ANDROID KOTLIN - INTERVIEW SECRETS

lateinit vs lazy?

Coroutines Interview Questions For Android Kotlin

What are Coroutines?

A coroutine can be thought of as a worker that performs some long-running/memory-intensive operations asynchronously. The asynchronous nature of a coroutine ensures that any long-running/memory-intensive operations do not block the main thread of execution. In essence, it takes a block of code and runs it on a particular thread.

How are they different from Threads?

Coroutines may be thought of as lightweight threads. They are called so because multiple coroutines can be scheduled to be executed on the same thread. So, the creation of 100,000 threads results in the creation of 100,000 threads. Whereas, the creation of 100,000 coroutines doesn't necessarily mean that 100,000 threads get created. Since the resources of a single thread are shared by multiple coroutines, they are much lighter when compared to creating raw threads.

What problem do Coroutines solve?

Coroutines solve many problems related to concurrency. They simplify the process of writing asynchronous code.

Elimination of chaining callbacks a.k.a "Callback Hell" : "Callback hell" refers to the infamous practice of nesting callback functions. There are a lot of scenarios wherein callback functions need to be nested. This results in the nesting of callback functions making the code extremely difficult to read and debug. The sequential nature of coroutines precludes the need for using nested callbacks, making the code much more readable and maintainable.

Exception handling and cancellation
Exception handling and cancellation can be difficult to manage in a concurrent environment. They make it very easy to handle exceptions. They not only provide methods of propagating and handling exceptions but also allow us to define the cancellation behavior. This feature allows us to write safe concurrent code.

Sequential Execution - Makes concurrent code easy to read and debug
Each operation within a coroutine executes sequentially. For example, if there is a network call to fetch the details of a product followed by another network call to fetch the image of the product, then, the second request doesn't execute unless the first request was successfully executed. This has several benefits. First and foremost, it allows us to skip unnecessary network calls. In this example, if a fetch operation to fetch the details of an invalid product is made, then, the fetch operation to get the image of the product can be completely skipped. Secondly, it also makes debugging very easy, since asynchronous code is represented synchronously. Now, this doesn't mean that it is not possible to execute more than one network call at the same time. The async{}

block provided by the coroutines library can be used to achieve concurrent execution.

Scoped Execution - Prevent inefficient use of resources
All coroutines must be started in a coroutine scope. If the scope gets canceled, then all coroutines that were started within that scope get canceled. This helps to prevent unnecessary use of resources. Especially in Android, where lifecycles are involved, canceling un-necessary background tasks is of paramount importance. Moreover, in the context of Android development, many jetpack libraries such as room and datastore, provide built-in coroutine scope. This is a big advantage because the users of these libraries do not need to think of when to start and stop the execution of coroutines. We just define what we have to execute and the library will take care of stopping/canceling/restarting the coroutines. This helps to utilize the resources in a very efficient manner.

What are dispatchers? Explain their types

A dispatcher allows us to specify which pool of threads the coroutines are executed. The dispatcher can be specified as a part of the coroutine context. There are 5 types of dispatchers that are available for use:

Default - The default dispatcher is used to schedule coroutines that perform CPU-intensive operations such as filtering a large list.

IO - This is the most common dispatcher. It is used to run coroutines that perform I/O operations such as making network requests and fetching data from the local database.

Main - The main dispatcher is used to execute coroutines on the main thread. It generally doesn't block the main thread, but, if several coroutines containing long-running operations get executed in the context of this dispatcher, then the main thread has a possibility of getting blocked. This is mainly used in conjunction with the withContext(){} method to switch the context of execution to the main thread. This comes in handy if it is required to perform some operation on a background thread and switching the context of execution to the main thread to update the UI. Since touching the UI from a background thread is not permitted in Android, this allows us to switch the context of execution to the main thread before updating the UI.

Unconfined - This dispatcher is very rarely used. Coroutines scheduled to run on this dispatcher run on the thread that they are started/resumed. When they are first called, they get executed in the thread that they are called in. When they resume, they get resumed in the thread that they are resumed in. In summary, they are not confined to any thread pool.

Immediate - The immediate dispatcher is a recently introduced dispatcher. It is used to reduce the cost of re-dispatching the coroutine to the main thread. There is a slight cost associated when switching the dispatcher using the withContext{} block. Using the Immediate dispatcher ensures the following.

If a coroutine is already executing in the main dispatcher, then the coroutine wouldn't be re-dispatched, therefore removing the cost associated with switching dispatchers.

If there is a queue of coroutines waiting to get executed in the main dispatcher, the immediate dispatcher ensures that the coroutine will be executed as immediately as possible.

What is the importance of a coroutine scope?

A coroutine scope manages the lifecycle of coroutines that are launched inside the scope. It determines when the coroutines inside the scope get started, stopped, and restarted. Coroutine scopes are especially helpful for the following reasons.

Allows the grouping of coroutines. If the scope gets canceled, then all coroutines that were started within that scope get canceled. This helps to prevent unnecessary use of resources when the coroutines are no longer needed.

Coroutine scope helps to define the context in which the coroutines are executed.

https://medium.com/@theAndroidDeveloper/5-common-kotlin-coroutines-interview-questions-f084d098f51d

What are coroutines?

Coroutines are a type of light-weight thread that can be used to improve the performance of concurrent code. Coroutines can be used to suspend and resume execution of code blocks, which can help to avoid the overhead of creating and destroying threads.

Can you explain the difference between a thread and a coroutine?

A thread is a unit of execution that can run independently from other threads. A coroutine is a light-weight thread that can be suspended and resumed.

How do threads compare with coroutines in terms of performance?

Threads are typically heavier than coroutines, so they can be more expensive in terms of performance. However, this is not always the case, and it really depends on the specific implementation. In general, coroutines tend to be more efficient when it comes to CPU usage, but threads may be better when it comes to I/O bound tasks.

Does Kotlin have any built-in support for concurrency? If yes, then what is it?

Yes, Kotlin has built-in support for concurrency via coroutines. Coroutines are light-weight threads that can be used to improve the performance of concurrent code.

Why are coroutines important?

Coroutines are important because they allow you to write asynchronous code that is more readable and easier to reason about than traditional asynchronous code. Coroutines also have the ability to suspend and resume execution, which can make your code more efficient.

What makes coroutines more efficient than threads?

Coroutines are more efficient than threads because they are lightweight and can be suspended and resumed without incurring the overhead of a context switch. This means that they can be used to perform tasks that would otherwise block a thread, without incurring the same performance penalty.

Can you explain what suspend functions are?

Suspend functions are functions that can be paused and resumed at a later time. This is useful for long-running tasks that might need to be interrupted, such as network requests. By using suspend functions, you can ensure that your code is more responsive and can avoid potential errors.

Are suspend functions executed by default on the main thread or some other one?

By default, suspend functions are executed on a background thread.

Is it possible to use coroutines outside Android development? If yes, then how?

Yes, it is possible to use coroutines outside of Android development. One way to do this is by using the kotlinx-coroutines-core library. This library provides support for coroutines on multiple platforms, including the JVM, JavaScript, and Native.

What's the difference between asynchronous code and concurrent code?

Asynchronous code is code that can run in the background without blocking the main thread. Concurrent code is code that can run in parallel with other code.

What happens if we call a suspend function from another suspend function?

When we call a suspend function from another suspend function, the first function will suspend execution until the second function completes. This can be used to our advantage to create asynchronous code that is easy to read and debug.

Can you give me an example of when you would want to run two tasks concurrently instead of sequentially?

There are a few reasons you might want to run two tasks concurrently instead of sequentially. One reason is if the tasks are independent of each other and can be run in parallel. Another reason is if one task is dependent on the other task and you want to avoid blocking the main thread. Finally, if you have a limited number of resources available, you might want to run tasks concurrently in order to make better use of those resources.

What is the difference between launch() and async()? Which should be used in certain situations?

The main difference between launch() and async() is that launch() will create a new coroutine and start it immediately, while async() will create a new coroutine but will not start it until something calls await() on the resulting Deferred object. In general, launch() should be used when you want a coroutine to run in the background without blocking the main thread, while async() should be used when you need to wait for the result of a coroutine before continuing.

What is the best way to cancel a running job in Kotlin?

The best way to cancel a running job in Kotlin is to use the cancel() function. This function will cancel the job and any associated children jobs.

What do you understand about Job objects? How are they different from CoroutineScope?

Job objects are the basic building blocks of coroutines. They define a coroutine's lifecycle and provide a way to cancel it. CoroutineScope is used to define a scope for a coroutine, which determines its lifetime and other properties.

What happens if there's an exception thrown inside a coroutine?

If there's an exception thrown inside a coroutine, then the coroutine will be cancelled. All the coroutine's children will also be cancelled, and any pending work in those coroutines will be lost.

What does the {Dispatchers.Main} expression mean?

The {Dispatchers.Main} expression is used to specify that a particular coroutine should run on the main thread. This is important because some operations can only be performed on the main thread, and so specifying that a coroutine should run on the main thread ensures that it will be able to perform those operations.

What are some common mistakes made when working with coroutines in Kotlin?

One common mistake is not using the right context when launching a coroutine. This can lead to your coroutine being cancelled when it shouldn't be, or not being able to access the data it needs. Another mistake is not using a structured concurrency approach, which can lead to race conditions and other issues. Finally, not using the right tools for debugging can make it difficult to find and fix problems with your coroutines.

What are some good practices to follow when using Kotlin coroutines?

Some good practices to follow when using Kotlin coroutines include:

Use coroutines for short-lived background tasks
Use coroutines for tasks that can be executed in parallel
Use coroutines for tasks that need to be executed on a different thread than the UI thread
Do not use coroutines for tasks that need to be executed on the UI thread
Do not use coroutines for tasks that need to be executed synchronously

What is your opinion on the future of coroutines in Kotlin?

I believe that coroutines will continue to be a popular feature of Kotlin, as they offer a convenient and efficient way to manage concurrency. Additionally, the Kotlin team has been very supportive of coroutines and is constantly working to improve the experience of using them.

https://climbtheladder.com/kotlin-coroutines-interview-questions/

What's the best way to manage the coroutines lifecycle in Android ViewModel?

However, you should treat them as guidelines and adapt them to your requirements as needed.
Inject Dispatchers.
Suspend functions should be safe to call from the main thread.
The ViewModel should create coroutines.
Don't expose mutable types.
The data and business layer should expose suspend functions and Flows.

How do I use coroutines in ViewModel?

Add a CoroutineScope to your ViewModel by creating a new scope with a SupervisorJob that you cancel in the onCleared() method. The coroutines created with that scope will live as long as the ViewModel is being used.

What is the function used for launching for coroutines?

Launching a Coroutine as a Job

The launch {} function returns a Job object, on which we can block until all the instructions within the coroutine are complete, or else they throw an exception. The actual execution of a coroutine can also be postponed until we need it with a start argument. If we use CoroutineStart.

What is the difference between MainScope and GlobalScope?

MainScope is a CoroutineScope that uses the Dispatchers. Main dispatcher by default, which is bound to the main UI thread. The GlobalScope is a CoroutineScope that has no dispatcher in its coroutine context.

What is the difference between GlobalScope and CoroutineScope?

Since it's alive along with application lifetime, GlobalScope is a singleton object. It's actually same as CoroutineScope but the syntax doesn't have CoroutineContext on its parameter. By default it will have default CoroutineContext which is Dispatchers.

What is the difference between GlobalScope async and launch?

The difference between async and launch is that async returns a value and launch doesn't. This is referring to the suspend lambda passed in, not the launch or async method itself. launch returns Job . async returns Deferred (which extends Job) which also has methods for getting the suspend lambda's result.

What is the difference between coroutineScope launch and runBlocking?

The main difference is that the runBlocking method blocks the current thread for waiting, while coroutineScope just suspends, releasing the underlying thread for other usages. Because of that difference, runBlocking is a regular function and coroutineScope is a suspending function.

What is the difference between launch and async coroutines?

launch: fire and forget. async: perform a task and return a result.

Launch vs Async in Kotlin Coroutines.

launch{} returns a Job and does not carry any resulting value.

async{} returns an instance of Deferred<T> , which has an await() function that returns the result of the coroutine.

https://developer.android.com/courses/quizzes/android-coroutines/use-coroutines-common-android-cases