Coroutines & kotlin Flow

- **coroutines suspend launch**

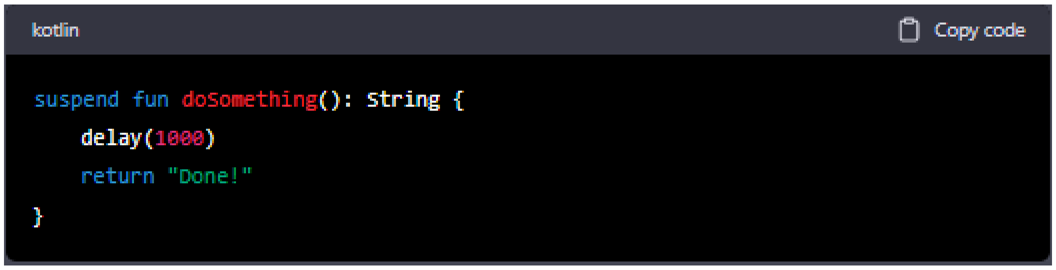
Coroutines

Coroutines are a lightweight concurrency framework introduced in Kotlin. They allow developers to write asynchronous code that looks and behaves like synchronous code, making it easier to read and reason about. Coroutines rely on suspending functions and cooperative multitasking to achieve their goal.

Suspend functions

Suspend functions are functions that can be paused and resumed later without blocking the thread they're running on. Suspend functions are defined with the suspend keyword and can only be called from within a coroutine or another suspend function.

Example of a suspend function:



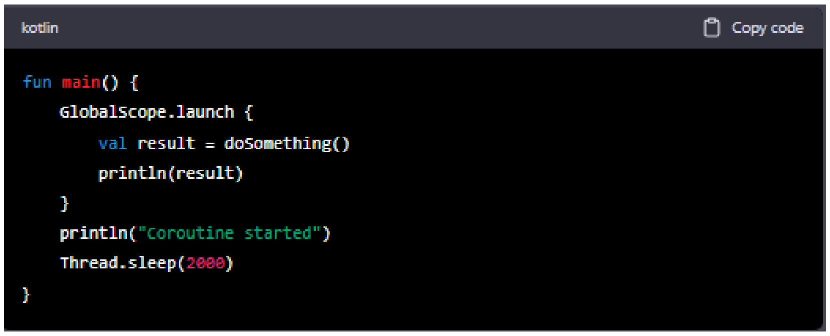
kotlin Copy code

```kotlin
suspend fun doSomething(): String {
    delay(1000)
    return "Done!"
}
```

Launch

launch is a coroutine builder used to start a new coroutine. It takes a suspending lambda as an argument, which is the code that will be executed asynchronously.

Example of launching a coroutine with launch:

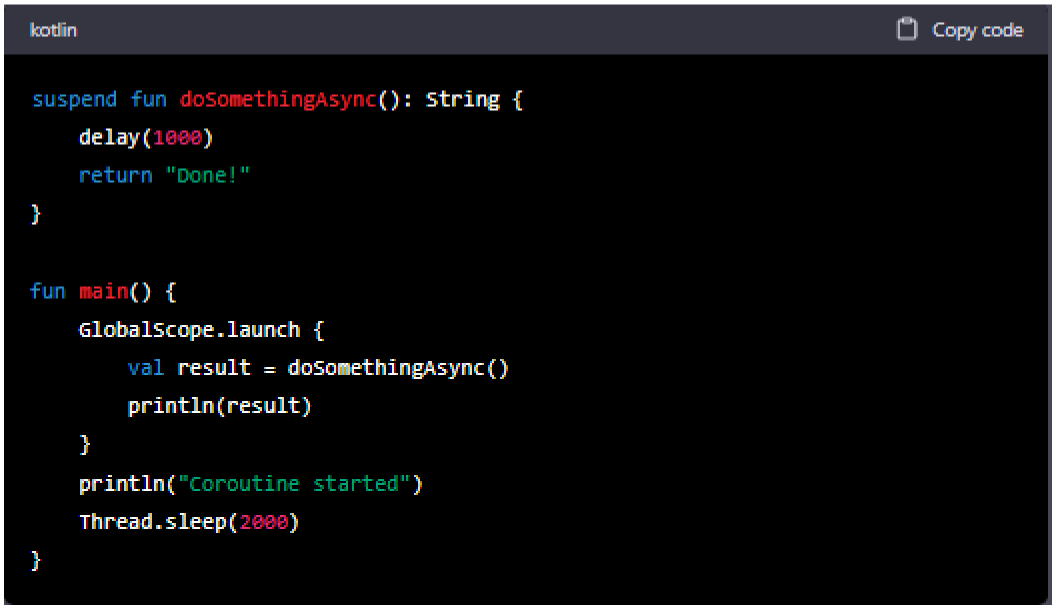


```kotlin
fun main() {
    GlobalScope.launch {
        val result = doSomething()
        println(result)
    }
    println("Coroutine started")
    Thread.sleep(2000)
}
```

In this example, the doSomething() function is called asynchronously within a coroutine created by the launch builder. The println() statement outside of the coroutine is executed immediately after launching the coroutine, so it is printed before the result of doSomething(). To make sure the coroutine has finished before the program exits, we use Thread.sleep(2000) to pause the main thread for 2 seconds.

Suspend and Launch Together

We can use suspend and launch together to create more complex asynchronous operations. For example:



```kotlin
suspend fun doSomethingAsync(): String {
    delay(1000)
    return "Done!"
}

fun main() {
    GlobalScope.launch {
        val result = doSomethingAsync()
        println(result)
    }
    println("Coroutine started")
    Thread.sleep(2000)
}
```

In this example, doSomethingAsync() is a suspending function that is called asynchronously within a coroutine created by launch. The result is printed after a 2-second delay, just like in the previous example.

That's it! These are the basics of coroutines, suspend, and launch in Kotlin.

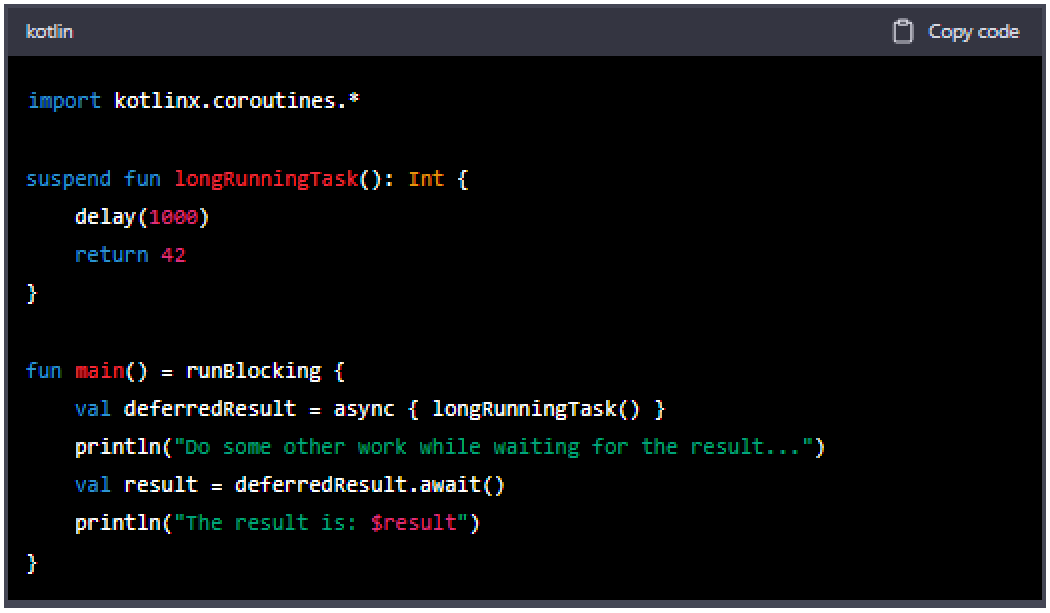
- **Async - await**

Async and await are concepts used in asynchronous programming to allow developers to write code that executes concurrently without blocking the main thread. In Kotlin, these concepts are implemented using the Kotlin Coroutines library.

async

Async allows you to create a new coroutine that runs concurrently with the main coroutine. You can use the async function to create a coroutine that returns a Deferred object, which is a lightweight future that represents a result that will be available at some point in the future. The async function starts a new coroutine to execute the specified block of code asynchronously and returns a Deferred object immediately.

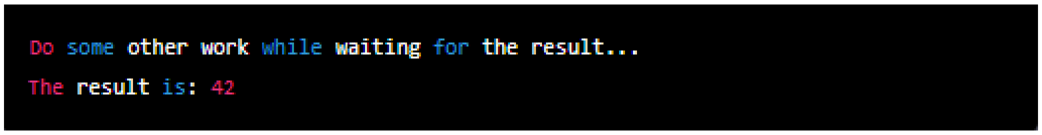
Here's an example of using async in Kotlin:



```kotlin
import kotlinx.coroutines.*

suspend fun longRunningTask(): Int {
    delay(1000)
    return 42
}

fun main() = runBlocking {
    val deferredResult = async { longRunningTask() }
    println("Do some other work while waiting for the result...")
    val result = deferredResult.await()
    println("The result is: $result")
}
```

In this example, the longRunningTask function simulates a long-running task by delaying for 1 second before returning a result. The async function is called to create a coroutine that executes the longRunningTask function asynchronously. The await function is then called on the Deferred object to retrieve the result when it becomes available.

The output of this program will be:

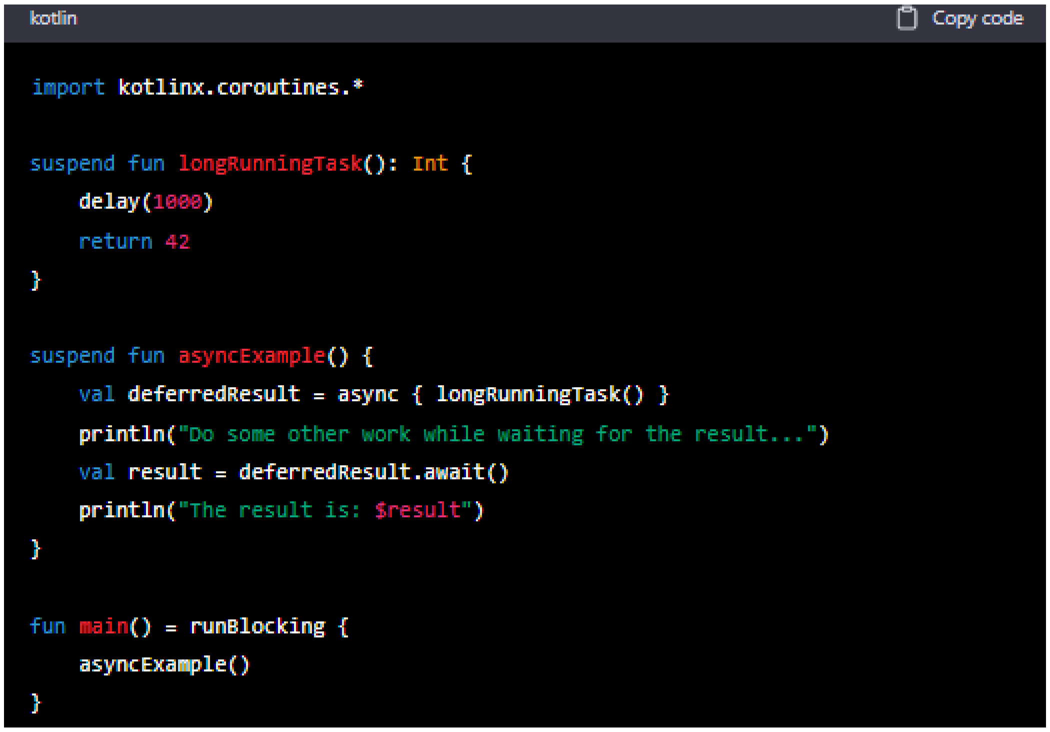


```
Do some other work while waiting for the result...
The result is: 42
```

await

Await is used to retrieve the result of a `Deferred` object. When you call `await` on a `Deferred` object, it suspends the current coroutine until the result is available. Once the result is available, the coroutine resumes execution.

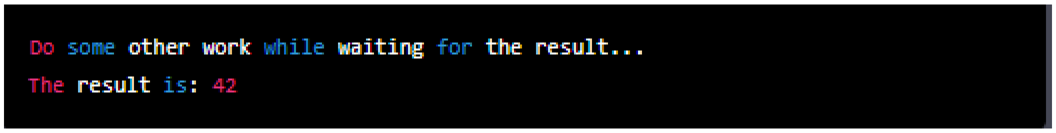
Here's an example of using await in Kotlin:



```kotlin
import kotlinx.coroutines.*

suspend fun longRunningTask(): Int {
    delay(1000)
    return 42
}

suspend fun asyncExample() {
    val deferredResult = async { longRunningTask() }
    println("Do some other work while waiting for the result...")
    val result = deferredResult.await()
    println("The result is: $result")
}

fun main() = runBlocking {
    asyncExample()
}
```

In this example, the `asyncExample` function creates a new coroutine using the `async` function, which executes the `longRunningTask` function asynchronously. The `await` function is then called on the `Deferred` object to retrieve the result when it becomes available.

The output of this program will be:



```
Do some other work while waiting for the result...
The result is: 42
```

In summary, async and await in Kotlin allow developers to write asynchronous code that executes concurrently without blocking the main thread. The `async` function creates a new coroutine that runs concurrently with the main coroutine and returns a `Deferred` object. The `await` function is used to retrieve the result of a `Deferred` object and suspends the current coroutine until the result is available.

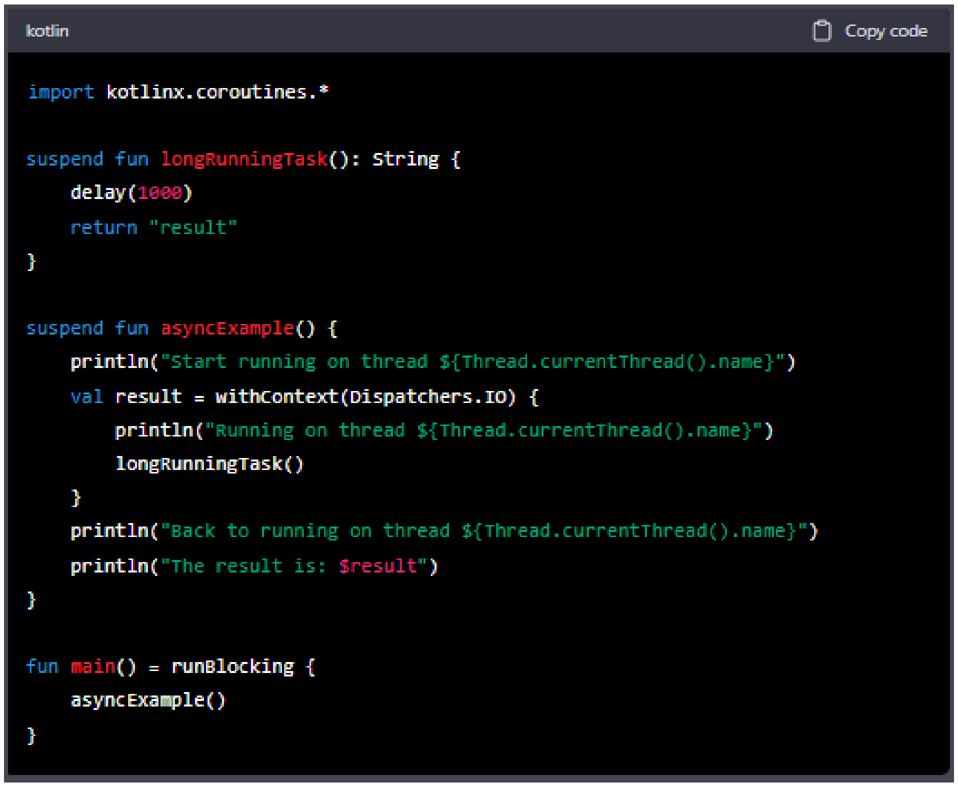
• withContext ,dispatchers ,scopes

withContext is a function provided by the Kotlin Coroutines library that allows you to switch the context in which a coroutine is running. The context of a coroutine defines the thread or threads on which the coroutine runs. By default, a coroutine runs on the same thread that launched it, but you can use withContext to switch to a different thread or dispatcher.

A dispatcher is an object that determines which thread or threads a coroutine runs on. The Kotlin Coroutines library provides several built-in dispatchers, such as Dispatchers.Default, which uses a shared pool of threads optimized for CPU-bound tasks, and Dispatchers.IO, which uses a shared pool of threads optimized for I/O-bound tasks.

A scope is an object that defines the lifecycle of a coroutine. Scopes are used to launch coroutines and to manage the cancellation of coroutines when they are no longer needed. The Kotlin Coroutines library provides several built-in scopes, such as GlobalScope, which launches coroutines that are not tied to any specific lifecycle, and CoroutineScope, which launches coroutines that are tied to a specific lifecycle, such as an activity or fragment.

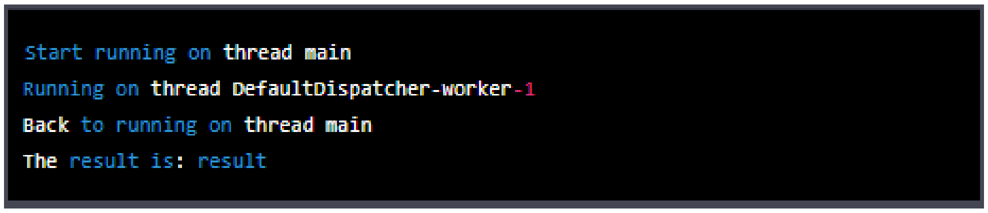
Here's an example of using withContext, dispatchers, and scopes in Kotlin:



```kotlin
import kotlinx.coroutines.*

suspend fun longRunningTask(): String {
    delay(1000)
    return "result"
}

suspend fun asyncExample() {
    println("Start running on thread ${Thread.currentThread().name}")
    val result = withContext(Dispatchers.IO) {
        println("Running on thread ${Thread.currentThread().name}")
        longRunningTask()
    }
    println("Back to running on thread ${Thread.currentThread().name}")
    println("The result is: $result")
}

fun main() = runBlocking {
    asyncExample()
}
```

In this example, the `longRunningTask` function simulates a long-running task by delaying for 1 second before returning a result. The `asyncExample` function uses `withContext` to switch to the `Dispatchers.IO` dispatcher, which uses a shared pool of threads optimized for I/O-bound tasks. The `longRunningTask` function is then executed on a thread from the `Dispatchers.IO` pool. Once the result is available, the `asyncExample` function switches back to the default dispatcher and resumes execution on the main thread.

The output of this program will be:



```
Start running on thread main
Running on thread DefaultDispatcher-worker-1
Back to running on thread main
The result is: result
```

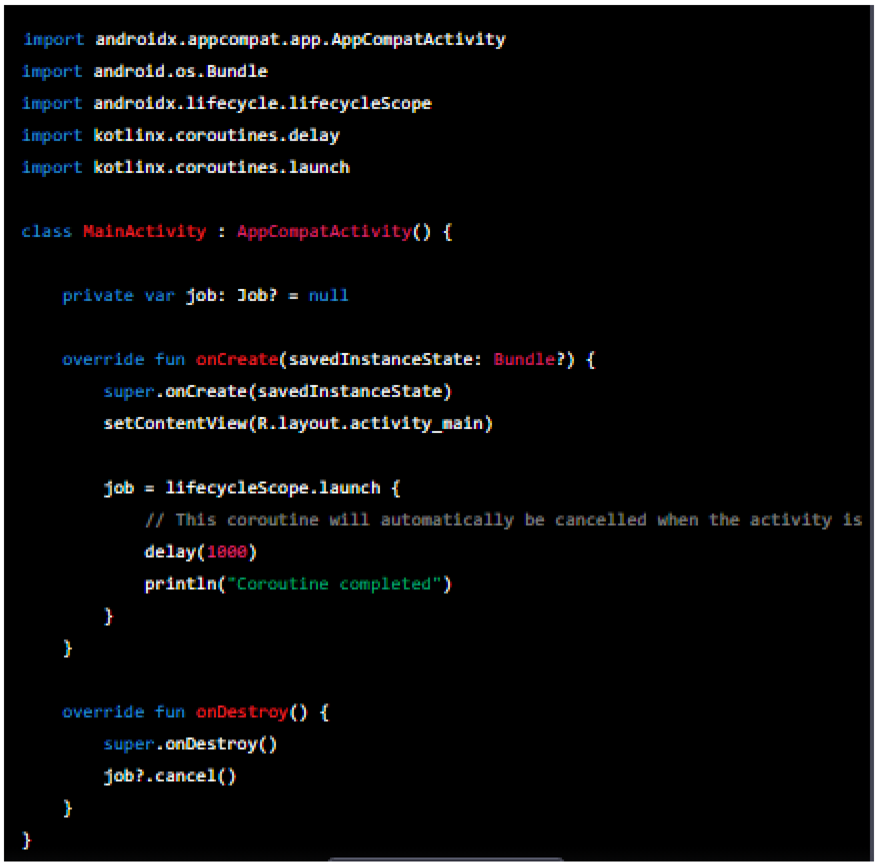
In summary, `withContext`, dispatchers, and scopes in Kotlin Coroutines allow developers to switch the context in which a coroutine is running, to define which thread or threads a coroutine runs on, and to manage the lifecycle of coroutines. The `withContext` function allows you to switch the context in which a coroutine is running. Dispatchers are objects that define which thread or threads a coroutine runs on. Scopes are objects that define the lifecycle of a coroutine. By using `withContext`, dispatchers, and scopes, developers can write concurrent and asynchronous code that is efficient and easy to read and maintain.

● Job lifeCycleScope

In Android, `Job` is an interface provided by the Kotlin Coroutines library that represents a unit of work that can be canceled or waited for completion. A `Job` is created when a coroutine is launched, and it can be used to track the status of the coroutine and to cancel the coroutine if necessary.

The lifecycleScope is a property of a LifecycleOwner, such as an activity or fragment, that provides a CoroutineScope tied to the lifecycle of the owner. This means that any coroutines launched using lifecycleScope will automatically be canceled when the owner is destroyed, such as when an activity is finished or a fragment is detached.

Here's an example of using Job and lifecycleScope in an Android activity:



```kotlin
import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import androidx.lifecycle.lifecycleScope
import kotlinx.coroutines.delay
import kotlinx.coroutines.launch

class MainActivity : AppCompatActivity() {

    private var job: Job? = null

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        job = lifecycleScope.launch {
            // This coroutine will automatically be cancelled when the activity is
            delay(1000)
            println("Coroutine completed")
        }
    }

    override fun onDestroy() {
        super.onDestroy()
        job?.cancel()
    }
}
```

In this example, the onCreate method of the activity launches a coroutine using lifecycleScope. The coroutine delays for 1 second before printing a message. The job variable is used to keep track of the coroutine and to cancel it when the activity is destroyed.

When the activity is created, the job variable is set to the result of launching a coroutine using lifecycleScope.launch. This means that the coroutine will automatically be cancelled when the activity is destroyed. When the activity is destroyed, the onDestroy method is called, which cancels the job if it is not null.

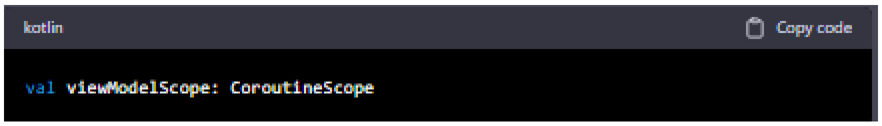
In summary, Job and lifecycleScope in Android allow developers to track the status of coroutines and to tie the lifecycles of coroutines to the lifecycles of Android components, such as activities and fragments. The Job interface represents a unit of work that can be cancelled or waited for completion. The lifecycleScope is a property of a LifecycleOwner that provides a CoroutineScope tied to the lifecycle of the owner. By using Job and lifecycleScope, developers can write concurrent and asynchronous code that is efficient and easy to read and maintain, while also ensuring that the code does not leak memory or cause other issues related to the Android lifecycle.

• viewModelScope

Overview:-

The viewModelScope is a CoroutineScope that is provided by the Android Architecture Components library to handle asynchronous operations in ViewModel classes. It allows you to launch coroutines that are bound to the lifecycle of the ViewModel, which means they will be canceled automatically when the ViewModel is cleared or destroyed.

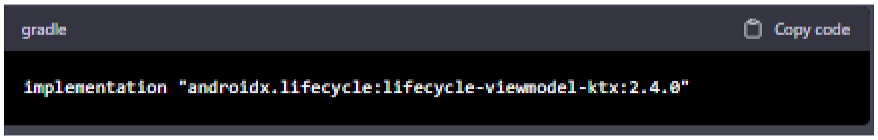
Declaration:-



```kotlin
val viewModelScope: CoroutineScope
```

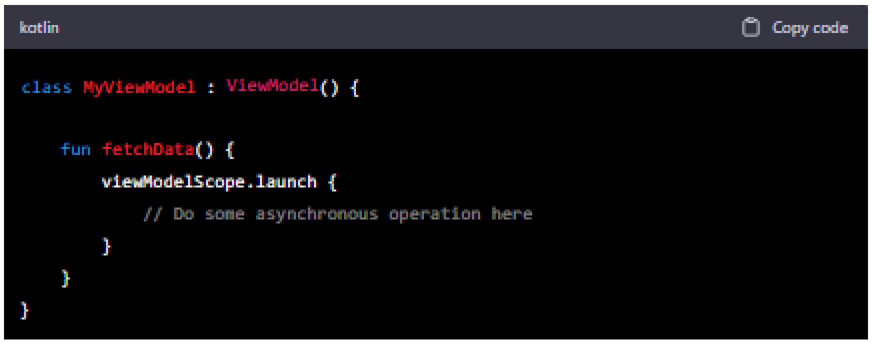
Usage:-

To use viewModelScope, you need to import the following dependency in your build.gradle file:



```gradle
implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.0"
```

Then, in your ViewModel class, you can launch a coroutine using `viewModelScope.launch` like this:



```kotlin
class MyViewModel : ViewModel() {

    fun fetchData() {
        viewModelScope.launch {
            // Do some asynchronous operation here
        }
    }
}
```

cancellation:-

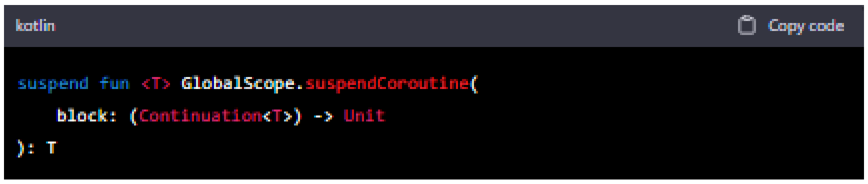
As mentioned earlier, the coroutines launched using viewModelScope will be canceled automatically when the ViewModel is cleared or destroyed. This means you don't have to worry about manually canceling your coroutines, as long as you're using viewModelScope.

- ## GlobalScope suspendCoroutine

Overview:-

The GlobalScope.suspendCoroutine function is a suspending function provided by the Kotlin coroutines library that allows you to suspend the current coroutine and start a new coroutine that can interact with the calling thread. It's typically used when you need to perform an asynchronous operation that doesn't have built-in coroutine support.

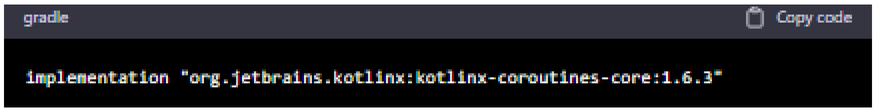
Declaration:-



```kotlin
suspend fun <T> GlobalScope.suspendCoroutine(
    block: (Continuation<T>) -> Unit
): T
```

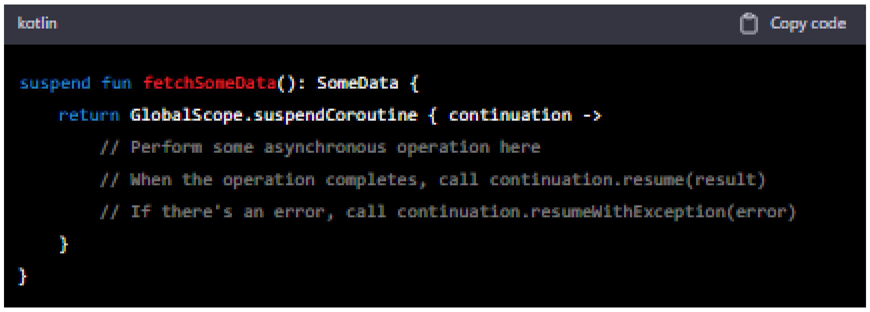
Usage:-

To use viewModelScope, you need to import the following dependency in your build.gradle file:



```gradle
implementation "org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.3"
```

Then, in your ViewModel class, you can launch a coroutine using GlobalScope.suspendCoroutine like this:



```kotlin
suspend fun fetchSomeData(): SomeData {
    return GlobalScope.suspendCoroutine { continuation ->
        // Perform some asynchronous operation here
        // When the operation completes, call continuation.resume(result)
        // If there's an error, call continuation.resumeWithException(error)
    }
}
```

In the code above, `fetchSomeData` is a suspending function that calls `GlobalScope.suspendCoroutine` to perform an asynchronous operation. The `Continuation` object passed to the `block` lambda represents the current coroutine, and it can be used to resume the coroutine with a result or an exception when the operation completes.

Cancellation:-

The coroutine launched by GlobalScope.suspendCoroutine is not bound to any specific lifecycle or scope, which means it won't be canceled automatically when the calling coroutine is canceled. You need to handle cancellation manually by checking the isActive property of the Continuation object.

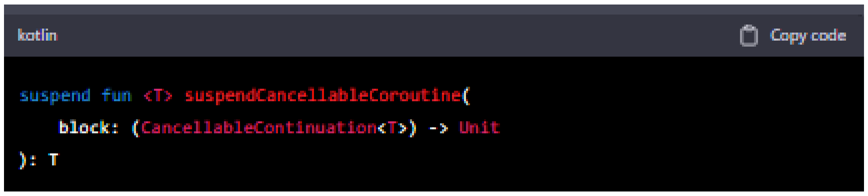
- ## suspendCancellableCoroutine , coroutineScope

suspendCancellableCoroutine

Overview:-

The suspendCancellableCoroutine function is a suspending function provided by the Kotlin coroutines library that allows you to suspend the current coroutine and start a new coroutine that can interact with the calling thread. It's similar to GlobalScope.suspendCoroutine, but it also provides a CancellableContinuation object that allows you to check and cancel the coroutine when needed.

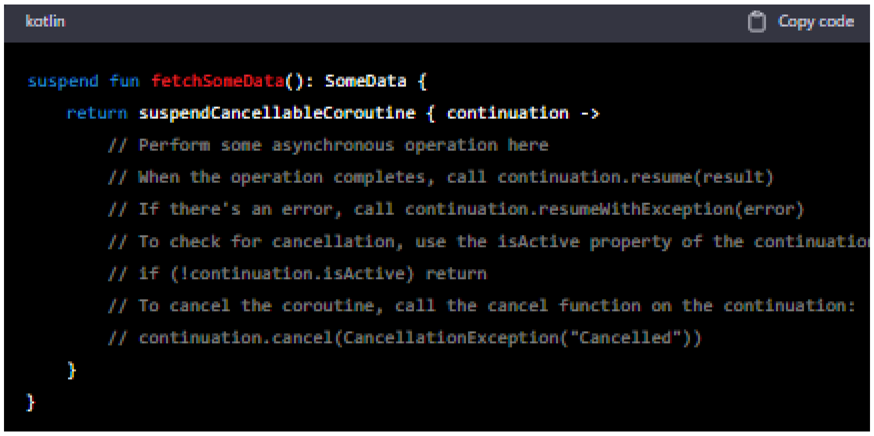
Declaration:-



```kotlin
suspend fun <T> suspendCancellableCoroutine(
    block: (CancellableContinuation<T>) -> Unit
): T
```

Usage:-

To use suspendCancellableCoroutine, you can call it like this:



```kotlin
suspend fun fetchSomeData(): SomeData {
    return suspendCancellableCoroutine { continuation ->
        // Perform some asynchronous operation here
        // When the operation completes, call continuation.resume(result)
        // If there's an error, call continuation.resumeWithException(error)
        // To check for cancellation, use the isActive property of the continuatio
        // if (!continuation.isActive) return
        // To cancel the coroutine, call the cancel function on the continuation:
        // continuation.cancel(CancellationException("Cancelled"))
    }
}
```

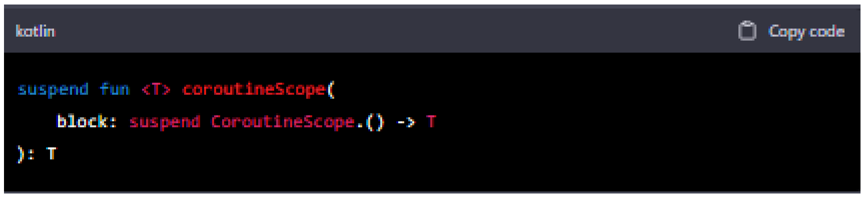
In the code above, fetchSomeData is a suspending function that calls suspendCancellableCoroutine to perform an asynchronous operation. The CancellableContinuation object passed to the block lambda represents the current coroutine, and it can be used to resume the coroutine with a result or an exception when the operation completes. It also provides methods to check and cancel the coroutine when needed.

CoroutineScope

Overview:-

The suspendCancellableCoroutine function is a suspending function provided by the Kotlin coroutines library that allows you to suspend the current coroutine and start a new coroutine that can interact with the calling thread. It's similar to GlobalScope.suspendCoroutine, but it also provides a CancellableContinuation object that allows you to check and cancel the coroutine when needed.

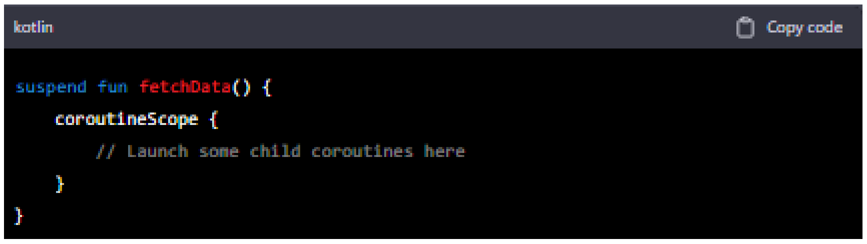
Declaration:-



```kotlin
suspend fun <T> coroutineScope(
    block: suspend CoroutineScope.() -> T
): T
```

Usage:-

To use coroutineScope, you can call it like this:



```kotlin
suspend fun fetchData() {
    coroutineScope {
        // Launch some child coroutines here
    }
}
```

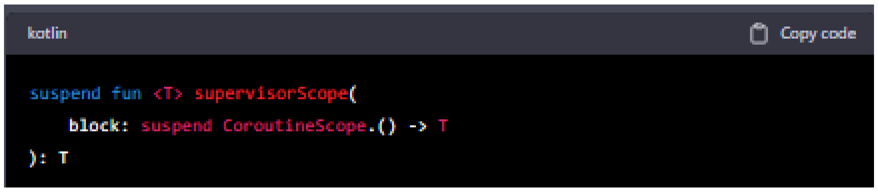
In the code above, fetchData is a suspending function that calls coroutineScope to create a new coroutine scope and launch some child coroutines in that scope. When the parent coroutine is canceled, all child coroutines will be canceled as well.

- ## supervisorScope

Overview:-

The supervisorScope function is similar to coroutineScope, but it creates a new coroutine scope with its own job that is independent of the parent coroutine's job. This means that if a child coroutine launched in the scope fails, it won't cancel the other child coroutines or the parent coroutine. Instead, the failure will be handled by the supervisorScope and can be caught and handled by the calling coroutine.

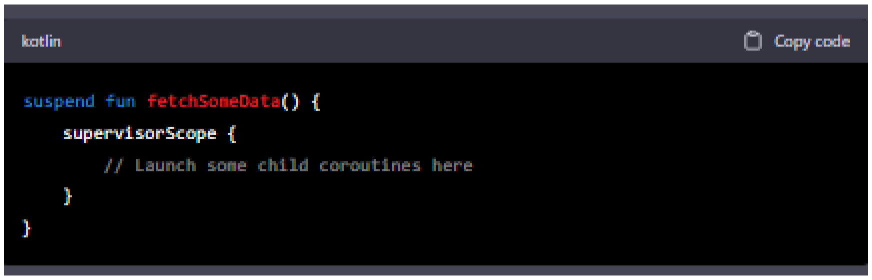
Declaration:-


kotlin Copy code

```kotlin
suspend fun <T> supervisorScope(
    block: suspend CoroutineScope.() -> T
): T
```

Usage:-

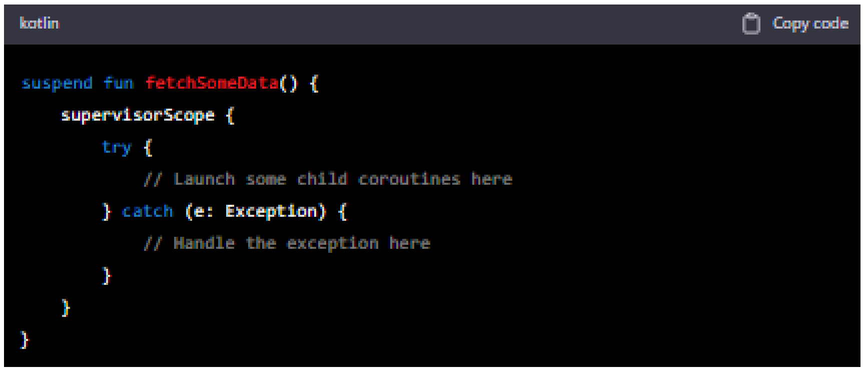
To use supervisorScope, you can call it like this:


kotlin Copy code

```kotlin
suspend fun fetchSomeData() {
    supervisorScope {
        // Launch some child coroutines here
    }
}
```

In the code above, fetchSomeData is a suspending function that calls supervisorScope to create a new coroutine scope and launch some child coroutines in that scope. If any of the child coroutines fail, they won't affect the other child coroutines or the parent coroutine.

You can also catch and handle exceptions thrown by child coroutines using try-catch blocks like this:



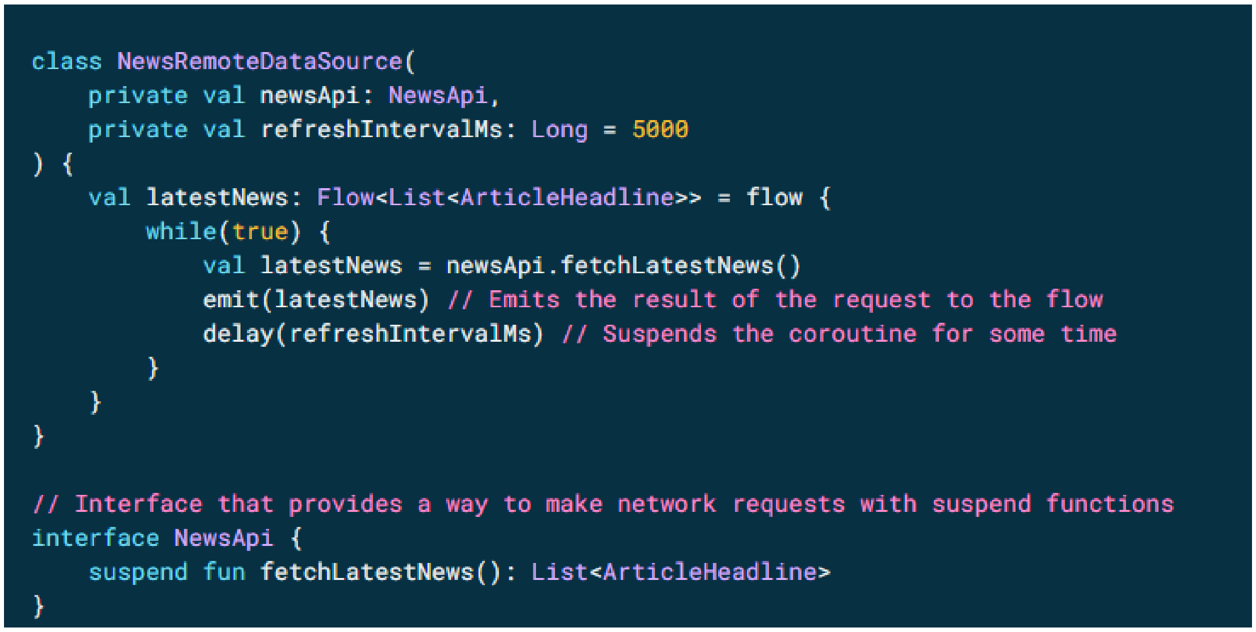
```kotlin
suspend fun fetchSomeData() {
    supervisorScope {
        try {
            // Launch some child coroutines here
        } catch (e: Exception) {
            // Handle the exception here
        }
    }
}
```

In the code above, the try-catch block is used to catch any exceptions thrown by the child coroutines launched in the supervisorScope.

- ## Flow Builder

To create flows, use the flow builder APIs. The flow builder function creates a new flow where you can manually emit new values into the stream of data using the emit function.

In the following example, a data source fetches the latest news automatically at a fixed interval. As a suspend function cannot return multiple consecutive values, the data source creates and returns a flow to fulfill this requirement. In this case, the data source acts as the producer.



```kotlin
class NewsRemoteDataSource(
    private val newsApi: NewsApi,
    private val refreshIntervalMs: Long = 5000
) {
    val latestNews: Flow<List<ArticleHeadline>> = flow {
        while(true) {
            val latestNews = newsApi.fetchLatestNews()
            emit(latestNews) // Emits the result of the request to the flow
            delay(refreshIntervalMs) // Suspends the coroutine for some time
        }
    }
}

// Interface that provides a way to make network requests with suspend functions
interface NewsApi {
    suspend fun fetchLatestNews(): List<ArticleHeadline>
}
```

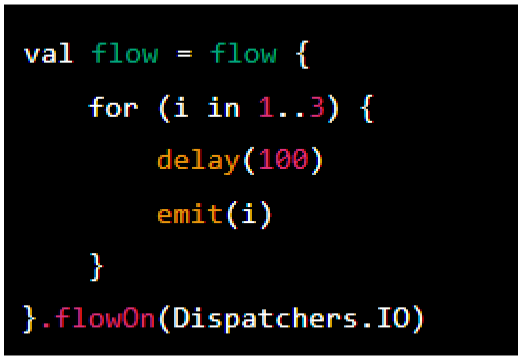
The flow builder is executed within a coroutine. Thus, it benefits from the same asynchronous APIs, but some restrictions apply:

- Flows are *sequential*. As the producer is in a coroutine, when calling a suspend function, the producer suspends until the suspend function returns. In the example, the producer suspends until the fetchLatestNews network request completes. Only then is the result emitted to the stream.
- With the flow builder, the producer cannot emit values from a different CoroutineContext. Therefore, don't call emit in a different CoroutineContext by creating new coroutines or by using withContext blocks of code. You can use other flow builders such as callbackFlow in these cases.

- **Collector flowOn**

flowOn is an operator in Kotlin Flow that allows you to change the context or thread on which the flow is executed. This operator is especially useful when you need to perform a CPU-intensive operation, such as parsing or filtering large amounts of data, on a different thread than the one on which the flow is originally executed.

Here's an example:



```
val flow = flow {
    for (i in 1..3) {
        delay(100)
        emit(i)
    }
}.flowOn(Dispatchers.IO)
```

In this example, we create a flow with the values 1, 2, and 3 using the `flow` builder. We also use the `delay` function to simulate a long-running operation. Finally, we use the `flowOn` operator to switch the context to the IO dispatcher, which is optimized for blocking I/O operations. This means that the values will be emitted on a different thread than the one on which the flow was originally executed.

The `flowOn` operator takes a `CoroutineDispatcher` parameter, which specifies the context or thread on which the flow should be executed. You can use one of the built-in dispatchers, such as `Dispatchers.IO`, `Dispatchers.Default`, or `Dispatchers.Main`, or you can create your own custom dispatcher.

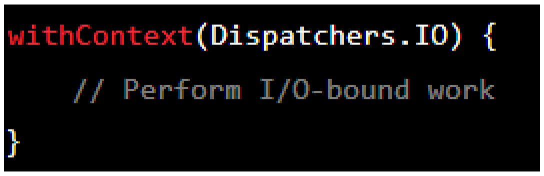
Note that the flowOn operator only affects the downstream operators in the flow. It does not affect the upstream operators or the initial producer of the flow. If you need to change the context of the initial producer of the flow, you can use the flow builder with the flowOn operator.

• dispatchers Operators

Dispatchers are used to control the execution context of coroutines. They determine on which thread or threads a coroutine should be executed. By default, coroutines run in the context of the calling thread, but we can use a dispatcher to change the execution context. Kotlin provides three main dispatchers, which are:

- `Dispatchers.Default`: For CPU-intensive work.
- `Dispatchers.IO`: For I/O-bound work.
- `Dispatchers.Main`: For executing work on the main thread.

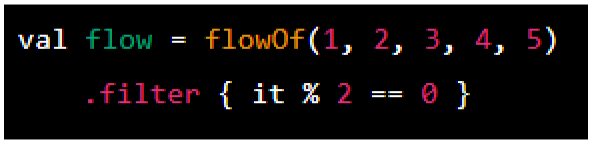
We can switch the execution context of a coroutine using the withContext function, which takes a dispatcher as an argument. Here's an example:



```
withContext(Dispatchers.IO) {
    // Perform I/O-bound work
}
```

- ## Filter operator

The filter operator is used to filter out items emitted by a flow based on a given condition. It takes a predicate function as a parameter that evaluates each emitted item and returns true if the item should be included in the resulting flow, or false if it should be excluded. Here's an example:



```kotlin
val flow = flowOf(1, 2, 3, 4, 5)
    .filter { it % 2 == 0 }
```

In this example, we create a flow of integers and apply the filter operator to only include even numbers in the resulting flow.

Overall, Dispatchers are used to switch the execution context of coroutines, while Operators are used to transform, filter, or combine flows. The filter operator is an example of an Operator used to filter out items emitted by a flow based on a given condition.

- ## Map

The map operator is one of the most commonly used operators in Kotlin Flow. It is used to transform each item emitted by a flow into a new item of a different type. The new item can be of the same type as the original item or a completely different type.

Here's an example of how to use the map operator:

```
val flow = flowOf(1, 2, 3, 4, 5)
    .map { it * 2 }
```

In this example, we create a flow of integers and apply the map operator to transform each integer into its double value. The resulting flow will emit 2, 4, 6, 8, and 10.
The map operator takes a lambda function that maps the emitted item to a new value. The lambda function can be a simple expression or a more complex function. Here's an example of a more complex function:

```
data class User(val id: Int, val name: String)

val flow = flowOf(User(1, "John"), User(2, "Jane"), User(3, "Bob"))
    .map { user -> "User ${user.id}: ${user.name}" }
```

In this example, we create a flow of User objects and apply the map operator to transform each user into a formatted string. The resulting flow will emit "User 1: John", "User 2: Jane", and "User 3: Bob".
The map operator can also be used to transform a flow of one type into a flow of a completely different type. Here's an example:

```
val flow = flowOf(1, 2, 3, 4, 5)
    .map { "Number $it" }
```

In this example, we create a flow of integers and apply the map operator to transform each integer into a string. The resulting flow will emit "Number 1", "Number 2", "Number 3", "Number 4", and "Number 5".
Overall, the map operator is a powerful tool in Kotlin Flow that allows you to transform each item emitted by a flow into a new item of a different type. It is commonly used to perform data transformations and mapping operations.

- ## Zip

Please refer below link:
https://www.section.io/engineering-education/kotlin-flow-zip-operator/#what-is-a-zip-operator

- ## flatMapConcat

```
@FlowPreview
fun <T, R> Flow<T>.flatMapConcat(transform: suspend (T) -> Flow<R>): Flow<R>
```

Transforms elements emitted by the original flow by applying transform, that returns another flow, and then concatenating and flattening these flows.

This method is a shortcut for `map(transform).flattenConcat()`. See flattenConcat.

Note that even though this operator looks very familiar, we discourage its usage in regular application-specific flows. Most likely, suspending operation in map operator will be sufficient and linear transformations are much easier to reason about.

- **retry**

Refer below link:

https://amitshekhar.me/blog/retry-operator-in-kotlin-flow

- **Debounce**

refer below link

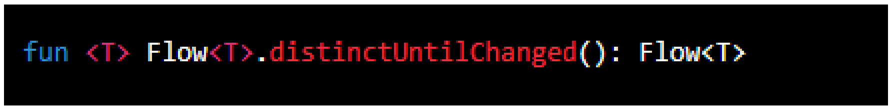
https://kotlinlang.org/api/kotlinx.coroutines/kotlinx-coroutines-core/kotlinx.coroutines.flow/debounce.html

- ## DistinctuntilChanged

distinctUntilChanged is an operator in Kotlin coroutines Flow that filters out consecutive duplicate values emitted by a Flow. It ensures that only unique values are passed along to downstream collectors.

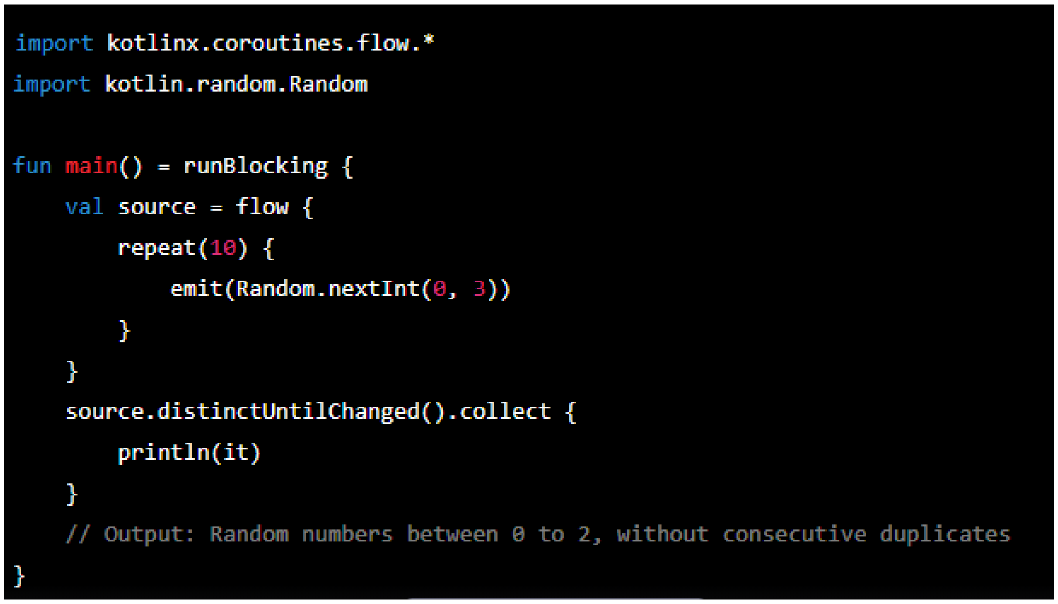
The operator compares the current value emitted by the Flow with the previous value. If the values are equal, the current value is discarded and not forwarded to downstream collectors. If the values are different, the current value is emitted and passed along to downstream collectors.

Here's the signature of distinctUntilChanged in Kotlin:



```kotlin
fun <T> Flow<T>.distinctUntilChanged(): Flow<T>
```

Example:



```kotlin
import kotlinx.coroutines.flow.*
import kotlin.random.Random

fun main() = runBlocking {
    val source = flow {
        repeat(10) {
            emit(Random.nextInt(0, 3))
        }
    }
    source.distinctUntilChanged().collect {
        println(it)
    }
    // Output: Random numbers between 0 to 2, without consecutive duplicates
}
```

In this example, the source Flow emits several consecutive duplicate values. The distinctUntilChanged operator filters out the duplicate values, so only unique values are passed along to the collect function.

• flatMapLatest

```
@ExperimentalCoroutinesApi
inline fun <T, R> Flow<T>.flatMapLatest(crossinline transform:
suspend (T) -> Flow<R>): Flow<R>
```

Returns a flow that switches to a new flow produced by the transform function every time the original flow emits a value. When the original flow emits a new value, the previous flow produced by the transform block is canceled.

For example, the following flow:

```
flow {
    emit("a")
    delay(100)
    emit("b")
}.flatMapLatest { value ->
    flow {
        emit(value)
        delay(200)
        emit(value + "_last")
    }
}
```

Output

produces `a b b_last`

- ## Terminal Operators

Refer below link:

https://amitshekhar.me/blog/terminal-operators-in-kotlin-flow

- ## Cold flow vs hot flow

Refer below link:

https://amitshekhar.me/blog/cold-flow-vs-hot-flow

- ## StateFlow vs SharedFlow

Refer below link:

https://amitshekhar.me/blog/stateflow-and-sharedflow

- ## callbackflow

fun <T> callbackFlow(block: **suspend** ProducerScope<T>.() -> Unit): Flow<T>

Creates an instance of a *cold*Flow with elements that are sent to a SendChannel provided to the builder's block of code via ProducerScope. It allows elements to be produced by code that is running in a different context or concurrently.

The resulting flow is *cold*, which means that block is called every time a terminal operator is applied to the resulting flow.

This builder ensures thread-safety and context preservation, thus the provided ProducerScope can be used from any context, e.g. from a callback-based API. The resulting flow completes as soon as the code in the block completes. awaitClose should be used to keep the flow running, otherwise the channel will be closed immediately when block completes. awaitClose argument is called either when a flow consumer cancels the flow collection or when a callback-based API invokes SendChannel.close manually and is typically used to cleanup the resources after the completion, e.g. unregister a callback. Using awaitClose is mandatory in order to prevent memory leaks when the flow collection is canceled, otherwise the callback may keep running even when the flow collector is already completed. To avoid such leaks, this method throws IllegalStateException if block returns, but the channel is not closed yet.

A channel with the default buffer size is used. Use the buffer operator on the resulting flow to specify a user-defined value and to control what happens when data is produced faster than consumed, i.e. to control the back-pressure behavior.

Adjacent applications of callbackFlow, flowOn, buffer, and produceIn are always fused so that only one properly configured channel is used for execution.

Example of usage that converts a multi-shot callback API to a flow. For single-shot callbacks use suspendCancellableCoroutine.

```kotlin
fun flowFrom(api: CallbackBasedApi): Flow<T> = callbackFlow {
    val callback = object : Callback { // Implementation of some callback interface
        override fun onNextValue(value: T) {
            // To avoid blocking you can configure channel capacity using
            // either buffer(Channel.CONFLATED) or buffer(Channel.UNLIMITED) to avoid overfill
            trySendBlocking(value)
                .onFailure { throwable ->
                    // Downstream has been cancelled or failed, can log here
                }
        }
        override fun onApiError(cause: Throwable) {
            cancel(CancellationException("API Error", cause))
        }
        override fun onCompleted() = channel.close()
    }
    api.register(callback)
    /*
     * Suspends until either 'onCompleted'/'onApiError' from the callback is invoked
     * or flow collector is cancelled (e.g. by 'take(1)' or because a collector's coroutine was cancelled).
     * In both cases, callback will be properly unregistered.
     */
        awaitClose { api.unregister(callback) }
    }
```

- ## Channelflow

```
fun <T> channelFlow(block: suspend ProducerScope<T>.() -> Unit):
Flow<T>
```

Creates an instance of a coldFlow with elements that are sent to a SendChannel provided to the builder's block of code via ProducerScope. It allows elements to be produced by code that is running in a different context or concurrently.
The resulting flow is cold, which means that block is called every time a terminal operator is applied to the resulting flow.

This builder ensures thread-safety and context preservation, thus the provided ProducerScope can be used from any context, e.g. from a callback-based API. The resulting flow completes as soon as the code in the block completes. awaitClose should be used to keep the flow running, otherwise the channel will be closed immediately when block completes. awaitClose argument is called either when a flow consumer cancels the flow collection or when a callback-based API invokes SendChannel.close manually and is typically used to cleanup the resources after the completion, e.g. unregister a callback. Using awaitClose is mandatory in order to prevent memory leaks when the flow collection is canceled, otherwise the callback may keep running even when the flow collector is already completed. To avoid such leaks, this method throws IllegalStateException if block returns, but the channel is not closed yet.

A channel with the default buffer size is used. Use the buffer operator on the resulting flow to specify a user-defined value and to control what happens when data is produced faster than consumed, i.e. to control the back-pressure behavior.

Adjacent applications of callbackFlow, flowOn, buffer, and produceIn are always fused so that only one properly configured channel is used for execution.

Example of usage that converts a multi-shot callback API to a flow. For single-shot callbacks use suspendCancellableCoroutine.

```kotlin
fun flowFrom(api: CallbackBasedApi): Flow<T> = callbackFlow {
    val callback = object : Callback { // Implementation of some callback interface
        override fun onNextValue(value: T) {
            // To avoid blocking you can configure channel capacity using
            // either buffer(Channel.CONFLATED) or buffer(Channel.UNLIMITED) to avoid overfill
            trySendBlocking(value)
                .onFailure { throwable ->
                    // Downstream has been cancelled or failed, can log here
                }
        }
        override fun onApiError(cause: Throwable) {
            cancel(CancellationException("API Error", cause))
        }
        override fun onCompleted() = channel.close()
    }
    api.register(callback)
    /*
     * Suspends until either 'onCompleted'/'onApiError' from the callback is invoked
     * or flow collector is cancelled (e.g. by 'take(1)' or because a collector's coroutine was cancelled).
     * In both cases, callback will be properly unregistered.
     */
        awaitClose { api.unregister(callback) }
    }
```