Kotlin Interview Questions And Answers

lateinit vs lazy?

lateinit can be used for var properties where Kotlin promises the compiler that the variable will be initialized later failure of which will lead to exception. lazy can only be used for val properties. It will be initialized during the first call where the value will be stored in a cache so that another call to the same variable will serve the value stored in cache.

What are the types of equality in Kotlin?

A) There are two types of equality in Kotlin -
1) Referential Equality (===): It tells whether the two references are pointing to same address or not. In Kotlin it is represented with '===', unlike in Java where it is represented with '=='.

2) Structural Equality (==): Structural equality tells whether the data inside objects is equal or not. Java is Structural equality in Kotlin is represented by '==' where as in Java it is done by .equals() method.

In Kotlin also we can use .equals method but it is recommended to use == because Kotlin internally converts a==b, for example, to the following code:

```
a?.equals(b) ?: (b === null)
```

What is the use of vararg keyword in Kotlin?

varargs are used to pass unlimited variables to the constructor.

```
fun sum(vararg values : Int) =  values.sum()
assertEquals(5, sum(2,3)) // true
```

What are Destructuring Declarations in Kotlin?

Destructuring declarations allows us to destructure an object to various variables.

Let us take a data class for example. We know that whenever we pass args to data class, Kotlin creates a component for each arg, named - component1(), component2() etc.

Destructured declarations simply points to those components in the same order respectively.

Here's an example:

```
data class Person(val name: String, val age: Int)

// destructuring declarations
val (username, userAge) = Person("vamsi", "21")
println(username) // vamsi
```

Here these username and userAge will directly point to the component functions of data class internally as follows:

```
val username = Person.component1()
val userAge = Person.component2()
```

What are Data Classes?

Data classes are specifically designed to hold the data. In data classes, the Designers of Kotlin has overrided methods - equals(), hashcode() and toString() internally to feciliate the data holding capabilities.

The main uses of data classes -

1. They can be easily copied structurally using copy() function,
2. They can be used for destructuring declarations
3. They also can inherit classes and interfaces

Limitations of Data Classes:

1. They cannot be open
2. They cannot inherit another data class
3. varargs cannot be used as arguments in data class as the data class internally needs to generate toString() and hashcode() method's logic.

If we create two data class objects with same data then are they equal?
A) Those two data are objects are equal structurally but not referentially.

Here is an example:

```
data class Person(name:String)
//creating objects for data class
val a = Person("Vamsi")
val b = Person("Vamsi")
println(a==b) // true; a normal class (without .equals() overridden) would return false in this case
println(a===b) // false
```

What are inline functions? When to use them?

Functions that take lambda parameter as arguments generates objects inside calling function's code. If these functions are called at multiple places, multiple objects are created which affects the performance of our Android App. To avoid these memory allocations created by lambda expressions (anonymous objects are created), we make the functions inline by adding a keyword - 'inline' to our function.

```
inline fun SharedPreferences.edit(commit: Boolean = false, action: SharedPreferences.Editor.() -> Unit) {
    val editor = edit()
    action(editor)
    if(commit)
        editor.commit()
    else
        editor.apply()

}
```

Inline functions are generally used when we need to pass small functions as parameters. It is generally not advisable to pass large functions to inline functions.

What are noinline keyword? Where we need to use them in realtime scenario?

We cannot pass a lambda function, which comes as argument inside inline function, to another function that accepts lambda. We will get an error stating 'Illegal usage of inline-parameter'. In this case we need to pass that lambda function with noinline keyword which makes the compiler instead of writing the code to the called location, creates the function for that specific function.

```
inline fun SharedPreferences.edit(
commit: Boolean = false,
noinline anotherFunction: Int.() -> Unit = {},
action: SharedPreferences.Editor.() -> Unit)
{
  // passing noinline function to another function
  myFun(anotherFunction)

  val editor = edit()
  action(editor)
  if(commit)
    editor.commit()
  else
    editor.apply()

}

fun myFun(importantAction: Int.() -> Unit) {
  importantAction(-1)
}
```

We'll use this only in case if multiple lambdas are passed to function arguments. If there is only one lambda which need to be referenced in another function, we better not use inline function at all.

What is the use of crossinline in Kotlin?

When we don't want to return inside lambda function (non-local returns) that is passed as inline argument, we use crossinline keyword on that lambda argument.

Here's an example:

```
fun createPerson() {
    val person = Person()
    person.name = "Vamsi"
    performFunction {
        println("Created a person with name: ${person.name}")
//      return  // Not allowed here as its crossinline
    }
}

inline fun performFunction (crossinline x : () -> Unit) {
    x()
}
```

What is the use of infix in Kotlin?

infix functions are used for declaring a short form notation of a function.

Here's an example:

```
infix fun Int.printSmallest(x: Int) {
   print(if(this < x) this else x)
}

1 printSmallest 5 // calling the function directly
```

What are Sealed classes in Kotlin?

Sealed classes are similar to enum classes which also has restrictive set of types allowed, except that Sealed classes can contains additional data to be propagated(which we cannot achieve with enum classes).

```
sealed class Result<out T: Any> {
    data class Success<out T: Any>(val data: T): Result<T>()
    data class Error(val exception: Exception): Result<Nothing>()
}
```

What are Coroutines?

Coroutines

A coroutine is a concurrency design pattern that you can use on Android to simplify code that executes asynchronously. Coroutines were added to Kotlin in version 1.3 and are based on established concepts from other languages.

On Android, coroutines help to manage long-running tasks that might otherwise block the main thread and cause your app to become unresponsive.

Coroutines offers many benefits over other asynchronous solutions, including the following:

- **Lightweight**. You can run many coroutines on a single thread due to support for suspension, which doesn't block the thread where the coroutine is running. Suspending saves memory over blocking while supporting many concurrent operations.
- **Fewer memory leaks**. Use structured concurrency to run operations within a scope.
- **Built-in cancellation support**. Cancellation is propagated automatically through the running coroutine hierarchy.
- **Jetpack integration**. Many Jetpack libraries include extensions that provide full coroutines support. Some libraries also provide their own coroutine scope that you can use for structured concurrency.

Example:

```
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch { // launch a new coroutine in background and continue
        delay(1000L) // non-blocking delay for 1 second (default time unit is ms)
        println("World!") // print after delay
    }
    println("Hello,") // main thread continues while coroutine is delayed
    Thread.sleep(2000L) // block main thread for 2 seconds to keep JVM alive
}

Output:
Hello,
World!
```

Essentially, coroutines are light-weight threads. They are launched with `launch` *coroutine builder* in a context of some `CoroutineScope`. Here we are launching a new coroutine in the `GlobalScope`, meaning that the lifetime of the new coroutine is limited only by the lifetime of the whole application.

The global idea behind coroutines is that a function can suspend its execution at some point and resume later on. One of the benefits however of coroutines is that when it comes to the developer, writing non-blocking code is essentially the same as writing blocking code. The programming model in itself doesn't really change.

What is the difference between suspending and blocking?

Take for instance the following code:

```
fun postItem(item: Item) {
    launch {
        val token = preparePost()
        val post = submitPost(token, item)
        processPost(post)
    }
}

suspend fun preparePost(): Token {
    // makes a request and suspends the coroutine
    return suspendCoroutine { /* ... */ }
}
```

Difference is explained assuming there are 2 functions Function A and Function B

Blocking

Function A has to be completed before Function B continues. The thread is locked for Function A to complete its execution.

Suspending

Function A while it has started could be suspended and let Function B execute, Function A can be resumed later. The thread is not locked by Function A. Function A can be started, paused and resumed without interrupting the execution of Function B

This code will launch a long-running operation without blocking the main thread. The `preparePost` is what's called a `suspendable function`, thus the keyword `suspend` prefixing it. What this means as stated above, is that the function will execute, pause execution and resume at some point in time.

- The function signature remains exactly the same. The only difference is `suspend` being added to it. The return type however is the type we want to be returned.
- The code is still written as if we were writing synchronous code, top-down, without the need of any special syntax, beyond the use of a function called `launch` which essentially kicks-off the coroutine
- The programming model and APIs remain the same. We can continue to use loops, exception handling, etc. and there's no need to learn a complete set of new APIs
- It is platform independent. Whether we targeting JVM, JavaScript or any other platform, the code we write is the same. Under the covers the compiler takes care of adapting it to each platform.

Example

Blocking

```
fun main(args: Array<String>) {
    println("Main execution started")
    threadFunction(1, 200)
    threadFunction(2, 500)
    Thread.sleep(1000)
    println("Main execution stopped")
}

fun threadFunction(counter: Int, delay: Long) {
    thread {
  println("Function ${counter} has started on ${Thread.currentThread().name}")
        Thread.sleep(200)
        println("Function ${counter} is finished on ${Thread.currentThread().name}")
    }
}
```

Output

```
Main execution started
Function 1 has started on Thread-0
Function 2 has started on Thread-1
Function 1 is finished on Thread-0
Function 2 is finished on Thread-1
Main execution stopped
```

In the above program the control is as follows

- The `main` method will invoke `threadFunction 1` and `threadFunction 2` in the main thread. Each method will create a separate thread , prints `Function has started` along with the thread name(Thread - 0/ Thread- 1), wait for a given time then prints `Function is finshed` along with the thread name.

- Finally the `main` method will wait for 1000 ms in order to wait the threads to finish and then stop executing.

- Here important point to notice is that The main thread is doing nothing just waiting until all other threads complete their execution which occupies the CPU with unnecessary waiting. It would be much better if it utilises the same time executing any other tasks. In android app this program will freeze the UI as it has only one main thread as the main thread is waiting for other threads to complete its execution.

Suspend

```
    fun main(args: Array<String>) = runBlocking {
      println("Main execution started")
        joinAll(
            async { suspendFunction(1, 200) },
            async { suspendFunction(2, 500) }  ,
            async {
   println(" Other task is working on ${Thread.currentThread().name}")
    delay(100)
}
       )
        println("Main execution stopped")
    }

    suspend fun suspendFunction(counter: Int, delay: Long) {
        println("Coroutine ${counter} has started work on ${Thread.currentThread().name}")
        delay(500)
        println("Function ${counter} is finished on ${Thread.currentThread().name}")
    }
```

Output

```
Main execution started
Coroutine 1 has started work on main
Coroutine 2 has started work on main
Other task is working on main
Coroutine 1 is finished on main
Coroutine 2 is finished on main
Main execution stopped
```

In the above program the control is as follows

- The `main` method will invoke `suspendFunction 1` and `suspendFunction 2` in the main thread.
- Each `suspend Function` will be executed in the main thread , prints `Coroutine has started work` along with thread name which main thread and when `delay` is called the coroutine is suspended as call to `delay` is a suspension point , once the waiting period is finished the coroutine wakesup and prints `Coroutine is finished.` While the coroutines are suspended the other coroutine wil starts and prints `Other task is working on main.`
- Here important point to notice is that **While the coroutine is executing the main method is not blocked so that it can execute other tasks.**

What do you know about Channels ?

Channels

Deferred values provide a convenient way to transfer a single value between coroutines. Channels provide a way to transfer a stream of values. Channel is a non-blocking primitive for communication between a sender (via `SendChannel`) and a receiver (via `ReceiveChannel`).

Channel basics

A Channel is conceptually very similar to `BlockingQueue`. One key difference is that instead of a blocking `put` operation it has a suspending send, and instead of a blocking `take` operation it has a suspending receive.

```
val channel = Channel<Int>()
launch {
    // this might be heavy CPU-consuming computation or async logic, we'll just send five squares
    for (x in 1..5) channel.send(x * x)
}
// here we print five received integers:
repeat(5) { println(channel.receive()) }
println("Done!")
```

The output of this code is:

```
1
4
9
16
25
Done!
```

Closing and iteration over channels

Unlike a queue, a channel can be closed to indicate that no more elements are coming. On the receiver side it is convenient to use a regular `for` loop to receive elements from the channel.

Conceptually, a close is like sending a special close token to the channel. The iteration stops as soon as this close token is received, so there is a guarantee that all previously sent elements before the close are received:

```
val channel = Channel<Int>()
launch {
    for (x in 1..5) channel.send(x * x)
    channel.close() // we're done sending
}
// here we print received values using `for` loop (until the channel is closed)
for (y in channel) println(y)
println("Done!")
```

Building channel producers

The pattern where a coroutine is producing a sequence of elements is quite common. This is a part of producer-consumer pattern that is often found in concurrent code. You could abstract such a producer into a function that takes channel as its parameter, but this goes contrary to common sense that results must be returned from functions.

There is a convenient coroutine builder named produce that makes it easy to do it right on producer side, and an extension function consumeEach, that replaces a `for` loop on the consumer side:

```
val squares = produceSquares()
squares.consumeEach { println(it) }
println("Done!")
```

Pipelines

A pipeline is a pattern where one coroutine is producing, possibly infinite, stream of values:

```
fun CoroutineScope.produceNumbers() = produce<Int> {
    var x = 1
    while (true) send(x++) // infinite stream of integers starting from 1
}
```

And another coroutine or coroutines are consuming that stream, doing some processing, and producing some other results. In the example below, the numbers are just squared:

```
fun CoroutineScope.square(numbers: ReceiveChannel<Int>): ReceiveChannel<Int> = produce {
    for (x in numbers) send(x * x)
}
```

The main code starts and connects the whole pipeline:

```
val numbers = produceNumbers() // produces integers from 1 and on
val squares = square(numbers) // squares integers
repeat(5) {
    println(squares.receive()) // print first five
}
println("Done!") // we are done
coroutineContext.cancelChildren() // cancel children coroutines
```

Buffered channels

The channels shown so far had no buffer. Unbuffered channels transfer elements when sender and receiver meet each other (aka rendezvous). If send is invoked first, then it is suspended until receive is invoked, if receive is invoked first, it is suspended until send is invoked.

Both Channel() factory function and produce builder take an optional `capacity` parameter to specify buffer size. Buffer allows senders to send multiple elements before suspending, similar to the `BlockingQueue` with a specified capacity, which blocks when buffer is full.

Take a look at the behavior of the following code:

```
val channel = Channel<Int>(4) // create buffered channel
val sender = launch { // launch sender coroutine
    repeat(10) {
        println("Sending $it") // print before sending each element
        channel.send(it) // will suspend when buffer is full
    }
}
// don't receive anything... just wait....
delay(1000)
sender.cancel() // cancel sender coroutine
```

It prints "sending" five times using a buffered channel with capacity of four:

```
Sending 0
Sending 1
Sending 2
Sending 3
Sending 4
```

The first four elements are added to the buffer and the sender suspends when trying to send the fifth one.

What do you know about Flow?

Flow

In coroutines, a *flow* is a type that can emit multiple values sequentially, as opposed to *suspend functions* that return only a single value. For example, you can use a flow to receive live updates from a database.

Flows are built on top of coroutines and can provide multiple values. A flow is conceptually a stream of data that can be computed asynchronously. To represent the stream of values that are being asynchronously computed, we can use a `Flow<Int>` type:

```
fun simple(): Flow<Int> = flow { // flow builder
    for (i in 1..3) {
        delay(100) // pretend we are doing something useful here
        emit(i) // emit next value
    }
}

fun main() = runBlocking<Unit> {
    // Launch a concurrent coroutine to check if the main thread is blocked
    launch {
        for (k in 1..3) {
            println("I'm not blocked $k")
            delay(100)
        }
    }
    // Collect the flow
    simple().collect { value -> println(value) }
}
```

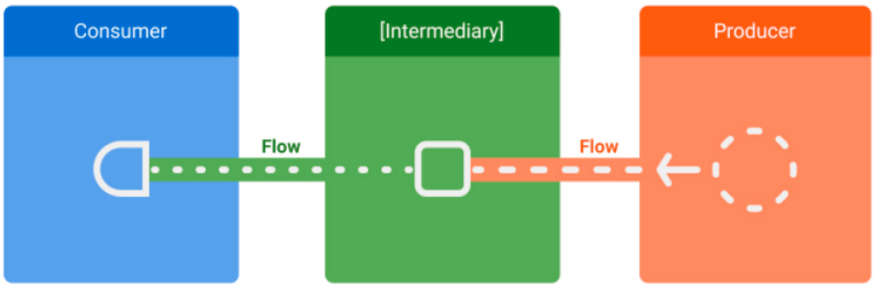
This code waits 100ms before printing each number without blocking the main thread. This is verified by printing "I'm not blocked" every 100ms from a separate coroutine that is running in the main thread:

```
I'm not blocked 1
1
I'm not blocked 2
2
I'm not blocked 3
3
```

A flow is very similar to an `Iterator` that produces a sequence of values, but it uses suspend functions to produce and consume values asynchronously. This means, for example, that the flow can safely make a network request to produce the next value without blocking the main thread.

There are three entities involved in streams of data:

- A producer produces data that is added to the stream. Thanks to coroutines, flows can also produce data asynchronously;
- (Optional) Intermediaries can modify each value emitted into the stream or the stream itself;
- A consumer consumes the values from the stream.



In Android, *a data source or repository* is typically a producer of UI data that has the `View` as the consumer that ultimately displays the data. Other times, the `View` layer is a producer of user input events and other layers of the hierarchy consume them. Layers in between the producer and consumer usually act as intermediaries that modify the stream of data to adjust it to the requirements of the following layer.

Flow builders

There are the following basic ways to create a flow:

`flowOf(...)` functions to create a flow from a fixed set of values;
`asFlow()` extension functions on various types to convert them into flows;
`flow { ... }` builder function to construct arbitrary flows from sequential calls to `emit` function;
`channelFlow { ... }` builder function to construct arbitrary flows from potentially concurrent calls to the `send` function.
`MutableStateFlow` and `MutableSharedFlow()` define the corresponding constructor functions to create a hot flow that can be directly updated.

Flows are cold

Flows are *cold streams similar* to sequences — the code inside a flow builder does not run until the flow is collected. This becomes clear in the following example:

```
fun simple(): Flow<Int> = flow {
    println("Flow started")
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}

fun main() = runBlocking<Unit> {
    println("Calling simple function...")
    val flow = simple()
    println("Calling collect...")
    flow.collect { value -> println(value) }
    println("Calling collect again...")
    flow.collect { value -> println(value) }
}
```

Which prints:

```
Calling simple function...
Calling collect...
Flow started
1
2
3
Calling collect again...
Flow started
1
```

```
Calling simple function...
Calling collect...
Flow started
1
2
3
Calling collect again...
Flow started
1
2
3
```

This is a key reason the `simple` function (which returns a flow) is not marked with `suspend` modifier. By itself, `simple()` call returns

Collecting from a flow

Use a *terminal operator* to trigger the flow to start listening for values. Terminal operators on the flow are either suspending functions such as `collect`, `single`, `reduce`, `toList`, etc. or `launchIn` operator that starts collection of the flow in the given scope. They are applied to the upstream flow and trigger execution of all operations. Execution of the flow is also called *collecting the flow* and is always performed in a suspending manner without actual blocking. Terminal operators complete normally or exceptionally depending on successful or failed execution of all the flow operations in the upstream. The most basic terminal operator is collect, for example:

```
try {
    flow.collect { value ->
        println("Received $value")
    }
} catch (e: Exception) {
    println("The flow has thrown an exception: $e")
}
```

By default, flows are *sequential* and all flow operations are executed sequentially in the same coroutine, with an exception for a few operations specifically designed to introduce concurrency into flow execution such as buffer and flatMapMerge.

What do you know about StateFlow and SharedFlow?

StateFlow and SharedFlow

`StateFlow` and `SharedFlow` are Flow APIs that enable flows to optimally emit state updates and emit values to multiple consumers.

StateFlow

`StateFlow` is a state-holder observable flow that emits the current and new state updates to its collectors. The current state value can also be read through its value `property`. To update state and send it to the flow, assign a new value to the `value` property of the `MutableStateFlow` class.

In Android, `StateFlow` is a great fit for classes that need to maintain an observable mutable state.

Following the examples from Kotlin flows, a `StateFlow` can be exposed from the `LatestNewsViewModel` so that the `View` can listen for UI state updates and inherently make the screen state survive configuration changes.

```
class LatestNewsViewModel(
    private val newsRepository: NewsRepository
) : ViewModel() {

    // Backing property to avoid state updates from other classes
    private val _uiState = MutableStateFlow(LatestNewsUiState.Success(emptyList()))
    // The UI collects from this StateFlow to get its state updates
    val uiState: StateFlow<LatestNewsUiState> = _uiState

    init {
        viewModelScope.launch {
            newsRepository.favoriteLatestNews
                // Update View with the latest favorite news
                // Writes to the value property of MutableStateFlow,
                // adding a new element to the flow and updating all
                // of its collectors
                .collect { favoriteNews ->
                    _uiState.value = LatestNewsUiState.Success(favoriteNews)
                }
        }
    }
}

// Represents different states for the LatestNews screen
sealed class LatestNewsUiState {
    data class Success(news: List<ArticleHeadline>): LatestNewsUiState()
    data class Error(exception: Throwable): LatestNewsUiState()
}
```

The class responsible for updating a `MutableStateFlow` is the producer, and all classes collecting from the `StateFlow` are the consumers. Unlike a cold flow built using the flow builder, a `StateFlow` is hot: collecting from the flow doesn't trigger any producer code. A `StateFlow` is always active and in memory, and it becomes eligible for garbage collection only when there are no other references to it from a garbage collection root.

To convert any flow to a `StateFlow`, use the `stateIn` intermediate operator.

`StateFlow` and `LiveData` have similarities. Both are observable data holder classes, and both follow a similar pattern when used in your app architecture.

Note, however, that StateFlow and LiveData do behave differently:

- `StateFlow` requires an initial state to be passed in to the constructor, while `LiveData` does not.
- `LiveData.observe()` automatically unregisters the consumer when the view goes to the `STOPPED` state, whereas collecting from a `StateFlow` or any other flow does not.

SharedFlow

The `shareIn` function returns a `SharedFlow`, a hot flow that emits values to all consumers that collect from it. A `SharedFlow` is a highly-configurable generalization of `StateFlow`.

You can create a `SharedFlow` without using `shareIn`. As an example, you could use a `SharedFlow` to send ticks to the rest of the app so that all the content refreshes periodically at the same time. Apart from fetching the latest news, you might also want to refresh the user information section with its favorite topics collection. In the following code snippet, a `TickHandler` exposes a `SharedFlow` so that other classes know when to refresh its content. As with `StateFlow`, use a backing property of type `MutableSharedFlow` in a class to send items to the flow:

```
// Class that centralizes when the content of the app needs to be refreshed
class TickHandler(
    private val externalScope: CoroutineScope,
    private val tickIntervalMs: Long = 5000
) {
    // Backing property to avoid flow emissions from other classes
    private val _tickFlow = MutableSharedFlow<Unit>(replay = 0)
    val tickFlow: SharedFlow<Event<String>> = _tickFlow

    init {
        externalScope.launch {
            while(true) {
                _tickFlow.emit(Unit)
                delay(tickIntervalMs)
            }
        }
    }
}

class NewsRepository(
    ...,
    private val tickHandler: TickHandler,
    private val externalScope: CoroutineScope
) {
    init {
        externalScope.launch {
            // Listen for tick updates
            tickHandler.tickFlow.collect {
                refreshLatestNews()
            }
        }
    }

    suspend fun refreshLatestNews() { ... }
    ...
}
```

You can customize the `SharedFlow` behavior in the following ways:

- `replay` lets you resend a number of previously-emitted values for new subscribers.
- `onBufferOverflow` lets you specify a policy for when the buffer is full of items to be sent. The default value is `BufferOverflow.SUSPEND`, which makes the caller suspend. Other options are `DROP_LATEST` or `DROP_OLDEST`.

`MutableSharedFlow` also has a `subscriptionCount` property that contains the number of active collectors so that you can optimize your business logic accordingly. `MutableSharedFlow` also contains a `resetReplayCache` function if you don't want to replay the latest information sent to the flow.

Example

Following class encapsulates an integer state and increments its value on each call to `inc:`

```
class CounterModel {
    private val _counter = MutableStateFlow(0) // private mutable state flow
    val counter = _counter.asStateFlow() // publicly exposed as read-only state flow

    fun inc() {
        _counter.value++
    }
}
```

Having two instances of the above `CounterModel` class one can define the sum of their counters like this:

```
val aModel = CounterModel()
val bModel = CounterModel()
val sumFlow: Flow<Int> = aModel.counter.combine(bModel.counter) { a, b -> a + b }
```