 1. Using an interface: In this approach, you would create an interface  in the parent fragment, and then have the child fragment  implement that interface.  The parent fragment can then call methods on the child fragment  through the interface. 2. Using a ViewModel: With this approach, you would create a ViewModel  that is shared between the parent and child fragments.  The ViewModel can then be used to store data that needs to be shared  between the fragments, and the fragments can observe changes  to the ViewModel. </ending> | </end> |
| <hitle> | Jetpack- Data Binding library | <chare> | 1 | <pext> | <starting>Jetpack- Data Binding library</heading> Data Binding library is used to bind data to UI elements in Android applications.  It eliminates the need for manual data handling in the application code,  making the code simpler and more efficient.  Data Binding library is an important part of Android Jetpack,  as it helps to improve the performance of Android applications. </ending> | </end> |
| <hitle> | Workmanager pros Jetpack | <chare> | 1 | <pext> | <starting>Workmanager pros Jetpack</heading> WorkManager is an Android Jetpack library  that allows you to schedule and manage background tasks in your app.  It is designed to be used with tasks that are not time-critical,  and it provides a number of features that make it a good choice  for managing background work in Android apps.  WorkManager is flexible, allowing you to specify constraints  on when your tasks should run,  and it is also able to intelligently schedule tasks based on the state of the app  and the device.  WorkManager is also able to persist tasks across device reboots,  and it integrates with other Android Jetpack libraries  to provide a complete solution for background work in Android apps. </ending> | </end> |
| <hitle> | Paging Library Jetpack | <chare> | 1 | <pext> | <starting>Paging Library Jetpack</heading> Paging Library makes it easier to load and display  large datasets from a database or network asynchronously  by providing a way to load data in small chunks (called “pages”).  This way, only the data that is needed is loaded at any given time,  which makes it more efficient  and reduces the amount of time and resources  required to load and display large datasets. </ending> | </end> |
| <hitle> | Jetpack Compose | <chare> | 1 | <pext> | Jetpack Compose is a modern UI toolkit that allows developers  to create complex and beautiful user interfaces for Android apps.  It is built on top of the Kotlin programming language.  Jetpack Compose uses a declarative programming model,  which means that developers can describe  what they want their UI to look like, rather than how to create it.  This allows developers to focus on the functionality of their app,  rather than the details of the UI.  --------how work---------------  How does Jetpack Compose work?  Jetpack Compose works by allowing developers to create  UI components using Kotlin code.  This code is then compiled into native Android views,  which can be rendered on the screen.  One of the key features of Jetpack Compose is the ability  to handle state in a simple and intuitive way.  Developers can define state variables in their code,  which can then be used to control the behavior of their UI.  For example, a state variable could be used  to control the visibility of a button,  or to change the text displayed in a label.  ------------why important-----------  Why is Jetpack Compose important?  Jetpack Compose is an important tool for Android app development  for several reasons.  First, it simplifies the process of building complex UIs,  which can save developers time and reduce the likelihood of errors.  Second, it makes it easier for developers to create responsive UIs  that work well on different screen sizes and orientations.  Finally, it allows developers to create UI components  that can be easily reused across different parts of their app,  which can make their codebase more modular and maintainable. | </end> |
| <hitle> | Jetpack Compose pros | <chare> | 2 | <pext> | 1. Declarative:  It is fully declarative so that we can describe UI components  by calling some predefined functions.  2. Compatible:  It is easily compatible with the existing views present in Android.  3. Increase development speed:  Previously developers have to work on the XML file and Kotlin file.  But with the help of jetpack compose this becomes easy  and developers only have to work on the Kotlin files  that’s why it will help developers in increasing development speed.  4. Concise and Idiomatic Kotlin:  Jetpack Compose built the UI with the benefit that Kotlin brings.  5. Easy to maintain:  As the codebase of any application is present in a single file.  It becomes easy to manage and handle the codebase of the application.  6. Written in Kotlin:  The application written using jetpack compose  uses 100 % of Kotlin programming language. | </end> |
| <hitle> | Composition/Recomposition State Management | <chare> | 2 | <pext> | -------------Composition---------------------  Composition in Jetpack Compose refers to the process  of combining smaller UI components to create complex user interfaces.  Instead of relying on XML layout files and modifying views programmatically,  Jetpack Compose embraces a declarative UI approach.  It means that the UI is defined as a function of the current state,  allowing for a more intuitive and concise way of building UIs.  In Jetpack Compose, we can create composable functions  that encapsulate individual UI components.  These functions can be combined and nested to create  a hierarchy of components, forming the complete UI.  Each composable function takes inputs (props)  and produces UI elements based on those inputs.  This modular and reusable nature of composable functions  simplifies UI development, promotes code organization,  and enhances maintainability.  @Composable  fun MyButton() {  Button(  onClick = { /\* Handle button click \*/ },  content = { Text("Click Me") }  )  }  @Composable  fun MyScreen() {  Column {  Text("Welcome to My App")  MyButton()  }  }  In this example, we have two composable functions:  MyButton and MyScreen.  MyButton represents a button component with a click handler,  and MyScreen represents a screen that contains a text  and the MyButton component.  By combining these composable functions in a hierarchical manner,  We can create a complete UI for the screen.  --------------Recomposition------------------------  Recomposition refers to the automatic process of updating  and re-rendering the UI based on changes to the underlying state or inputs.  When the state of your app changes, such as user interactions  or data updates, Compose triggers recomposition.  During recomposition, Compose calculates the new state of the UI  and efficiently updates only the affected components,  minimizing unnecessary rendering.  To enable recomposition, Jetpack Compose follows a unidirectional data flow.  The state of the app is represented by immutable data objects.  Any changes to the state create new instances of those objects,  and Compose compares the new state with the previous state  to identify the minimal set of UI elements that need to be updated.  It then applies the necessary changes to the UI to reflect the new state.  To manage the state in Jetpack Compose,  we can use various state management techniques like mutableStateOf,  remember, and StateFlow.  These constructs provide convenient ways to create and handle the state  within your composable functions.  By separating the state from the UI logic, we can achieve  better separation of concerns and facilitate easy testing and debugging.  @Composable  fun Counter() {  var count by remember { mutableStateOf(0) }  Button(  onClick = { count++ },  content = { Text("Count: $count") }  )  }  In this example, we have a Counter composable function  that maintains a mutable state variable called count  using the mutableStateOf function.  This count represents the current count value.  Whenever the button is clicked, the onClick lambda increments the count variable.  Jetpack Compose automatically triggers recomposition,  and the UI is updated to reflect the new value of count.  ------------Combine Composition/Recomposition---------------  @Composable  fun MyApp() {  var buttonText by remember { mutableStateOf("Click Me") }  var count by remember { mutableStateOf(0) }  Column {  Text("Count: $count")  Button(  onClick = {  count++  buttonText = "Clicked!"  },  content = { Text(buttonText) }  )  }  }  In this example, the MyApp composable function combines both composition and recomposition.  It maintains two state variables: buttonText and count.  When the button is clicked, the onClick lambda increments the count variable and updates the buttonText variable to "Clicked!".  Jetpack Compose triggers recomposition, and the UI is updated accordingly. | </end> |
| <hitle> | Basic Functions of Jetpack Compose | <chare> | 2 | <pext> | ------------------Composble---------------  Composable Function is represented in code  by using @Composable annotation to the function name.  This function will let us define app’s UI programmatically  by describing its shape and data dependencies  rather than focusing on the UI construction process.  ------------------ Preview---------------  The name of the function itself tells us that the function is used  to generate the preview of the composable function.  It is used to display a preview of our composable functions  within Android Studio rather than installing APK in an emulator  or a virtual device. The preview function is annotated by @Preview.  ------------------ setContent---------------  It basically allow to display the actual UI to the user  after the onCreate method has been called.  It similar to setContentView which we have in the XML layout  but instead it takes composable functions.  ------------------ Modifier---------------  It’s a component or parameter in every composable.  Used to modify properties of the composable like height, width, padding, etc.  One note in setting modifiers is, the order matters  when we are applying modifiers.  Modifiers are applied from outer toward inner layer.  ------------------ Column---------------  The column function is used to stack UI elements in a vertical manner.  This function will stack all the children directly one after another  in a vertical manner with no spacing between them.  It is annotated with Column().  ------------------ Row---------------  The row function is used to stack UI elements in a horizontal manner.  This function will stack all the children one after the other  in a horizontal manner with no spacing between them.  It is annotated with Row().  ------------------Box---------------  Box is a widget that is used to positioned elements one over another.  It will position its children relative to its edges.  The stack is used to draw the children which will overlap in the order  that they are specified. It is annotated with Box().  ------------------Text---------------  Text() for making a TextView, Image() for making a ImageView  ------------------ Spacer---------------  It is used to give spacing between two views.  We can specify the height and width of the box.  It is an empty box that is used to give spacing between the views.  It is annotated with Spacer().  ------------------ Vertical Scroll---------------  If the UI components inside the app do not fit the height of the screen  then we have to use the scrolling type of view.  With the help of a vertical scroller, we can provide  a vertically scrolling behavior to our view.  The contents inside the vertical scroller will be clipped  to the bounds of the vertical scroller.  It is annotated with VerticalScroll().  ------------------ Padding---------------  The padding function is used to provide extra white space  according to the requirement around the specific view.  It is simply annotated with Padding().  ------------------ Lazy List, LazyColumn , LazyRow---------------  Lazy List: This composable is equivalent to a recycler view in android’s view system.  It is annotated with LazyColumn().. | </end> |
| <hitle> | Core UI elements of Jetpack Compose | <chare> | 2 | <pext> | ----------------Core UI elements of Compose---------------  1. Surface Composable  2. Wrap content of Composable  3. Align Composables  4. Row Column Box Composables  5. Layout with Row and Columns  6. Extract Composables for reuse  7. Arrangements for Rows and Columns  8. Combine and nest Rows with Columns  -------------Surface Composable------------------  One of the easiest composable which is a bloc of UI  that is displayed on the screen, which can have a color and background color.  ------------------Wrap content of Composable------------------  It’s basically to tell the compose that take only that much space  to fit the content, not more than that.  ------------------Align Composables------------------  This is used to align a composable on the screen  like centre, top end, bottom start, etc. which is relative to its parent.  ------------------Row, Column and Box Composables------------------  A surface can have only one direct child.  So to have multiple children we can use Row, Column or Box.  Column is a vertical container which places children in vertical manner,  Row is a horizontal container which places children in horizontal manner  and Box put the children on top of each other like a stack.  ------------------Arrangements for Rows and Columns------------------  We can align the children of rows or column using arrangements,  like space around.  Also an alignment option is also provided to align  the whole in the entire screen, like put it in the centre of the screen. | </end> |
| <hitle> | MutableState of Jetpack Compose | <chare> | 2 | <pext> | In Jetpack Compose, state management can be accomplished  by using the remember API to store an object in memory,  and the mutableStateOf to declare a state variable.  We can store both mutable and immutable objects.  The mutableStateOf creates an observable MutableState<T>,  which is an observable type.  Any changes to value schedules a recomposition (re-rendering)  of any composable functions that read it.  There are three ways to declare a MutableState object:  val mutableState = remember { mutableStateOf(0) }  var value by remember { mutableStateOf(false) }  val (value, setValue) = remember { mutableStateOf("Hello, Compose!") } | </end> |
| <hitle> | LiveData of Jetpack Compose | <chare> | 2 | <pext> | LiveData is a data holder class that can be observed within a given lifecycle,  meaning it respects the lifecycle of other app components,  such as activities, fragments, or services.  This ensures LiveData only updates observers  that are in an active lifecycle state,  which also ensures no memory leaks happen within your app.  1. We need to create an instance of the LiveData class  to hold a certain type of data, which is usually done  within the ViewModel class.  We can use MutableLiveData if we’d like to update the value at some point.  class HomeViewModel : ViewModel() {  // Create a MutableLiveData instance that keeps a string  val userName = MutableLiveData<String>()  }  2. Obtain the value in our composable  by calling the observeAsState method.  @Composable  fun HomeScreen (viewModel: HomeViewModel = viewModel()) {  // Create an observer of the state of userName  val userName = viewModel.userName.observeAsState()  // Use the value in your UI  Column {  Text(userName)  }  }  3. To update the value, we need to create a function  that sets the new value to its value property.  fun updateUserName(newName: String) {  userName.value = newName  }  4. We’d use the new function in the Compose file  as viewModel.updateUserName("..."). | </end> |
| <hitle> | StateFlow of Jetpack Compose | <chare> | 2 | <pext> | StateFlow is a newer alternative to LiveData.  Both have similarities, and both are observable.  1. We need to create an instance of StateFlow to hold  a certain type of data (we can use MutableStateFlow  if we’d like to update the value at some point.)  class HomeViewModel : ViewModel() {  // Create a MutableStateFlow instance that keeps a string  val userName = MutableStateFlow<String>()  }  2. We need to obtain the value in the composable  by calling the collectAsState method:  @Composable  fun HomeScreen (viewModel: HomeViewModel = viewModel()) {  // Create an observer to collect the state of userName  val userName = viewModel.userName.collectAsState()  // Use the value in your UI  Column {  Text(userName)  }  }  3. To update a value, we need to create a function  that sets the new value to its value property.  fun updateUserName(newName: String) {  userName.value = newName  }  4. You’d use the new function in the Compose file  like viewModel.updateUserName("..."). | </end> |
| <hitle> | State management in Jetpack Compose | <chare> | 2 | <pext> | ----------State-------------  State in an application is any value that can change over time.  It depends on the data flow.  Since data flow is unidirectional then,  when an event is triggered the state is updated  and then the state is displayed.  Advantages are testability, state encapsulation and UI consistency.  --------------remember----------------  It allows the state to be remembered over the recomposition.  --------------ViewModel---------------  It’s a class that is responsible for preparing  and managing the state of the UI for Fragments or Activities.  It has a relationship with the view to prepare the data  that has to be displayed.  View points to the ViewModel, and expects data from it.  ViewModel bounds on the UI State which is basically responsible  for preparing and managing, and what the UI will display.  -----------LiveData-------------  It’s basically a lifecycle-aware observable data holder.  Which means it holds some data but is a more reactive in nature.  View observe the changes on live data and it aspects to be notified  when the data changes.  LiveData doesn’t know which views are observing it,  but all the views get notified when there is any change to the data. | </end> |
| <hitle> | Navigation in Jetpack Compose | <chare> | 2 | <pext> | ----------How navigation work in Compose?--------------  Jetpack Compose works on the logic of Activity and Fragment,  a view navigating between fragments, adding to the fragment manager.  Much easier to navigate.  So instead of fragment Jetpack Compose uses Composable Screen,  so we can switch between composables.  ----------What is a Navigation Graph in Compose?------------  It’s a graph which maps out all the composable screens in a graph,  so that you have a very good understanding on what destinations  we can have in different points of the actual flow of an application.  ----------What is a Navigation Stack in Compose?------------  It’s a container similar to the stack that we had in our program manager  in which we holds composables.  And they are those composables that we have defined  in the navigation graph that are represented as screens. | </end> |