|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| <hitle> | Coroutine | <chare> | 1 | <pext> | Coroutines is a concurrency design pattern than simplify code by executing asynchronously.  Coroutines are NOT a replacement of threads.  Threads are limited in number but we can have as many coroutines as we want.  Threads are managed by the OS while as the coroutines are managed by the user. Multiple coroutines can run on a single thread,  such that one thread is utilized in a far more efficient manner.  -----------suspend function----------------  It is simply a function that can be paused and resumed at later time.  The syntax of a suspending function is similar to that of a regular function  except for the addition of the suspend keyword.  It can take a parameter and have a return type.  However, suspending functions can only be invoked  by another suspending function or within a coroutine scope.  -------------Coroutine Builders---------------  Coroutine Builders are:  Launch, Async, RunBlocking.  ------------Launch Coroutine Builder---------------  Launch: is a fire-and-forget type of coroutine that does not return any value.  -----------Async Coroutine Builder-------------  Async: async{} returns an instance of Deferred<T>,  which has an await()function that returns the result of the coroutine  like we have future in Java in which we do future.get() to the get the result.  ------------RunBlocking Coroutine Builder------------  RunBlocking is a coroutine function.  By not providing any context, it will get run on the main thread.  Runs a new coroutine and blocks the current thread interruptible  until its completion.  This function should not be used from a coroutine.  It is designed to bridge regular blocking code to libraries  that are written in suspending style, to be used in main functions  and in unit testing classes in Android development.  -----------withContext usecases-------------  1. Use withContext when you do not need the parallel execution.  2. Use async only when you need the parallel execution.  3. Both withContext and async can be used to get the result  which is not possible with the launch.  4. Use withContext to return the result of a single task.  5. Use async for results from multiple tasks that run in parallel.  ------------Coroutine Dispatchers-------------  Coroutine Dispatcher basically determines  which thread or thread pool the coroutine runs on.  So we don’t necessarily need to define a dispatcher if we don’t want to.  Coroutine Dispatchers are:  Main Dispatcher, Default Dispatcher, IO Dispatcher, Unconfined Dispatcher.  --------Coroutine Main Dispatcher---------------------  We will use the Main dispatcher when we want to update the UI  from liveData, or Presenter view method etc.  runBlocking {  launch(Dispatchers.Main) {  Log.d(TAG, "Main dispatcher. Thread: ${Thread.currentThread().name}")  }  }  -----------Coroutine Default Dispatcher-----------------  It is Useful for performing CPU intensive work like searching, sorting or filtering of data.  runBlocking {  launch(Dispatchers.Default) {  Log.i(TAG, "Default. Thread: ${Thread.currentThread().name}")  }  }  --------------Coroutine IO Dispatcher------------------  It is useful for network communication or reading/writing files,  Making API call or storing to Database.  runBlocking {  launch(Dispatchers.IO) {  Log.i(MainActivity.TAG, "IO. Thread: ${Thread.currentThread().name}")  }  }  ----------Coroutine Unconfined Dispatcher------------------  It starts the coroutine in the inherited dispatcher that called it.  launch(Dispatchers.Unconfined) {  Log.i(TAG, "Unconfined1. Thread: ${Thread.currentThread().name}")  delay(100L)  Log.i(TAG, "Unconfined2. Thread: ${Thread.currentThread().name}")  }  ------------Coroutine Scope-------------  In Kotlin, Coroutines must run in something called a CoroutineScope.  A CoroutineScope keeps track of coroutines, even coroutines that are suspended.  To ensure that all coroutines are tracked, Kotlin does not allow you to start  a new coroutine without a CoroutineScope.  Coroutine Scopes are:  Global Scope, LifeCycle Scope, ViewModel Scope.  -----------Coroutine Global Scope--------------  When Coroutines are launched within the global scope,  they live long as the application does.  Coroutines launched in the global scope will be launched in a separate thread.  GlobalScope.launch {  Log.d(TAG, Thread.currentThread().name.toString())  }  Log.d("Outside Global Scope", Thread.currentThread().name.toString())  ----------------Coroutine LifeCycle Scope---------------  The lifecycle scope is the same as the global scope,  but the only difference is that all the coroutines launched  within the activity also die when the activity dies.  // launching the coroutine in the lifecycle scope  lifecycleScope.launch {  while (true) {  delay(1000L)  Log.d(TAG, "Still Running..")  }  }  GlobalScope.launch {  delay(5000L)  val intent = Intent(this@MainActivity, SecondActivity::class.java)  startActivity(intent)  finish()  }  -----------Coroutine ViewModel Scope---------------  The coroutine in this scope will live as long the View Model is alive.  viewModelScope.launch {  //doSomething  }  -----------Coroutine Job--------------  Coroutine Jog is a Job can be used to wait for the coroutine to do some work  or it can be used to cancel it.  We can also pass a Job to a CoroutineScope to keep a handle on its lifecycle.  Coroutines can be controlled through the functions that are available  on the Job interface.  Coroutine Job has three states:  Start()  Join()  Cancel()  // A job is returned bu the builder block  val job = GlobalScope.launch(Dispatchers.Default) {  }  ----------Coroutine Job Cancelling--------------------  cancel() method is used to cancel the coroutine,  without waiting for it to finish its work.  fun playWithCoroutineJobCancellation(){  val job = GlobalScope.launch(Dispatchers.Default) {  repeat(5)  {  Log.d("TAG", "Coroutines is still working")  delay(1000)  }  }  runBlocking {  // delaying the coroutine by 2sec  delay(2000)  // canceling/stopping the coroutine  job.cancel()  Log.d("TAG", "Main Thread is Running")  }  }  ------------------Coroutine Exception Handling-------------  Exception handling can be done via 2 diffrent ways.  Way 1 : Using general try catch way to handle it but this way  is not really special for coroutines.  GlobalScope.launch(Dispatchers.Main) {  try {  doLogin() // do on IO thread and back to UI Thread  } catch (exception: Exception) {  Log.d(TAG, "$exception caught here !")  }  }  Way 2 : Using CoroutineExceptionHandler that is more preferred way  of handling exception in coroutines like below snippet.  //How to do declaration  val handler = CoroutineExceptionHandler { \_, exception ->  Log.d("TAG", "$exception caught here !")  error.postValue("$exception caught here !")  }  fun loadWeather() {  val job = viewModelScope.launch(Dispatchers.IO + handler) {  loading.postValue(true)  val response = repository.getRemoteData()  loading.postValue(false)  if (response.isSuccessful) {  response.body()?.let {  weatherLiveData.postValue(it)  }  }  }  } | </end> |
| <hitle> | async/await vs launch/join | <chare> | 2 | <pext> | 1. launch / join:-  The launch command is used to start and stop a coroutine.  It's as though a new thread has been started.  If the code inside the launch throws an exception,  it's considered as an uncaught exception in a thread,  which is typically written to stderr in backend JVM programs  and crashes Android applications.  Join is used to wait for the launched coroutine to complete  before propagating its exception.  A crashed child coroutine, on the other hand, cancels its parent  with the matching exception.  2. async / await:-  The async keyword is used to initiate a coroutine that computes a result.  You must use await on the result,  which is represented by an instance of Deferred.  Uncaught exceptions in async code are held in the resultant Deferred  and are not transmitted anywhere else.  They are not executed until processed. | </end> |
| <hitle> | thread vs coroutine | <chare> | 2 | <pext> | 1. A thread is an execution unit that can run independently from other threads.  1-1. A coroutine is a unit of execution that can be suspended and resumed,  allowing it to share resources with other concurrent or parallel executions.  2. Threads are typically heavier than coroutines,  so they can be more expensive in terms of performance.  However, this is not always the case,  and it really depends on the specific implementation.  2-1. In general, coroutines tend to be more efficient  when it comes to CPU usage, but threads may be better  when it comes to I/O bound tasks. | </end> |
| <hitle> | coroutine launch | <chare> | 2 | <pext> | Call the launch function on the CoroutineScope object to launch a coroutine,  passing in the function you wish to execute.  This will create a new coroutine and launch it immediately. | </end> |
| <hitle> | coroutine cancel | <chare> | 2 | <pext> | cancel() method is used to cancel the coroutine, without waiting for it  to finish its work.  It can be said that it is just opposite to that of the join method,  in the sense that join() method waits for the coroutine to finish  its whole work and block all other threads,  whereas the cancel() method when encountered, kills the coroutine  like stops the coroutine. | </end> |
| <hitle> | coroutine join | <chare> | 2 | <pext> | join() function is a suspending function, like it can be called from a coroutine  or from within another suspending function.  Job blocks all the threads until the coroutine in which it is written  or have context finished its work.  Only when the coroutine gets finishes, lines after the join() function  will be executed. | </end> |
| <hitle> | coroutine pass data | <chare> | 2 | <pext> | One of the best ways to pass data between coroutines in Kotlin  is with channel objects, which allow for safe and synchronized communication  between separate threads or processes.  To create a channel object, simply use the Channel constructor  and define any channels you want to send data through. | </end> |
| <hitle> | Job object kotlin | <chare> | 2 | <pext> | Job objects are the basic building blocks of coroutines.  They define a coroutine’s lifecycle and provide a way to cancel it.  CoroutineScope is used to define a scope for a coroutine,  which determines its lifetime and other properties. | </end> |
| <hitle> | exception in coroutine | <chare> | 2 | <pext> | If there’s an exception thrown inside a coroutine,  then the coroutine will be cancelled.  All the coroutine’s children will also be cancelled,  and any pending work in those coroutines will be lost. | </end> |
| <hitle> | withTimeOut() | <chare> | 2 | <pext> | Kotlin coroutine has come up with a good solution to the above problems  like, the coroutine will automatically be canceled and don’t do  any further calculations when a certain time has elapsed and  this is just withTimeOut().  There is no need to cancel the coroutine manually  by using runBlocking() function. | </end> |
| <hitle> | ensureActive, isActive | <chare> | 2 | <pext> | ensureActive ensures that current scope is active.  If the job is no longer active, throws CancellationException.  If the job was cancelled, thrown exception contains  the original cancellation cause.  This function does not do anything if there is no Job  in the scope's coroutineContext.  isActive returns true when the current Job is still active  or has not completed and was not cancelled yet. | </end> |
| <hitle> | Launch vs async | <chare> | 2 | <pext> | 1. launch{} cannot be used when we need the parallel execution  of network calls.  1-1. But we can use async only when we need the parallel execution  network calls.  2. launch{} will not block the main thread.  2-1. But async will block the main thread at the entry point  of the await() function.  3. For the launch, execution of other parts of the code will not wait  for the launch result since launch is not a suspend call.  3-1. But for the async, execution of the other parts of the code  will have to wait for the result of the await() function.  4. It is not possible for the launch to work like async in any case or condition.  4-1. Also we use async and do not wait for the result,  it will work exactly the same as launch.  5. Launch can be used at places if we don’t need the result  from the method called.  5-1 We can use async when we need the results from the multiple tasks  that run in parallel.  6. For Launch example, It can be used at places involving tasks  like update or changing color like fetch User And Save In Database.  6-1. For Async example, Imagine the condition, when we have to fetch  two users’ data from the database by using two parallel network calls  and then use them for computing some results based on their data. | </end> |
| <hitle> | Dispatchers.Main | <chare> | 2 | <pext> | The {Dispatchers.Main} expression is used to specify that a particular coroutine  should run on the main thread.  This is important because some operations can only be performed  on the main thread, and so specifying that a coroutine  should run on the main thread  ensures that it will be able to perform those operations. | </end> |
| <hitle> | Coroutine Features | <chare> | 2 | <pext> | **1. Lightweight**: We can run many coroutines on a single thread  due to support for [*suspension*](https://kotlinlang.org/docs/reference/coroutines/basics.html), which doesn't block the thread  where the coroutine is running.  Suspending saves memory over blocking while supporting  many concurrent operations.  **2. Fewer memory leaks**: We can use [*structured concurrency*](https://kotlinlang.org/docs/reference/coroutines/basics.html#structured-concurrency)   to run operations within a scope.  **3. Built-in cancellation support**: [Cancellation](https://kotlinlang.org/docs/reference/coroutines/cancellation-and-timeouts.html) is propagated automatically  through the running coroutine hierarchy.  **4. Jetpack integration**: Many Jetpack libraries include [extensions](https://developer.android.com/kotlin/ktx)   that provide full coroutines support. Some libraries also provide  their own [coroutine scope](https://developer.android.com/topic/libraries/architecture/coroutines) that we can use for structured concurrency. | </end> |
| <hitle> | corutine intro example | <chare> | 2 | <pext> | sealed class Result<out R> {     data class Success<out T>(val data: T) : Result<T>()     data class Error(val exception: Exception) : Result<Nothing>() }  class LoginRepository(private val responseParser: LoginResponseParser) {     private const val loginUrl = "https://example.com/login"      // Function that makes the network request, blocking the current thread     fun makeLoginRequest(         jsonBody: String     ): Result<LoginResponse> {         val url = URL(loginUrl)         (url.openConnection() as? HttpURLConnection)?.run {             requestMethod = "POST"             setRequestProperty("Content-Type", "application/json; utf-8")             setRequestProperty("Accept", "application/json")             doOutput = true             outputStream.write(jsonBody.toByteArray())             return Result.Success(responseParser.parse(inputStream))         }         return Result.Error(Exception("Cannot open HttpURLConnection"))     } }  class LoginViewModel(     private val loginRepository: LoginRepository ): ViewModel() {      fun login(username: String, token: String) {         val jsonBody = "{ username: \"$username\", token: \"$token\"}"         loginRepository.makeLoginRequest(jsonBody)     } }  makeLoginRequest is synchronous and blocks the calling thread.  The ViewModel triggers the network request  like when the user clicks on a login button.  With the previous code, LoginViewModel is blocking the UI thread  when making the network request.  The simplest solution to move the execution off the main thread  is to create a new coroutine and execute the network request  on an I/O thread, So we can do like this.  class LoginViewModel(     private val loginRepository: LoginRepository ): ViewModel() {      fun login(username: String, token: String) {         // Create a new coroutine to move the execution off the UI thread         viewModelScope.launch(Dispatchers.IO) {             val jsonBody = "{ username: \"$username\", token: \"$token\"}"             loginRepository.makeLoginRequest(jsonBody)         }     } }  The login function is executed as follows:  The app calls the login function from the View layer on the main thread.  launch creates a new coroutine, and the network request is made  independently on a thread reserved for I/O operations.  While the coroutine is running, the login function continues execution  and returns, possibly before the network request is finished. | </end> |
| <hitle> | coroutine lifecycle on viewModelScope | <chare> | 3 | <pext> | A coroutine started with viewModelScope, it is executed in the scope  of the ViewModel and if the ViewModel is destroyed  like the user is navigating away from that ViewModel’s screen,   viewModelScope is automatically cancelled, and all running coroutines  are canceled as well. | </end> |
| <hitle> | main-safety | <chare> | 3 | <pext> | We consider a function main-safe when it doesn't block UI updates  on the main thread.  class LoginRepository(private val responseParser: LoginResponseParser) {  private const val loginUrl = "https://example.com/login"  // Function that makes the network request, blocking the current thread  fun makeLoginRequest(  jsonBody: String  ): Result<LoginResponse> {  val url = URL(loginUrl)  (url.openConnection() as? HttpURLConnection)?.run {  requestMethod = "POST"  setRequestProperty("Content-Type", "application/json; utf-8")  setRequestProperty("Accept", "application/json")  doOutput = true  outputStream.write(jsonBody.toByteArray())  return Result.Success(responseParser.parse(inputStream))  }  return Result.Error(Exception("Cannot open HttpURLConnection"))  }  }  The makeLoginRequest function is not main-safe,  as calling makeLoginRequest from the main thread does block the UI.  So we can make like this using withContext() to make it main-safe.  class LoginRepository(...) {     ...     suspend fun makeLoginRequest(         jsonBody: String     ): Result<LoginResponse> {          // Move the execution of the coroutine to the I/O dispatcher         return withContext(Dispatchers.IO) {             // Blocking network request code         }     } } | </end> |
| <hitle> | move the execution of coroutine | <chare> | 3 | <pext> | We can use the withContext() function from the coroutines library  to move the execution of a coroutine to a different thread.  withContext(Dispatchers.IO) moves the execution  of the coroutine to an I/O thread, making our calling function main-safe  and enabling the UI to update as needed. | </end> |
| <hitle> | withContext() example | <chare> | 3 | <pext> | As makeLoginRequest moves the execution off the main thread,  the coroutine in the login function can be now executed  in the main thread:  class LoginViewModel(     private val loginRepository: LoginRepository ): ViewModel() {      fun login(username: String, token: String) {          // Create a new coroutine on the UI thread         viewModelScope.launch {             val jsonBody = "{ username: \"$username\", token: \"$token\"}"              // Make the network call and suspend execution until it finishes             val result = loginRepository.makeLoginRequest(jsonBody)              // Display result of the network request to the user             when (result) {                 is Result.Success<LoginResponse> -> // Happy path                 else -> // Show error in UI             }         }     } }  The coroutine is still needed here, since makeLoginRequest is  a suspend function, and all suspend functions must be executed  in a coroutine.  The login function now executes like this:  1. The app calls the login() function from the View layer  on the main thread.  2. launch creates a new coroutine on the main thread,  and the coroutine begins execution.  3. Within the coroutine,  the call to loginRepository.makeLoginRequest() now *suspends*   further execution of the coroutine until the withContext block  in makeLoginRequest() finishes running.  4. Once the withContext block finishes, the coroutine in login()   resumes execution *on the main thread* with the result  of the network request. | </end> |
| <hitle> | launch without Dispatcher | <chare> | 3 | <pext> | When we don't pass a Dispatcher to launch, any coroutines  launched from viewModelScope run in the main thread. | </end> |
| <hitle> | viewModelScope | <chare> | 3 | <pext> | viewModelScope is a predefined CoroutineScope   that is included with the ViewModel KTX extensions.  All coroutines must run in a scope.  A CoroutineScope manages one or more related coroutines. | </end> |
| <hitle> | launch | <chare> | 3 | <pext> | launch is a function that creates a coroutine and dispatches the execution  of its function body to the corresponding dispatcher. | </end> |
| <hitle> | Dispatchers.IO | <chare> | 3 | <pext> | Dispatchers.IO indicates that this coroutine should be executed  on a thread reserved for I/O operations. | </end> |
| <hitle> | advanced corutine example | <chare> | 2 | <pext> | 1. Manage long-running tasks  Coroutines build upon regular functions by adding two operations  to handle long-running tasks. In addition to invoke (or call) and return,  coroutines add suspend and resume:  1) suspend pauses the execution of the current coroutine,  saving all local variables.  2) resume continues execution of a suspended coroutine from the place  where it was suspended.  We can call suspend functions only from other suspend functions  or by using a coroutine builder such as launch to start a new coroutine.  suspend fun fetchDocs() { // Dispatchers.Main  val result = get("https://developer.android.com") // Dispatchers.IO for `get`  show(result) // Dispatchers.Main  }  suspend fun get(url: String) = withContext(Dispatchers.IO) { /\* ... \*/ }  get() still runs on the main thread, but it suspends  the coroutine before it starts the network request.  When the network request completes, get resumes the suspended coroutine  instead of using a callback to notify the main thread.  2. Use coroutines for main-safety  Let’s see the example.  suspend fun fetchDocs() { // Dispatchers.Main  val result = get("developer.android.com") // Dispatchers.Main  show(result) // Dispatchers.Main  }  suspend fun get(url: String) = // Dispatchers.Main  withContext(Dispatchers.IO) { // Dispatchers.IO (main-safety block)  /\* perform network IO here \*/ // Dispatchers.IO (main-safety block)  } // Dispatchers.Main  }  Inside the body of get, we can call withContext(Dispatchers.IO) to create a block that runs on the IO thread pool.  Any code we put inside that block always executes via the IO dispatcher.  Since withContext is itself a suspend function, the function get is also  a suspend function.  With coroutines, we can dispatch threads with fine-grained control.  Because withContext() lets we control the thread pool of any line of code  without introducing callbacks, we can apply it to very small functions  like reading from a database or performing a network request.  A good practice is to use withContext() to make sure every function  is main-safe, which means that we can call the function from the main thread.  This way, the caller never needs to think about which thread should be used  to execute the function.  In the example, fetchDocs() executes on the main thread;  however, it can safely call get, which performs a network request  in the background. Because coroutines support suspend and resume,  the coroutine on the main thread is resumed with the get result as soon as  the withContext block is done.  Important: Using suspend doesn't tell Kotlin to run a function  on a background thread. It's normal for suspend functions to operate  on the main thread. It's also common to launch coroutines  on the main thread. We should always use withContext()  inside a suspend function when we need main-safety,  such as when reading from or writing to disk, performing network operations,  or running CPU-intensive operations. | </end> |
| <hitle> | dispatchers | <chare> | 3 | <pext> | Kotlin coroutines use dispatchers to determine which threads are used  for coroutine execution. To run code outside of the main thread,  we can tell Kotlin coroutines to perform work on either the Default  or IO dispatcher. In Kotlin, all coroutines must run in a dispatcher,  even when they're running on the main thread.  Coroutines can suspend themselves, and the dispatcher is responsible  for resuming them.  1. Dispatchers.Main – we can use this dispatcher to run a coroutine  on the main Android thread. This should be used only for interacting  with the UI and performing quick work like, calling suspend functions,  running Android UI framework operations, and updating LiveData objects.  2. Dispatchers.IO - is optimized to perform disk or network I/O  outside of the main thread like, include using the Room component,  reading from or writing to files, and running any network operations.  3. Dispatchers.Default - is optimized to perform CPU-intensive work  outside of the main thread like, sorting a list and parsing JSON. | </end> |
| <hitle> | coroutine stack frame | <chare> | 3 | <pext> | Kotlin uses a stack frame to manage which function is running  along with any local variables. When suspending a coroutine,  the current stack frame is copied and saved for later. When resuming,  the stack frame is copied back from where it was saved,  and the function starts running again. Even though the code might look like  an ordinary sequential blocking request, the coroutine ensures  that the network request avoids blocking the main thread. | </end> |
| <hitle> | withContext() | <chare> | 3 | <pext> | A function that is very commonly used in coroutines is withContext.  Basically, withContext is allowed us to easily change context.  It means It easily switches between dispatchers.  Suppose in an android application we have some heavy processing  that we need to do.  So we will do that in the Default Dispatchers  then when the outcome completed then we want to switch  to the Main Dispatchers in order to publish to display outcome on the UI.  So we need a way to easily switch between context  and the best way to do that is the withContext function.  Because it’s very very lightweight.  launch(Dispatchers.Default) {  Log.d(TAG,"First context: $coroutineContext")  // default context  withContext(Dispatchers.IO) {  Log.d(TAG,"Second context: $coroutineContext")  // IO context  }  Log.d(TAG,"Third context: $coroutineContext")  // back to default context  } | </end> |
| <hitle> | Update UI from Coroutine | <chare> | 3 | <pext> | We can use withContext() to update UI from coroutine.  launch(Dispatchers.Default) {  Log.d(TAG,"First context: $coroutineContext")  // default context  withContext(Dispatchers.IO) {  Log.d(TAG,"Second context: $coroutineContext")  // IO context  }  Log.d(TAG,"Third context: $coroutineContext")  // back to default context  } | </end> |
| <hitle> | Performance of withContext() | <chare> | 3 | <pext> | withContext() does not add extra overhead compared to  an equivalent callback-based implementation.  Furthermore, it's possible to optimize withContext() calls beyond  an equivalent callback-based implementation in some situations.  For example, if a function makes ten calls to a network, you can tell Kotlin  to switch threads only once by using an outer withContext().  Then, even though the network library uses withContext() multiple times,  it stays on the same dispatcher and avoids switching threads.  In addition, Kotlin optimizes switching between Dispatchers.Default  and Dispatchers.IO to avoid thread switches whenever possible.  Important: Using a dispatcher that uses a thread pool like Dispatchers.IO  or Dispatchers.Default does not guarantee that the block executes  on the same thread from top to bottom.  In some situations, Kotlin coroutines might move execution to another thread  after a suspend-and-resume.  This means thread-local variables might not point to the same value  for the entire withContext() block. | </end> |
| <hitle> | Start/create coroutine | <chare> | 3 | <pext> | To start coroutine we can use launch or async.  1. launch: starts a new coroutine and doesn't return the result to the caller.  Any work that is considered "fire and forget" can be started using launch.  2. async: starts a new coroutine and allows you to return a result  with a suspend function called await.  Typically, we should launch a new coroutine from a regular function,  as a regular function cannot call await.  We can use async only when inside another coroutine  or when inside a suspend function and performing parallel decomposition. | </end> |
| <hitle> | exceptions launch/async | <chare> | 3 | <pext> | launch and async handle exceptions differently.  Since async expects an eventual call to await, it holds exceptions and  rethrows them as part of the await call.  This means if we use async to start a new coroutine from a regular function,  we might silently drop an exception. These dropped exceptions won't appear  in the crash metrics or be noted in logcat. | </end> |
| <hitle> | Parallel decomposition | <chare> | 3 | <pext> | All coroutines that are started inside a suspend function must be stopped  when that function returns, so you likely need to guarantee  that those coroutines finish before returning.  With structured concurrency in Kotlin, you can define a coroutineScope  that starts one or more coroutines, and then, using await() for a single coroutine, or using awaitAll() for multiple coroutines you can guarantee  that these coroutines finish before returning from the function.  If we define a coroutineScope that fetches two documents asynchronously,  by calling await() on each deferred reference, we guarantee  that both async operations finish before returning a value:  suspend fun fetchTwoDocs() =  coroutineScope {  val deferredOne = async { fetchDoc(1) }  val deferredTwo = async { fetchDoc(2) }  deferredOne.await()  deferredTwo.await()  }  using awaitAll()  suspend fun fetchTwoDocs() = // called on any Dispatcher (any thread, possibly Main)  coroutineScope {  val deferreds = listOf( // fetch two docs at the same time  async { fetchDoc(1) }, // async returns a result for the first doc  async { fetchDoc(2) } // async returns a result for the second doc  )  deferreds.awaitAll() // use awaitAll to wait for both network requests  }  Even though fetchTwoDocs() launches new coroutines with async,  the function uses awaitAll() to wait for those launched coroutines to finish  before returning.  However, that even if we had not called awaitAll(), the coroutineScope builder  does not resume the coroutine that called fetchTwoDocs  until after all of the new coroutines completed.  In addition, coroutineScope catches any exceptions that the coroutines throw  and routes them back to the caller. | </end> |
| <hitle> | CoroutineScope | <chare> | 3 | <pext> | A CoroutineScope keeps track of any coroutine it creates using launch or async.  The ongoing work like the running coroutines can be cancelled by calling  scope.cancel() at any point in time.  In Android, some KTX libraries provide their own CoroutineScope  for certain lifecycle classes.  For example, ViewModel has a viewModelScope, and Lifecycle has  lifecycleScope.  Unlike a dispatcher, a CoroutineScope doesn't run the coroutines. | </end> |
| <hitle> | CoroutineScope control lifecycle | <chare> | 3 | <pext> | If we need to create our own CoroutineScope to control the lifecycle of coroutines in a particular layer of the app, we can create it like this:  class ExampleClass {  // Job and Dispatcher are combined into a CoroutineContext which  // will be discussed shortly  val scope = CoroutineScope(Job() + Dispatchers.Main)  fun exampleMethod() {  // Starts a new coroutine within the scope  scope.launch {  // New coroutine that can call suspend functions  fetchDocs()  }  }  fun cleanUp() {  // Cancel the scope to cancel ongoing coroutines work  scope.cancel()  }  }  A cancelled scope cannot create more coroutines. Therefore, we should call  scope.cancel() only when the class that controls its lifecycle is being destroyed.  When using viewModelScope, the ViewModel class cancels the scope  automatically in the ViewModel's onCleared() method. | </end> |
| <hitle> | Job | <chare> | 3 | <pext> | A Job is a handle to a coroutine. Each coroutine that you create with launch  or async returns a Job instance that uniquely identifies the coroutine  and manages its lifecycle. You can also pass a Job to a CoroutineScope  to further manage its lifecycle.  class ExampleClass {  ...  fun exampleMethod() {  // Handle to the coroutine, you can control its lifecycle  val job = scope.launch {  // New coroutine  }  if (...) {  // Cancel the coroutine started above, this doesn't affect the scope  // this coroutine was launched in  job.cancel()  }  }  } | </end> |
| <hitle> | CoroutineContext | <chare> | 3 | <pext> | A CoroutineContext defines the behavior of a coroutine using a set of elements  like  1. Job: Controls the lifecycle of the coroutine.  2. CoroutineDispatcher: Dispatches work to the appropriate thread.  3. CoroutineName: The name of the coroutine, useful for debugging.  4. CoroutineExceptionHandler: Handles uncaught exceptions.  For new coroutines created within a scope, a new Job instance is assigned  to the new coroutine, and the other CoroutineContext elements are inherited  from the containing scope.  You can override the inherited elements by passing a new CoroutineContext  to the launch or async function.  Passing a Job to launch or async has no effect, as a new instance of Job  is always assigned to a new coroutine.  class ExampleClass {  val scope = CoroutineScope(Job() + Dispatchers.Main)  fun exampleMethod() {  // Starts a new coroutine on Dispatchers.Main as it's the scope's default  val job1 = scope.launch {  // New coroutine with CoroutineName = "coroutine" (default)  }  // Starts a new coroutine on Dispatchers.Default  val job2 = scope.launch(Dispatchers.Default + "BackgroundCoroutine") {  // New coroutine with CoroutineName = "BackgroundCoroutine" (overridden)  }  }  } | </end> |
| <hitle> | Unit testing coroutines | <chare> | 2 | <pext> | Make sure to add kotlinx.coroutines.test library as a test dependency. dependencies {  testImplementation "org.jetbrains.kotlinx:kotlinx-coroutines-test:$coroutines\_version"  }  1. How to Invoke suspending functions in tests?  To call suspending functions in tests, you need to be in a coroutine. As JUnit test functions themselves aren't suspending functions, you need to call a coroutine builder inside your tests to start a new coroutine.  2. We need to understand TestDispatchers  TestDispatchers are CoroutineDispatcher implementations for testing purposes. You’ll need to use TestDispatchers if new coroutines are created during the test to make the execution of the new coroutines predictable.  3. We may use StandardTestDispatcher.  When you start new coroutines on a StandardTestDispatcher, they are queued up on the underlying scheduler, to be run whenever the test thread is free to use. To let these new coroutines run, you need to yield the test thread (free it up for other coroutines to use). This queueing behavior gives you precise control over how new coroutines run during the test, and it resembles the scheduling of coroutines in production code.  4. And also we can use UnconfinedTestDispatcher.  When new coroutines are started on an UnconfinedTestDispatcher, they are started eagerly on the current thread. This means that they’ll start running immediately, without waiting for their coroutine builder to return. In many cases, this dispatching behavior results in simpler test code, as you don’t need to manually yield the test thread to let new coroutines run.  5. It’s good injecting test dispatchers.  Using dependency injection to provide dispatchers to your classes makes it easy to replace the real dispatchers in tests.  6. It’s important Setting the Main dispatcher  In local unit tests, the Main dispatcher that wraps the Android UI thread will be unavailable, as these tests are executed on a local JVM and not an Android device. If your code under test references the main thread, it’ll throw an exception during unit tests.  7. How to create dispatchers outside a test  In some cases, we might need a TestDispatcher to be available outside the test method. For example, during the initialization of a property in the test class: | </end> |
| <hitle> | test suspending function | <chare> | 3 | <pext> | To call suspending functions in tests, you need to be in a coroutine.  As JUnit test functions themselves aren't suspending functions,  we need to call a coroutine builder inside tests to start a new coroutine.  runTest is a coroutine builder designed for testing.  We can use this to wrap any tests that include coroutines.  Coroutines can be started not only directly in the test body,  but also by the objects being used in the test.  suspend fun fetchData(): String {  delay(1000L)  return "Hello world"  }  @Test  fun dataShouldBeHelloWorld() = runTest {  val data = fetchData()  assertEquals("Hello world", data)  }  Wrapping test’s code in runTest will work  for testing basic suspending functions, and it will automatically skip  any delays in coroutines, making the test above complete  much faster than one second. | </end> |
| <hitle> | runTest() | <chare> | 3 | <pext> | runTest is a coroutine builder designed for testing.  There are considerations to use runTest()  1. When your code creates new coroutines other than  the top-level test coroutine that runTest creates, you’ll need to control  how those new coroutines are scheduled by choosing  the appropriate TestDispatcher.  2. If your code moves the coroutine execution to other dispatchers  ,like by using withContext, runTest will still generally work, but delays  will no longer be skipped, and tests will be less predictable  as code runs on multiple threads. For these reasons, in tests we should inject  test dispatchers to replace real dispatchers. | </end> |
| <hitle> | TestDispatchers | <chare> | 3 | <pext> | TestDispatchers are CoroutineDispatcher implementations  for testing purposes. We need to use TestDispatchers if new coroutines  are created during the test to make the execution of the new coroutines  predictable.  There are two available implementations of TestDispatcher:  StandardTestDispatcher and UnconfinedTestDispatcher,  which perform different scheduling of newly-started coroutines.  These both use a TestCoroutineScheduler to control virtual time  and manage running coroutines within a test.  There should only be one scheduler instance used in a test, shared between all TestDispatchers. See Injecting TestDispatchers to learn about sharing schedulers.  To start the top-level test coroutine, runTest creates a TestScope, which is an implementation of CoroutineScope that will always use a TestDispatcher. If not specified, a TestScope will create a StandardTestDispatcher by default, and use that to run the top-level test coroutine.  runTest keeps track of the coroutines that are queued on the scheduler used by the dispatcher of its TestScope, and will not return as long as there’s pending work on that scheduler. | </end> |
| <hitle> | StandardTestDispatcher | <chare> | 3 | <pext> | When you start new coroutines on a StandardTestDispatcher, they are queued up on the underlying scheduler, to be run whenever the test thread is free to use. To let these new coroutines run, you need to yield the test thread (free it up for other coroutines to use). This queueing behavior gives you precise control over how new coroutines run during the test, and it resembles the scheduling of coroutines in production code. | </end> |
| <hitle> | yield test coroutine | <chare> | 3 | <pext> | There are several ways to yield the test coroutine to let queued-up coroutines run. All of these calls let other coroutines run on the test thread before returning:  1. advanceUntilIdle: Runs all other coroutines on the scheduler until there is nothing left in the queue. This is a good default choice to let all pending coroutines run, and it will work in most test scenarios.  2. advanceTimeBy: Advances virtual time by the given amount and runs any coroutines scheduled to run before that point in virtual time.  3. runCurrent: Runs coroutines that are scheduled at the current virtual time.  If the test thread is never yielded during the execution of the top-level test coroutine, any new coroutines will only run after the test coroutine is done (but before runTest returns):  @Test  fun standardTest() = runTest {  val userRepo = UserRepository()  launch { userRepo.register("Alice") }  launch { userRepo.register("Bob") }  assertEquals(listOf("Alice", "Bob"), userRepo.getAllUsers()) // ❌ Fails  }  To fix this, advanceUntilIdle can be used to let the two pending coroutines perform their work before continuing to the assertion:  @Test  fun standardTest() = runTest {  val userRepo = UserRepository()  launch { userRepo.register("Alice") }  launch { userRepo.register("Bob") }  advanceUntilIdle() // Yields to perform the registrations  assertEquals(listOf("Alice", "Bob"), userRepo.getAllUsers()) // ✅ Passes  } | </end> |
| <hitle> | UnconfinedTestDispatcher | <chare> | 3 | <pext> | When new coroutines are started on an UnconfinedTestDispatcher, they are started eagerly on the current thread. This means that they’ll start running immediately, without waiting for their coroutine builder to return. In many cases, this dispatching behavior results in simpler test code, as you don’t need to manually yield the test thread to let new coroutines run.  However, this behavior is different from what you’ll see in production with non-test dispatchers. If your test focuses on concurrency, prefer using StandardTestDispatcher instead.  Key Point: UnconfinedTestDispatcher executes new coroutines eagerly, and it can be a good choice for simple tests with coroutines.  To use this dispatcher for the top-level test coroutine in runTest instead of the default one, create an instance and pass it in as a parameter. This will make new coroutines created within runTest execute eagerly, as they inherit the dispatcher from the TestScope.  @Test  fun unconfinedTest() = runTest(UnconfinedTestDispatcher()) {  val userRepo = UserRepository()  launch { userRepo.register("Alice") }  launch { userRepo.register("Bob") }  assertEquals(listOf("Alice", "Bob"), userRepo.getAllUsers()) // ✅ Passes  }  In this example, the launch calls will start their new coroutines eagerly on the UnconfinedTestDispatcher, which means that each call to launch will only return after the registration is completed.  Remember that UnconfinedTestDispatcher starts new coroutines eagerly, but this doesn’t mean that it’ll run them to completion eagerly as well. If the new coroutine suspends, other coroutines will resume executing.  For example, the new coroutine launched within this test will register Alice, but then it suspends when delay is called. This lets the top-level coroutine proceed with the assertion, and the test fails as Bob is not registered yet:  @Test  fun yieldingTest() = runTest(UnconfinedTestDispatcher()) {  val userRepo = UserRepository()  launch {  userRepo.register("Alice")  delay(10L)  userRepo.register("Bob")  }  assertEquals(listOf("Alice", "Bob"), userRepo.getAllUsers()) // ❌ Fails  } | </end> |
| <hitle> | Injecting test dispatchers | <chare> | 3 | <pext> | Code under test might use dispatchers to switch threads (using withContext) or to start new coroutines. When code is executed on multiple threads in parallel, tests can become flaky. It can be difficult to perform assertions at the correct time or to wait for tasks to complete if they’re running on background threads that you have no control over.  In tests, replace these dispatchers with instances of TestDispatchers. This has several benefits:  1. The code will run on the single test thread, making tests more deterministic.  2. We can control how new coroutines are scheduled and executed.  3. TestDispatchers use a scheduler for virtual time,  which skips delays automatically and lets you advance time manually.  Using dependency injection to provide dispatchers to your classes makes it easy to replace the real dispatchers in tests.  Key Point: In tests, replace real dispatchers with instances of TestDispatchers to ensure that all code runs on the single test thread.  TestDispatchers will, by default, create a new scheduler when they’re instantiated. Inside runTest, you can access the testScheduler property of the TestScope and pass it in to any newly created TestDispatchers. This will share their understanding of virtual time, and methods like advanceUntilIdle will run coroutines on all test dispatchers to completion.  Let’s see an example,  // Example class demonstrating dispatcher use cases  class Repository(private val ioDispatcher: CoroutineDispatcher = Dispatchers.IO) {  private val scope = CoroutineScope(ioDispatcher)  val initialized = AtomicBoolean(false)  // A function that starts a new coroutine on the IO dispatcher  fun initialize() {  scope.launch {  initialized.set(true)  }  }  // A suspending function that switches to the IO dispatcher  suspend fun fetchData(): String = withContext(ioDispatcher) {  require(initialized.get()) { "Repository should be initialized first" }  delay(500L)  "Hello world"  }  }  you can see a Repository class that creates a new coroutine using the IO dispatcher in its initialize method and switches the caller to the IO dispatcher in its fetchData method.  In tests, you can inject a TestDispatcher implementation to replace the IO dispatcher.  Let’s see another example,  class RepositoryTest {  @Test  fun repoInitWorksAndDataIsHelloWorld() = runTest {  val dispatcher = StandardTestDispatcher(testScheduler)  val repository = Repository(dispatcher)  repository.initialize()  advanceUntilIdle() // Runs the new coroutine  assertEquals(true, repository.initialized.get())  val data = repository.fetchData() // No thread switch, delay is skipped  assertEquals("Hello world", data)  }  }  We inject a StandardTestDispatcher into the repository, and use advanceUntilIdle to make sure that the new coroutine started in initialize completes before proceeding.  New coroutines started on a TestDispatcher can be advanced manually with initialize. Note, however, that this would not be possible or desirable in production code. Instead, this method should be redesigned to be either suspending (for sequential execution), or to return a Deferred value (for concurrent execution).  For example, we can use async to start a new coroutine and create a Deferred:  class BetterRepository(private val ioDispatcher: CoroutineDispatcher = Dispatchers.IO) {  private val scope = CoroutineScope(ioDispatcher)  fun initialize() = scope.async {  // ...  }  }  This lets you safely await the completion of this code in both tests and production code:  @Test  fun repoInitWorks() = runTest {  val dispatcher = StandardTestDispatcher(testScheduler)  val repository = BetterRepository(dispatcher)  repository.initialize().await() // Suspends until the new coroutine is done  assertEquals(true, repository.initialized.get())  // ...  }  runTest will wait for pending coroutines to complete before returning if the coroutines are on a TestDispatcher that it shares a scheduler with. It will also wait for coroutines that are children of the top-level test coroutine, even if they’re on other dispatchers (up to a timeout specified by the dispatchTimeoutMs parameter, which is 60 seconds by default). | </end> |
| <hitle> | Setting the Main dispatcher | <chare> | 3 | <pext> | In local unit tests, the Main dispatcher that wraps the Android UI thread will be unavailable, as these tests are executed on a local JVM and not an Android device. If your code under test references the main thread, it’ll throw an exception during unit tests.  Note: This only applies to local unit tests. You should not replace the Main dispatcher in instrumented tests where the real UI thread is available.  In some cases, you can inject the Main dispatcher the same way as other dispatchers, as described in the previous section, allowing you to replace it with a TestDispatcher in tests. However, some APIs such as viewModelScope use a hardcoded Main dispatcher under the hood.  Here’s an example of a ViewModel implementation that uses viewModelScope to launch a coroutine that loads data:  class HomeViewModel : ViewModel() {  private val \_message = MutableStateFlow("")  val message: StateFlow<String> get() = \_message  fun loadMessage() {  viewModelScope.launch {  \_message.value = "Greetings!"  }  }  }  To replace the Main dispatcher with a TestDispatcher in all cases, use the Dispatchers.setMain and Dispatchers.resetMain functions.  class HomeViewModelTest {  @Test  fun settingMainDispatcher() = runTest {  val testDispatcher = UnconfinedTestDispatcher(testScheduler)  Dispatchers.setMain(testDispatcher)  try {  val viewModel = HomeViewModel()  viewModel.loadMessage() // Uses testDispatcher, runs its coroutine eagerly  assertEquals("Greetings!", viewModel.message.value)  } finally {  Dispatchers.resetMain()  }  }  }  If the Main dispatcher has been replaced with a TestDispatcher, any newly-created TestDispatchers will automatically use the scheduler from the Main dispatcher, including the StandardTestDispatcher created by runTest if no other dispatcher is passed to it.  This makes it easier to ensure that there is only a single scheduler in use during the test. For this to work, make sure to create all other TestDispatcher instances after calling Dispatchers.setMain.  Key Point: The Main dispatcher should be replaced by a TestDispatcher in local unit tests. Replacing the Main dispatcher also simplifies sharing schedulers between TestDispatchers.  A common pattern to avoid duplicating the code that replaces the Main dispatcher in each test is to extract it into a JUnit test rule:  // Reusable JUnit4 TestRule to override the Main dispatcher  class MainDispatcherRule(  val testDispatcher: TestDispatcher = UnconfinedTestDispatcher(),  ) : TestWatcher() {  override fun starting(description: Description) {  Dispatchers.setMain(testDispatcher)  }  override fun finished(description: Description) {  Dispatchers.resetMain()  }  }  class HomeViewModelTestUsingRule {  @get:Rule  val mainDispatcherRule = MainDispatcherRule()  @Test  fun settingMainDispatcher() = runTest { // Uses Main’s scheduler  val viewModel = HomeViewModel()  viewModel.loadMessage()  assertEquals("Greetings!", viewModel.message.value)  }  }  This rule implementation uses an UnconfinedTestDispatcher by default, but a StandardTestDispatcher can be passed in as a parameter if the Main dispatcher shouldn’t execute eagerly in a given test class.  When you need a TestDispatcher instance in the test body, you can reuse the testDispatcher from the rule, as long as it’s the desired type. If you want to be explicit about the type of TestDispatcher used in the test, or if you need a TestDispatcher that’s a different type than the one used for Main, you can create a new TestDispatcher within runTest. As the Main dispatcher is set to a TestDispatcher, any newly created TestDispatchers will share its scheduler automatically.  class DispatcherTypesTest {  @get:Rule  val mainDispatcherRule = MainDispatcherRule()  @Test  fun injectingTestDispatchers() = runTest { // Uses Main’s scheduler  // Use the UnconfinedTestDispatcher from the Main dispatcher  val unconfinedRepo = Repository(mainDispatcherRule.testDispatcher)  // Create a new StandardTestDispatcher (uses Main’s scheduler)  val standardRepo = Repository(StandardTestDispatcher())  }  } | </end> |
| <hitle> | Creating dispatchers outside a test | <chare> | 3 | <pext> | In some cases, you might need a TestDispatcher to be available outside the test method. For example, during the initialization of a property in the test class:  class Repository(private val ioDispatcher: CoroutineDispatcher) { /\* ... \*/ }  class RepositoryTestWithRule {  private val repository = Repository(/\* What TestDispatcher? \*/)  @get:Rule  val mainDispatcherRule = MainDispatcherRule()  @Test  fun someRepositoryTest() = runTest {  // Test the repository...  }  }  If you’re replacing the Main dispatcher as shown in the previous section, TestDispatchers created after the Main dispatcher has been replaced will automatically share its scheduler.  This isn’t the case, however, for TestDispatchers created as properties of the test class or TestDispatchers created during the initialization of properties in the test class. These are initialized before the Main dispatcher is replaced. Therefore, they would create new schedulers.  Caution: If you create a TestDispatcher as a property of a test class, it won’t take its scheduler from the Main dispatcher, as the Main dispatcher is only replaced before each test method is executed.  To make sure that there’s only one scheduler in your test, create the MainDispatcherRule property first. Then reuse its dispatcher (or its scheduler, if you need a TestDispatcher of a different type) in the initializers of other class-level properties as needed.  class RepositoryTestWithRule {  @get:Rule  val mainDispatcherRule = MainDispatcherRule()  private val repository = Repository(mainDispatcherRule.testDispatcher)  @Test  fun someRepositoryTest() = runTest { // Takes scheduler from Main  // Any TestDispatcher created here also takes the scheduler from Main  val newTestDispatcher = StandardTestDispatcher()  // Test the repository...  }  }  Note that both runTest and TestDispatchers created within the test will still automatically share the scheduler of the Main dispatcher.  If you’re not replacing the Main dispatcher, create your first TestDispatcher (which creates a new scheduler) as a property of the class. Then, manually pass that scheduler to each runTest invocation and each new TestDispatcher created, both as properties and within the test:  class RepositoryTest {  // Creates the single test scheduler  private val testDispatcher = UnconfinedTestDispatcher()  private val repository = Repository(testDispatcher)  @Test  fun someRepositoryTest() = runTest(testDispatcher.scheduler) {  // Take the scheduler from the TestScope  val newTestDispatcher = UnconfinedTestDispatcher(this.testScheduler)  // Or take the scheduler from the first dispatcher, they’re the same  val anotherTestDispatcher = UnconfinedTestDispatcher(testDispatcher.scheduler)  // Test the repository...  }  }  In this sample, the scheduler from the first dispatcher is passed to runTest. This will create a new StandardTestDispatcher for the TestScope using that scheduler. You could also pass in the dispatcher to runTest directly to run the test coroutine on that dispatcher. | </end> |
| <hitle> | Creating own TestScope | <chare> | 3 | <pext> | Like with TestDispatchers, you might need to access a TestScope outside the test body. While runTest creates a TestScope under the hood automatically, you can also create your own TestScope to use with runTest.  When doing this, make sure to call runTest on the TestScope you’ve created:  class SimpleExampleTest {  val testScope = TestScope() // Creates a StandardTestDispatcher  @Test  fun someTest() = testScope.runTest {  // ...  }  }  Caution: If you create your own TestScope, you must call runTest on that scope within your test. There can only be one TestScope instance in a test.  The code above creates a StandardTestDispatcher for the TestScope implicitly, as well as a new scheduler. These objects can all also be created explicitly. This can be useful if you need to integrate it with dependency injection setups.  class ExampleTest {  val testScheduler = TestCoroutineScheduler()  val testDispatcher = StandardTestDispatcher(testScheduler)  val testScope = TestScope(testDispatcher)  @Test  fun someTest() = testScope.runTest {  // ...  }  }  Key Point: You can create all the coroutine testing constructs outside of tests if your project setup requires it. | </end> |
| <hitle> | coroutines best practices | <chare> | 2 | <pext> | 1. We need to inject Dispatchers.  It’s not good to hardcode Dispatchers when creating new coroutines or calling withContext.  // DO inject Dispatchers  class NewsRepository(  private val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default  ) {  suspend fun loadNews() = withContext(defaultDispatcher) { /\* ... \*/ }  }  // DO NOT hardcode Dispatchers  class NewsRepository {  // DO NOT use Dispatchers.Default directly, inject it instead  suspend fun loadNews() = withContext(Dispatchers.Default) { /\* ... \*/ }  }  This dependency injection pattern makes testing easier as we can replace those dispatchers in unit and instrumentation tests with a test dispatcher to make your tests more deterministic.  2. Suspend functions should be safe to call from the main thread  Suspend functions should be main-safe, meaning they're safe to call from the main thread. If a class is doing long-running blocking operations in a coroutine, it's in charge of moving the execution off the main thread using withContext. This applies to all classes in your app, regardless of the part of the architecture the class is in.  class NewsRepository(private val ioDispatcher: CoroutineDispatcher) {  // As this operation is manually retrieving the news from the server  // using a blocking HttpURLConnection, it needs to move the execution  // to an IO dispatcher to make it main-safe  suspend fun fetchLatestNews(): List<Article> {  withContext(ioDispatcher) { /\* ... implementation ... \*/ }  }  }  // This use case fetches the latest news and the associated author.  class GetLatestNewsWithAuthorsUseCase(  private val newsRepository: NewsRepository,  private val authorsRepository: AuthorsRepository  ) {  // This method doesn't need to worry about moving the execution of the  // coroutine to a different thread as newsRepository is main-safe.  // The work done in the coroutine is lightweight as it only creates  // a list and add elements to it  suspend operator fun invoke(): List<ArticleWithAuthor> {  val news = newsRepository.fetchLatestNews()  val response: List<ArticleWithAuthor> = mutableEmptyList()  for (article in news) {  val author = authorsRepository.getAuthor(article.author)  response.add(ArticleWithAuthor(article, author))  }  return Result.Success(response)  }  }  This pattern makes your app more scalable, as classes calling suspend functions don't have to worry about what Dispatcher to use for what type of work. This responsibility lies in the class that does the work.  3. The ViewModel should create coroutines  ViewModel classes should prefer creating coroutines instead of exposing suspend functions to perform business logic. Suspend functions in the ViewModel can be useful if instead of exposing state using a stream of data, only a single value needs to be emitted.  // DO create coroutines in the ViewModel  class LatestNewsViewModel(  private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase  ) : ViewModel() {  private val \_uiState = MutableStateFlow<LatestNewsUiState>(LatestNewsUiState.Loading)  val uiState: StateFlow<LatestNewsUiState> = \_uiState  fun loadNews() {  viewModelScope.launch {  val latestNewsWithAuthors = getLatestNewsWithAuthors()  \_uiState.value = LatestNewsUiState.Success(latestNewsWithAuthors)  }  }  }  // Prefer observable state rather than suspend functions from the ViewModel  class LatestNewsViewModel(  private val getLatestNewsWithAuthors: GetLatestNewsWithAuthorsUseCase  ) : ViewModel() {  // DO NOT do this. News would probably need to be refreshed as well.  // Instead of exposing a single value with a suspend function, news should  // be exposed using a stream of data as in the code snippet above.  suspend fun loadNews() = getLatestNewsWithAuthors()  }  Views shouldn't directly trigger any coroutines to perform business logic. Instead, defer that responsibility to the ViewModel. This makes your business logic easier to test as ViewModel objects can be unit tested, instead of using instrumentation tests that are required to test views.  In addition to that, your coroutines will survive configuration changes automatically if the work is started in the viewModelScope. If you create coroutines using lifecycleScope instead, you'd have to handle that manually. If the coroutine needs to outlive the ViewModel's scope, check out the Creating coroutines in the business and data layer section.  4. Don't expose mutable types  Prefer exposing immutable types to other classes. In this way, all changes to the mutable type is centralized in one class making it easier to debug when something goes wrong.  // DO expose immutable types  class LatestNewsViewModel : ViewModel() {  private val \_uiState = MutableStateFlow(LatestNewsUiState.Loading)  val uiState: StateFlow<LatestNewsUiState> = \_uiState  /\* ... \*/  }  class LatestNewsViewModel : ViewModel() {  // DO NOT expose mutable types  val uiState = MutableStateFlow(LatestNewsUiState.Loading)  /\* ... \*/  }  5. The data and business layer should expose suspend functions and Flows  Classes in the data and business layers generally expose functions to perform one-shot calls or to be notified of data changes over time. Classes in those layers should expose suspend functions for one-shot calls and Flow to notify about data changes.  // Classes in the data and business layer expose  // either suspend functions or Flows  class ExampleRepository {  suspend fun makeNetworkRequest() { /\* ... \*/ }  fun getExamples(): Flow<Example> { /\* ... \*/ }  }  This best practice makes the caller, generally the presentation layer, able to control the execution and lifecycle of the work happening in those layers, and cancel when needed.  How to creating coroutines in the business and data layer?  For classes in the data or business layer that need to create coroutines for different reasons, there are different options.  If the work to be done in those coroutines are relevant only when the user is present on the current screen, it should follow the caller's lifecycle. In most cases, the caller will be the ViewModel. In this case, coroutineScope or supervisorScope should be used.  class GetAllBooksAndAuthorsUseCase(  private val booksRepository: BooksRepository,  private val authorsRepository: AuthorsRepository,  private val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default  ) {  suspend fun getBookAndAuthors(): BookAndAuthors {  // In parallel, fetch books and authors and return when both requests  // complete and the data is ready  return coroutineScope {  val books = async(defaultDispatcher) {  booksRepository.getAllBooks()  }  val authors = async(defaultDispatcher) {  authorsRepository.getAllAuthors()  }  BookAndAuthors(books.await(), authors.await())  }  }  }  If the work to be done is relevant as long as the app is opened, and the work is not bound to a particular screen, then the work should outlive the caller's lifecycle. For this scenario, an external CoroutineScope should be used as explained in the Coroutines & Patterns for work that shouldn’t be cancelled blog post.  class ArticlesRepository(  private val articlesDataSource: ArticlesDataSource,  private val externalScope: CoroutineScope,  private val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default  ) {  // As we want to complete bookmarking the article even if the user moves  // away from the screen, the work is done creating a new coroutine  // from an external scope  suspend fun bookmarkArticle(article: Article) {  externalScope.launch(defaultDispatcher) {  articlesDataSource.bookmarkArticle(article)  }  .join() // Wait for the coroutine to complete  }  }  externalScope should be created and managed by a class that lives longer than the current screen, it could be managed by the Application class or a ViewModel scoped to a navigation graph.  7. Inject TestDispatchers in tests  An instance of TestDispatcher should be injected into your classes in tests. There are two available implementations in the kotlinx-coroutines-test library:  StandardTestDispatcher: Queues up coroutines started on it with a scheduler, and executes them when the test thread is not busy. You can suspend the test thread to let other queued coroutines run using methods such as advanceUntilIdle.  UnconfinedTestDispatcher: Runs new coroutines eagerly, in a blocking way. This generally makes writing tests easier, but gives you less control over how coroutines are executed during the test.  See the documentation of each dispatcher implementation for additional details.  To test coroutines, use the runTest coroutine builder. runTest uses a TestCoroutineScheduler to skip delays in tests and to allow you to control virtual time. You can also use this scheduler to create additional test dispatchers as needed.  class ArticlesRepositoryTest {  @Test  fun testBookmarkArticle() = runTest {  // Pass the testScheduler provided by runTest's coroutine scope to  // the test dispatcher  val testDispatcher = UnconfinedTestDispatcher(testScheduler)  val articlesDataSource = FakeArticlesDataSource()  val repository = ArticlesRepository(  articlesDataSource,  testDispatcher  )  val article = Article()  repository.bookmarkArticle(article)  assertThat(articlesDataSource.isBookmarked(article)).isTrue()  }  }  All TestDispatchers should share the same scheduler. This allows you to run all your coroutine code on the single test thread to make your tests deterministic. runTest will wait for all coroutines that are on the same scheduler or are children of the test coroutine to complete before returning.  8. Avoid GlobalScope  This is similar to the Inject Dispatchers best practice. By using GlobalScope, you're hardcoding the CoroutineScope that a class uses bringing some downsides with it:  Promotes hard-coding values. If you hardcode GlobalScope, you might be hard-coding Dispatchers as well.  Makes testing very hard as your code is executed in an uncontrolled scope, you won't be able to control its execution.  You can't have a common CoroutineContext to execute for all coroutines built into the scope itself.  Instead, consider injecting a CoroutineScope for work that needs to outlive the current scope. Check out the Creating coroutines in the business and data layer section to learn more about this topic.  // DO inject an external scope instead of using GlobalScope.  // GlobalScope can be used indirectly. Here as a default parameter makes sense.  class ArticlesRepository(  private val articlesDataSource: ArticlesDataSource,  private val externalScope: CoroutineScope = GlobalScope,  private val defaultDispatcher: CoroutineDispatcher = Dispatchers.Default  ) {  // As we want to complete bookmarking the article even if the user moves  // away from the screen, the work is done creating a new coroutine  // from an external scope  suspend fun bookmarkArticle(article: Article) {  externalScope.launch(defaultDispatcher) {  articlesDataSource.bookmarkArticle(article)  }  .join() // Wait for the coroutine to complete  }  }  // DO NOT use GlobalScope directly  class ArticlesRepository(  private val articlesDataSource: ArticlesDataSource,  ) {  // As we want to complete bookmarking the article even if the user moves away  // from the screen, the work is done creating a new coroutine with GlobalScope  suspend fun bookmarkArticle(article: Article) {  GlobalScope.launch {  articlesDataSource.bookmarkArticle(article)  }  .join() // Wait for the coroutine to complete  }  }  Learn more about GlobalScope and its alternatives in the Coroutines & Patterns for work that shouldn’t be cancelled blog post.  9. Make your coroutine cancellable  Cancellation in coroutines is cooperative, which means that when a coroutine's Job is cancelled, the coroutine isn't cancelled until it suspends or checks for cancellation. If you do blocking operations in a coroutine, make sure that the coroutine is cancellable.  For example, if you're reading multiple files from disk, before you start reading each file, check whether the coroutine was cancelled. One way to check for cancellation is by calling the ensureActive function.  someScope.launch {  for(file in files) {  ensureActive() // Check for cancellation  readFile(file)  }  }  All suspend functions from kotlinx.coroutines such as withContext and delay are cancellable. If your coroutine calls them, you shouldn't need to do any additional work.  For more information about cancellation in coroutines, check out the Cancellation in coroutines blog post.  10. Watch out for exceptions  Unhandled exceptions thrown in coroutines can make your app crash. If exceptions are likely to happen, catch them in the body of any coroutines created with viewModelScope or lifecycleScope.  class LoginViewModel(  private val loginRepository: LoginRepository  ) : ViewModel() {  fun login(username: String, token: String) {  viewModelScope.launch {  try {  loginRepository.login(username, token)  // Notify view user logged in successfully  } catch (exception: IOException) {  // Notify view login attempt failed  }  }  }  } | </end> |
| <hitle> | Coroutine memory leaks | <chare> | 2 | <pext> | When we use Global Scope, it still can have memory leaks. | </end> |
| <hitle> | coroutine exception handler | <chare> | 2 | <pext> | If we do not handle exceptions in current coroutine,  it will be canceled and also the parent corutines are canceled.  supervisorScope is used not to cancel parents coroutines  in case we met coroutine exception. | </end> |
| <hitle> | coroutineScope vs supervisorScope | <chare> | 2 | <pext> | 1. A coroutineScope will cancel whenever any of its children fail.  1-2. A supervisorScope won't cancel other children when one of them fails.  2. We can use coroutineScope with the top-level try-catch,  when you do NOT want to continue with other tasks  if any of them have failed.  2-1. If we want to continue with the other tasks even when one fails,  we go with the supervisorScope.  We can use supervisorScope with the individual try-catch for each task,  when we want to continue with other tasks  if one or some of them have failed. | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> | 2 | <pext> |  | </end> |
| <hitle> |  | <chare> |  |  |  |  |
| <hitle> |  | <chare> |  |  |  |  |
| <hitle> |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  | <chare> |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |