1. Flavouring in android
2. How to update column size in room android
3. Difference between between workmanager and coroutines
4. Coroutines flow
5. Types of Coroutines scopes

Coroutines in Kotlin have different types of scopes that determine their lifecycle and behavior. Here are some common coroutine scopes:

GlobalScope: A coroutine scope that is not bound to any specific lifecycle and has the longest lifecycle. Coroutines launched in the global scope are not automatically cancelled, and their lifecycle is managed manually.

CoroutineScope: An interface representing a coroutine scope that can be used to launch coroutines. It typically corresponds to a specific context or component in your application, such as an activity, fragment, or ViewModel.

MainScope: A specific coroutine scope tied to the main thread's lifecycle in Android applications. It's typically used for launching coroutines that interact with the UI, ensuring that they are automatically cancelled when the associated UI component is destroyed.

ViewModelScope: A coroutine scope tied to the ViewModel lifecycle in Android Jetpack's ViewModel component. Coroutines launched in this scope are automatically cancelled when the associated ViewModel is cleared or destroyed.

LifecycleScope: Introduced in Android Jetpack's Lifecycle library, this coroutine scope is tied to the lifecycle of a specific lifecycle owner, such as an activity or fragment. Coroutines launched in this scope are automatically cancelled when the associated lifecycle is destroyed, helping prevent memory leaks and resource leaks.

These coroutine scopes provide different levels of lifecycle management and are used to launch coroutines in various contexts within your application. Choosing the appropriate coroutine scope ensures that your coroutines are properly managed and cancelled when they are no longer needed, improving the overall performance and reliability of your app.

1. Data binding
2. View binding vs data binding

View Binding: Simplifies view access by generating a binding class for each XML layout. Lightweight, efficient, but doesn't support data binding.

Data Binding: Facilitates automatic data synchronization between UI and data sources using declarative XML layouts. Supports two-way binding, custom binding adapters, and is suitable for complex data interactions.

1. Which is the default launch in coroutine

In Kotlin Coroutines, the default coroutine dispatcher used for launching coroutines without specifying any dispatcher explicitly is the **dispatchers.main**. This dispatcher is typically associated with the main UI thread in Android applications and is used for performing UI-related tasks.

It's important to note that this default behavior can vary depending on the context in which coroutines are launched. For example, in a non-Android environment, such as a server-side application, the default dispatcher might be different. However, in Android applications, where UI-related operations are common, the main dispatcher is typically the default.

1. How to pass the data back from intent
2. Whats the usecase of domain layer in android

The domain layer in MVVM architecture is crucial for separating business logic from the UI and data layers. It encapsulates business rules, making the application more modular, maintainable, and testable. The domain layer typically includes use cases, which represent specific actions or business processes, and interfaces with repositories to handle data operations. This separation ensures clean architecture principles, improves reusability, and enhances testability by isolating the business logic from the Android framework.

1. Difference between traditional navigation vs jetpack navigation

**\*\*Traditional Navigation:\*\***

- \*\*Manual Management:\*\* Developers manually manage fragment transactions and back stack operations using `FragmentManager` and `FragmentTransaction`.

- \*\*Code Complexity:\*\* Navigation logic is scattered throughout the codebase, often leading to complex and hard-to-maintain code.

- \*\*Lack of Consistency:\*\* Handling of deep links and up/back actions can be inconsistent and error-prone.

**\*\*Jetpack Navigation:\*\***

- \*\*Simplified Navigation:\*\* Provides a consistent and declarative approach using a navigation graph (XML) to define all navigation paths.

- \*\*Single Source of Truth:\*\* Centralizes navigation logic, making it easier to visualize and manage within the navigation graph.

- \*\*Enhanced Features:\*\* Supports advanced features like deep linking, safe args (type-safe argument passing), and navigation animations out-of-the-box.

- \*\*Lifecycle Awareness:\*\* Integrates seamlessly with the lifecycle of activities and fragments, reducing boilerplate and potential bugs.

1. Latinit vs lazy