Coroutines Kotlin

Download the Full App created in this Guide:
https://drive.google.com/file/d/1PDhzzYXb4lRXYNuniU2YMpQflv07JAKZ/view?usp=sharing

Download the Architecture Project with Coroutines support
https://drive.google.com/file/d/1azpucisy6VxHNAPIVg2EecD_MqAxn4L7/view?usp=sharing

Coroutines is Google's recommended solution for asynchronous programming on Android.

Coroutines are:

- **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.

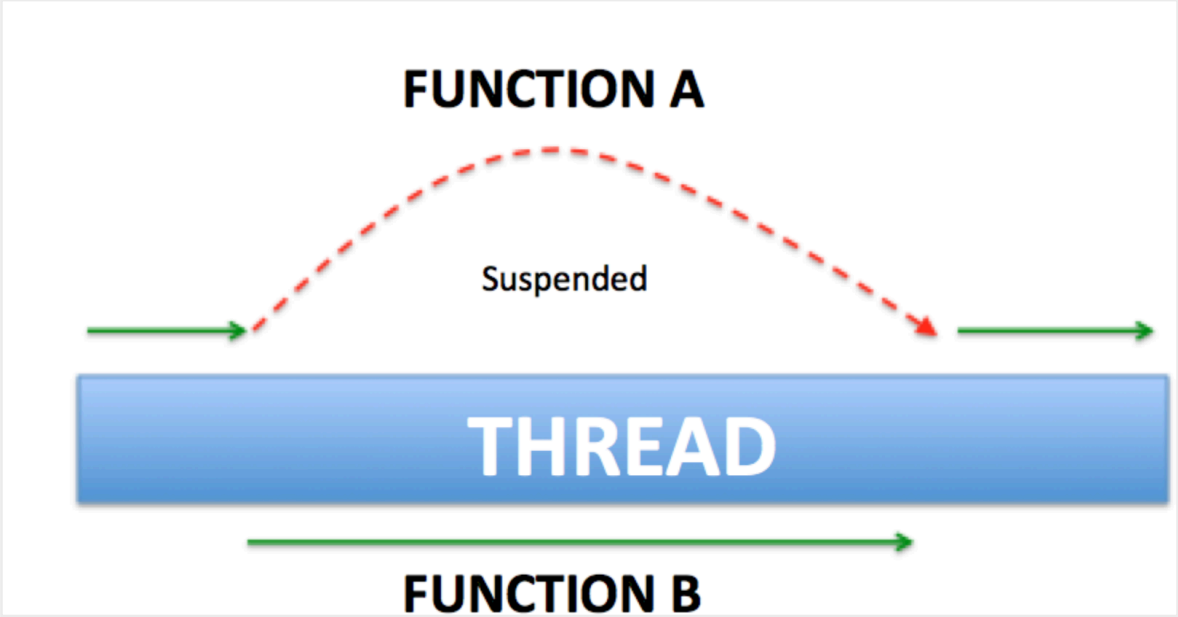
Co - cooperate
Routines - functions

Timing functions

One can think of a coroutine as a light-weight thread. Like threads, coroutines can run in parallel, wait for each other and communicate. The biggest difference is that coroutines are very cheap, almost free: we can create thousands of them, and pay very little in terms of performance. True threads, on the other hand, are expensive to start and keep around. A thousand threads can be a serious challenge for a modern machine.

Suspended functions are at the center of everything in coroutines. A suspended function is simply a function that can be paused and resumed at a later time. They can execute a long running operation and wait for it to complete without blocking. You can even stop or suspend your function while you are waiting for a callback and continue it when you get the result.
This is why You can run many coroutines on a single thread. Suspension doesn't block the thread where the coroutine is running. We save allot of

memory by not blocking the thread.



The exact definition of Coroutines: A framework to manage concurrency in a more performant and simple way with its lightweight thread which is written on top of the actual threading framework to get the most out of it by taking the advantage of cooperative nature of functions.

ViewModel, LiveData and LifeCycle includes a set of KTX extensions that work directly with coroutines. We will see later on.

First you need to import the latest Kotlin coroutines to your android studio project. You can find the latest version here. Follow the Gradle instructions. Add both the core and the android libraries :

https://github.com/Kotlin/kotlinx.coroutines

implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-android:1.6.0'
implementation 'org.jetbrains.kotlinx:kotlinx-coroutines-core:1.6.0'

Then create the following activity that suppose to fetch a user from the database (fetching illustrated here by sleep - sometimes working and sleeping is exactly the same:)) and updating it's details in Text View(make sure your default activity_main.xml file has a Text View whose id is text_view and use view binding.

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding

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

    private fun fetchAndShowUser() {
        val user = fetchUser()
        showUser(user)
    }

    private fun fetchUser() : User {
        Thread.sleep( millis: 3000) //no checked exceptions on kotlin
        return User( name: "Moshe", email: "moshe@moshe.com")
    }

    private fun showUser(user:User) {
        binding.textView.text = user.toString()
    }
}

data class User(val name:String, val email:String)
```

If we execute it like this it would result in a very poor user experience - a stuck app for 3 secs!

We have to define a Coroutine scope.

Coroutine scope promotes structured concurrency, whereby you can launch multiple coroutines in the same scope and cancel the scope (which in turn cancels all the coroutines within that scope).

if we are not already in a coroutine scope we can use the **GlobalScope** which the the scope of all the app. As along as the app is alive all our coroutines can run in this scope(there are more scopes and working with this scope is not recommended)
A global CoroutineScope not bound to any job. Global scope is used to launch top-level coroutines which are operating on the whole application lifetime and

are not cancelled prematurely.

Active coroutines launched in GlobalScope do not keep the process alive. They are like daemon threads.
This is a delicate API. It is easy to accidentally create resource or memory leaks when GlobalScope is used. A coroutine launched in GlobalScope is not subject to the principle of structured concurrency, so if it hangs or gets delayed due to a problem (e.g. due to a slow network), it will stay working and consuming resources until the app finishes.

From the GlobalScope we can call **launch**() that can execute our coroutines and return a job object that can be started if the coroutine start is lazy (by default all coroutines created with this function are executed immediately) or cancelled later on, or **async()** which is almost the same except it returns a Deferred object containing the Coroutine result. The coroutine created with this function is cancelled if the deferred object is cancelled.

Both function need a CoroutuineContext. The coroutine Context has a default value.
The coroutine context includes a *coroutine dispatcher* that determines what thread or threads the corresponding coroutine uses for its execution. The coroutine dispatcher can confine coroutine execution to a specific thread, dispatch it to a thread pool, or let it run unconfined.

CoroutineDispatcher tells the coroutine builder (in our case launch{} or async{}) as to which pool of threads is to be used. There are a few predefined Dispatchers available.

- **Dispatchers.Default**: CPU-intensive work, such as sorting large lists, doing complex calculations and similar. A shared pool of threads on the JVM backs it.
- **Dispatchers.IO**: networking or reading and writing from files. In short – any input and output, as the name states
- **Dispatchers.Main**: mandatory dispatcher for performing UI-related events in Android's main or UI thread.
- Dispatchers.Unconfined - A coroutine dispatcher that is not confined to any specific thread. The unconfined dispatcher is appropriate for coroutines which neither consume CPU time nor update any shared data (like UI) confined to a specific thread. From the Kotlin docs: The unconfined dispatcher is an advanced mechanism that can be helpful in certain corner cases where dispatching of a coroutine for its execution later is not needed or produces undesirable side-effects, because some operation in a coroutine must be performed right away. The unconfined dispatcher should not be used in general code.

When **launch {}** is used without parameters, it inherits the context (and thus dispatcher) from the CoroutineScope it is being launched from. In this case of

the Global Scope its default context is the the Dispatcher.Default. We can specify Any other Context we want to the GloablScope and run the launch or async result on the Dispatcher.IO like that:

GlobalScope.launch(Dispatchers.IO) {}

Try commenting out the existing code and run the following (watch the output):

```
GlobalScope.launch {
    println("Default      : I'm working in thread ${Thread.currentThread().name}")
}
GlobalScope.launch(Dispatchers.IO) {
    println("IO           : I'm working in thread ${Thread.currentThread().name}")
}
GlobalScope.launch(Dispatchers.Main) {
    println("Main              : I'm working in thread $
{Thread.currentThread().name}")
}
GlobalScope.launch(newSingleThreadContext("MyOwnThread")) { // will get its
own new thread
    println("newSingleThreadContext: I'm working in thread $
{Thread.currentThread().name}")
}
```

So to shorten our definitions: Each coroutine is a Job, a job must run in a scope for efficient memory management and must receive a Context which include the Dispatcher - which threads the coroutine will run on.

First step:

```
GlobalScope.launch(Dispatchers.IO) {  this: CoroutineScope

    fetchAndShowUser()
}
```

This will cause the app to crash when updating ui from the background!

If we call **async** we get Deferred value that can be extract using **await()** - this is very similar to Future in Java
Please note the await() is a suspended function (it make sense cause it can't run before the function that it is working on will finish) that can only be called from another suspended function or a Coroutine context.

Second step:

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

    GlobalScope.launch(Dispatchers.Main) { this: CoroutineScope
        fetchAndShowUser()
    }
}

private suspend fun fetchAndShowUser() {
    val user = fetchUser()
    showUser(user)
}

private suspend fun fetchUser()  = GlobalScope.async (Dispatchers.IO) {
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Moshe", email: "moshe@moshe.com")
}.await()
```

Another option instead of turning the fetch function to suspended is to call await() in the fetchAndShowUser

```
private suspend fun fetchAndShowUser() {
    val user = fetchUser()
    showUser(user.await())
}
```

Instead of calling from the GlobalScope and await for the result we can use the function withContext that creates a suspended function that runs in the given (or from the scope) Context.

Complete definition: Calls the specified suspending block with a given coroutine context, suspends until it completes, and returns the result.
Before the KTX this is what we used.

suspend fun <T> withContext(context: CoroutineContext, block: suspend CoroutineScope.() -> T): T

```
private suspend fun fetchUser()  = withContext(Dispatchers.IO) {
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Moshe",  email: "moshe@moshe.com")
}
```

withContext like await() it is also a suspended function (it make sense since it must suspend the coroutine until the work is finished) so the fetchUser must also be suspended.
Let's see what are the consequences of using withContext:

Add another Text View to your xml file and change both ids to text_view_one and text_view_two

For example this code will take 6 secs - First the first user is fetched and only then does the fetching of the second user starts (you can change the Thread.sleep() to the suspended function delay())

```
private suspend fun fetchAndShowUser() {
    val user1 = fetchUserOne()
    Log.d(TAG, msg: "user one fetched")

    val user2 = fetchUserTwo()
    Log.d(TAG, msg: "user two fetched")

    showUser(user1,user2)
}

private fun showUser(user1:User, user2:User) {
    binding.textViewOne.text = user1.toString()
    binding.textViewTwo.text = user2.toString()
}

private suspend fun fetchUserOne() = withContext(Dispatchers.IO) { this: Co
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Moshe",  email: "moshe@moshe.com")
}

private suspend fun fetchUserTwo() = withContext(Dispatchers.IO) { this: Co
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Misha",  email: "misha@misha.com")
}
```

Which is exactly the same as this:

```
private suspend fun fetchUserOne() = GlobalScope.async(Dispatchers.IO) {
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Moshe",  email: "moshe@moshe.com")
}.await()

private suspend fun fetchUserTwo() = GlobalScope.async(Dispatchers.IO) {
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Misha",  email: "misha@misha.com")
}.await()
```

Our fetch and show user waits for each user to be fetched and they are fetched one after the other and not simultaneously.
But if we change our code to the following all fetching work will at the same time:

```
private suspend fun fetchAndShowUser() {
    val user1 = fetchUserOne()
    Log.d(TAG, msg: "user one fetched")

    val user2 = fetchUserTwo()
    Log.d(TAG, msg: "user two fetched")

    showUser(user1.await(),user2.await())
}

private fun showUser(user1:User, user2:User) {
    binding.textViewOne.text = user1.toString()
    binding.textViewTwo.text = user2.toString()
}

private fun fetchUserOne() = GlobalScope.async(Dispatchers.IO) {
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Moshe",  email: "moshe@moshe.com")
}

private fun fetchUserTwo() = GlobalScope.async(Dispatchers.IO) {
    Thread.sleep( millis: 3000) //no checked exceptions on kotlin
    User( name: "Misha",  email: "misha@misha.com")
}
```

And this is an advantage async has, we can't achieve with witchContext!

Coroutines are not a new concept, let alone invented by Kotlin. They've been around for decades and are popular in some other programming languages such as Go. What is important to note though is that the way they're implemented in Kotlin, most of the functionality is delegated to libraries. In fact, beyond the suspend keyword, no other keywords are added to the language. This is somewhat different from languages such as C# that have async and await as part of the syntax. With Kotlin, these are just library functions.

A big improvement to the above code is taking advantage of the fact that the Activity itself can serve as the Coroutine Context for creating new Coroutines!

Our activity needs to implement the CoroutineScope interface and override the get method that returns the Coroutine context needed for calling the launch and async functions the can run our Coroutines. This way we don’t needed to call launch or async on the global scope. We also doesn’t need to specify the Coroutine Context and the Dispatchers since the get() method return the Context

And thus our code can be altered to this:

```
class MainActivity : AppCompatActivity(),CoroutineScope {

    private lateinit var binding: ActivityMainBinding
    private val TAG = MainActivity::class.java.name
    override val coroutineContext: CoroutineContext
        get() = Dispatchers.Main

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)
        binding = ActivityMainBinding.inflate(layoutInflater)
       launch { this: CoroutineScope
            fetchAndShowUser()
       }
    }

    private suspend fun fetchAndShowUser() {
        val user1 = fetchUserOneAsync()
        Log.d(TAG, msg: "user one fetched")
        val user2 = fetchUserTwoAsync()
        Log.d(TAG, msg: "user two fetched")
        showUser(user1.await(),user2.await())
    }

    private fun showUser(user1:User, user2:User) {
        binding.textViewOne.text = user1.toString()
        binding.textViewTwo.text = user2.toString()
    }

    private fun fetchUserOneAsync() = async(Dispatchers.IO) { this: CoroutineScope
        Thread.sleep( millis: 3000) //no checked exceptions on kotlin
        User( name: "Moshe",  email: "moshe@moshe.com")
    }
    private fun fetchUserTwoAsync() = async(Dispatchers.IO) { this: CoroutineScope
        Thread.sleep( millis: 3000) //no checked exceptions on kotlin
        User( name: "Misha",  email: "misha@misha.com")
    }
```

Again, when no Dispatchers is given to the launch or async coroutines it runs on the general context they are in.
So we don't have to specify to launch which scope and on which dispatcher to run in since it gets them from the activity.

Again one can think that all the coroutines launched from that scope will be auto cancelled when the activity is destroyed. But this is not the case. Just add this to the on create function. This code will create a coroutine that unless cancelled will last for a very long time. We will close our activity and see if the exception is thrown when the activity is destroyed and whether the coroutines ends with it:

```
launch {  this: CoroutineScope
    try {
        delay(Long.MAX_VALUE)
    } catch (e: Exception) {
        Log.d(TAG,  msg: "coroutine cancelled")
        // e will be a JobCancellationException if the activty is destroyed
    }
}
```

We kill the activity and the exception is not thrown! No one killed the coroutine when onDestroyed got called.

This can be done by bounding the scope to a specific Job. First lets understand what is a Job:

Job
A coroutine itself is represented by a Job. A Job is a handle to a coroutine. For every coroutine that you create (by launch or async - deferred is also a job - you can cancel it), it returns a Job instance that uniquely identifies the coroutine and manages its lifecycle. You can also pass a Job to a CoroutineScope to keep a handle on its lifecycle.
The coroutine scope is determined by an empty Job it create for himself if you won't pass any and using the + operator he add this job to his internal hash map and detains the scope lifecycle by it.

```
public fun CoroutineScope(context: CoroutineContext): CoroutineScope =
    ContextScope(if (context[Job] != null) context else context + Job())

internal class ContextScope(context: CoroutineContext) : CoroutineScope {
    override val coroutineContext: CoroutineContext = context
    // CoroutineScope is used intentionally for user-friendly representation
    override fun toString(): String = "CoroutineScope(coroutineContext=$coroutine
}
```

```
interface Deferred<out T> : Job
```

So we will create an empty job in the concrete and cancel is on the OnDestroy and add it to the get function like this:

```
private lateinit var job: Job

override val coroutineContext: CoroutineContext
    get() = job + Dispatchers.Main

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    binding = ActivityMainBinding.inflate(layoutInflater)
    setContentView(binding.root)

    job = Job()
```

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

No test your code again. Kill the activity. Is the exception thrown? Oh, ya...

We can create a lazy task that can be saved for later execution and will only be executed when needed, when we call start on the job returned.
This job will wait for the start function in oppose to regular launch call with default CoroutineStart parameter of DEFAULT and it start immediately

```
launch { this: CoroutineScope
    fetchAndShowUser()
}

val job = launch (start = CoroutineStart.LAZY){ this: CoroutineScope
    println("Task preformed")
}

job.start()
```

Another Coroutines useful suspended functions is:
joinAll() - that waits for all coroutines to return
Job.**join()** - called on specific Coroutine we want to wait for it to finish
repeat() - for repeated actions

delay(mills) which is much better than Thread.sleep because the later blocks

the whole thread while the former stops only the specific Coroutine and the others running on the same thread are not stopped.

Now we will see there is much more elegant solution for bounding our coroutines to the activity lifecycle that comes with the KTX-Extenssions kit.

More on Coroutines Scope

To avoid work leaks you should organize your coroutines by adding them to a CoroutineScope, which is an object that keeps track of coroutines. CoroutineScopes can be cancelled; and when you cancel a scope, they cancel all the associated coroutines. Above I'm using the GlobalScope, which is, as the name implies, a CoroutineScope that is available globally. It's generally **not** good practice to use the GlobalScope for the same reasons it's generally not good to write globally accessible variables. So you'll need to either make a scope, or get access to one.
In Activity or Fragment you can use the lifeCycleScope
In ViewModels, this is easy if you use viewModelScope.
And in LiveData you can use the liveDataScope

Add the following implementations (if needed) in your app Gradle file:

- For ViewModelScope, use androidx.lifecycle:lifecycle-viewmodel-ktx:2.4.1 or higher.
- For LifecycleScope, use androidx.lifecycle:lifecycle-runtime-ktx:2.4.1 or higher.
- For liveData, use androidx.lifecycle:lifecycle-livedata-ktx:2.4.1 or higher.

LifecycleScope

A LifecycleScope is defined for each Lifecycle object. Any coroutine launched in this scope is canceled when the Lifecycle is destroyed. You can access the CoroutineScope of the Lifecycle either via lifecycle.coroutineScope or lifecycleOwner.lifecycleScope properties. It is important to understand that the default Dispatchers of the lifeCycleScope is the Dispatchers.Main meaning the main thread of the application.

In an activity or fragment se the lifeCycleScope like this:

```
// launch in main/ui thread
// lifecylerOwner.lifecycleScope.launch {  }
lifecycleScope.launch { // Dispatchers.Main

}

// launch in background thread
lifecycleScope.launch(Dispatchers.Default) {

}
```

Even though the CoroutineScope provides a proper way to cancel long-running operations automatically, you might have other cases where you want to suspend execution of a code block unless the Lifecycle is in a certain state. For example, to run a FragmentTransaction, you must wait until the Lifecycle is at least STARTED. For these cases, Lifecycle provides additional methods: lifecycle.whenCreated, lifecycle.whenStarted, and lifecycle.whenResumed. Any coroutine run inside these blocks is suspended if the Lifecycle isn't at least in the minimal desired state.

```
class MyFragment : Fragment() {

    init {
        lifecycleScope.launch {   this: CoroutineScope
            whenCreated {  }
            whenStarted {  }
            whenResumed {  }
        }

    }

}
```

```
class MyFragment: Fragment {
    init { // Notice that we can safely launch in the constructor of the Fragment.
        lifecycleScope.launch {
            whenStarted {
                // The block inside will run only when Lifecycle is at least STARTE
                // It will start executing when fragment is started and
                // can call other suspend methods.
                loadingView.visibility = View.VISIBLE
                val canAccess = withContext(Dispatchers.IO) {
                    checkUserAccess()
                }

                // When checkUserAccess returns, the next line is automatically
                // suspended if the Lifecycle is not *at least* STARTED.
                // We could safely run fragment transactions because we know the
                // code won't run unless the lifecycle is at least STARTED.
                loadingView.visibility = View.GONE
                if (canAccess == false) {
                    findNavController().popBackStack()
                } else {
                    showContent()
                }
            }

            // This line runs only after the whenStarted block above has completed

        }
    }
}
```

And our revised code will look life this (check it, kill your activity and see the exception is thrown):

```
class MainActivity : AppCompatActivity() {

    private lateinit var binding: ActivityMainBinding
    private val TAG = MainActivity::class.java.name

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        binding = ActivityMainBinding.inflate(layoutInflater)
        setContentView(binding.root)

        // launch in main/ui thread
        // lifecylerOwner.lifecycleScope.launch {  } - No need for lifeCycleOwner cause it is this
        lifecycleScope.launch { this: CoroutineScope
            try {
                delay(Long.MAX_VALUE)
            } catch (e: Exception) {
                Log.d(TAG, msg: "coroutine cancelled")
                // e will be a JobCancellationException if the activty is destroyed
            }
        }

        lifecycleScope.launch { this: CoroutineScope
            fetchAndShowUser()
            delay( timeMillis: 700)
        }

        val job = lifecycleScope.launch (start = CoroutineStart.LAZY){ this: CoroutineScope
            println("Task preformed")
        }

        job.start()
    }
```

```
    private suspend fun fetchAndShowUser() {
        val user1 = fetchUserOneAsync()
        Log.d(TAG, msg: "user one fetched")
        val user2 = fetchUserTwoAsync()
        Log.d(TAG, msg: "user two fetched")
        showUser(user1.await(),user2.await())
    }

    private fun showUser(user1:User, user2:User) {
        binding.textViewOne.text = user1.toString()
        binding.textViewTwo.text = user2.toString()
    }

    private fun fetchUserOneAsync() = lifecycleScope.async(Dispatchers.IO) { this: CoroutineScope
        Thread.sleep( millis: 3000) //no checked exceptions on kotlin
        User( name: "Moshe", email: "moshe@moshe.com") ^async
    }

    private fun fetchUserTwoAsync() = lifecycleScope.async(Dispatchers.IO) { this: CoroutineScope
        Thread.sleep( millis: 3000) //no checked exceptions on kotlin
        User( name: "Misha", email: "misha@misha.com") ^async
    }

}

data class User(val name:String, val email:String)
```

viewModelScope

Often if your ViewModel is destroyed, there's a bunch of "work" associated with the ViewModel that should be stopped as well. For example, let's say you're preparing a bitmap to show on-screen. That's an example of work you should do without blocking the main thread *and* work that should be stopped if you permanently navigate away from or close the screen. For work like this, you should use viewModelScope.

viewModelScope is a Kotlin extension property on the ViewModel class. It is a CoroutineScope that is cancelled once the ViewModel is destroyed (when onCleared() is called). Thus when you're using a ViewModel, you can start all of your coroutines using this scope.

For ViewModelScope, use androidx.lifecycle:lifecycle-viewmodel-ktx:2.3.1

Here is an example:

```
class MyViewModel : ViewModel() {


    /**
     * Heavy operation that cannot be done in the Main Thread
     */
    fun launchDataLoad() {
        viewModelScope.launch {
            sortList()
            // Modify UI
        }
    }


    suspend fun sortList() = withContext(Dispatchers.Default) {
        // Heavy work
    }
}
```

Just think that the above code save all the code below:

```
class MyViewModel : ViewModel() {

    /**
     * This is the job for all coroutines started by this ViewModel.
     * Cancelling this job will cancel all coroutines started by this ViewModel.
     */
    private val viewModelJob = SupervisorJob()

    /**
     * This is the main scope for all coroutines launched by MainViewModel.
     * Since we pass viewModelJob, you can cancel all coroutines
     * launched by uiScope by calling viewModelJob.cancel()
     */
    private val uiScope = CoroutineScope(Dispatchers.Main + viewModelJob)

    /**
     * Cancel all coroutines when the ViewModel is cleared
     */
    override fun onCleared() {
        super.onCleared()
        viewModelJob.cancel()
    }

    /**
     * Heavy operation that cannot be done in the Main Thread
     */
    fun launchDataLoad() {
        uiScope.launch {
            sortList() // happens on the background
            // Modify UI
        }
    }

    // Move the execution off the main thread using withContext(Dispatchers.Default)
    suspend fun sortList() = withContext(Dispatchers.Default) {
        // Heavy work
    }
```

More on Coroutines And View Model
https://medium.com/androiddevelopers/easy-coroutines-in-android-viewmodelscope-25bffb605471
(Also available testing coroutines with mockito)
https://medium.com/androiddevelopers/viewmodels-with-saved-state-jetpack-navigation-data-binding-and-coroutines-df476b78144e

LiveData Scope special use cases of coroutines and Live Data

When using LiveData, you might need to calculate values asynchronously. For example, you might want to retrieve a user's preferences and serve them to your UI. In these cases, you can use the liveData builder function to call a suspend function, serving the result as a LiveData object.

```
val user: LiveData<User> = liveData {
    val data = database.loadUser() // loadUser is a suspend function.
    emit(data)
}
```

The code block starts executing when LiveData becomes active and is automatically canceled after a configurable timeout when the LiveData becomes inactive. If it is canceled before completion, it is restarted if the LiveData becomes active again. If it completed successfully in a previous run, it doesn't restart.

You can also emit multiple values from the block. **Each emit() call suspends the execution of the block until the LiveData value is set on the main thread.**

```
val user: LiveData<Result> = liveData {
    emit(Result.loading())
    try {
        emit(Result.success(fetchUser()))
    } catch(ioException: Exception) {
        emit(Result.error(ioException))
    }
}
```

You can emit multiple values from a LiveData by calling the emitSource() function whenever you want to emit a new value. Note that each call to emit() or emitSource() removes the previously-added source.

This means that you can use emit whenever you want to set a value once, but if you want to connect your live data to another live data value you use emit source.

```
liveData{
    emit(db.getData())
    val latest = webService.getLatestData()
    db.insert(latest)
    emit(db.getData())
}
```

But with emitSource it looks like this:

```
liveData{
    emitSource(db.getData())
    val latest = webService.getLatestData()
    db.insert(latest)
}
```

Don't need to call emit again since the liveData already have a source.

suspendCoroutine

Obtains the current continuation instance inside suspend functions and suspends the currently running coroutine.
This is usually done to prevent nesting of callbacks and use a single suspended function instead

```
suspend fun <T> awaitTask(task: Task<T>): T = suspendCoroutine { continuation ->
    task.addOnCompleteListener { task ->
        if (task.isSuccessful) {
            continuation.resume(task.result)
        } else {
            continuation.resumeWithException(task.exception!!)
        }
    }
}
```

When we use this code we can simply call awaitTask function and get the info result without any callback hassle from our side.
The block of code passed to suspendCoroutine { ... } should not block a thread that it is being invoked on, allowing the coroutine to be suspended. This way, the actual thread can be used for other tasks. This is a key feature that allows Kotlin coroutines to scale and to run multiple coroutines even on the single UI thread.