

1. Introduction

In this codelab you'll learn how to use [Kotlin Coroutines](#) in an Android app—a new way of managing background threads that can simplify code by reducing the need for callbacks. Coroutines are a Kotlin feature that convert async callbacks for long-running tasks, such as database or network access, into *sequential* code.

Here is a code snippet to give you an idea of what you'll be doing.

```
// Async callbacks
networkRequest { result ->
    // Successful network request
    databaseSave(result) { rows ->
        // Result saved
    }
}
```

The callback-based code will be converted to sequential code using coroutines.

```
// The same code with coroutines
val result = networkRequest()
// Successful network request
databaseSave(result)
// Result saved
```

You will start with an existing app, built using [Architecture Components](#), that uses a callback style for long-running tasks.

By the end of this codelab you will have enough experience to convert an existing API to use coroutines, and you will be able to integrate coroutines into an app. You'll also be familiar with best practices for coroutines, and how to write a test against code that uses coroutines.

What you'll learn

- How to call code written with coroutines and obtain results.
- How to use suspend functions to make async code sequential.
- How to use `launch` and `runBlocking` to control how code executes.
- Techniques to convert existing APIs to coroutines using `suspendCoroutine`.
- How to use coroutines with Architecture Components.
- Best practices for testing coroutines.

Prerequisites

- Experienced with the Architecture Components `ViewModel`, `LiveData`, `Repository`, and `Room`.
- Experienced with Kotlin syntax, including extension functions and lambdas.
- A basic understanding of using threads on Android, including the main thread, background threads, and callbacks.

- For an introduction to the Architecture Components used in this codelab, see [Room with a View](#).
- For an introduction to Kotlin syntax, see [Kotlin Bootcamp for Programmers](#).
- For an introduction to threading basics on Android, see [Guide to background processing](#).



What you'll need

- [Android Studio 3.3](#) (the codelab may work with other versions, but some things might be missing or look different).

2. Getting set up

Download the code




Click the following link to download all the code for this codelab:

[Download Zip](#)

... or clone the GitHub repository from the command line by using the following command:

```
$ git clone https://github.com/googlecodelabs/kotlin-coroutines.git
```

The **kotlin-coroutines** repository contains three different app projects:

-  **kotlin-coroutines-start** — Simple app to explore making your first coroutine
-  **kotlin-coroutines-repository** — Project based on callbacks that you will convert to use coroutines
-  **kotlin-coroutines-end** — The project with coroutines already added


Frequently asked questions

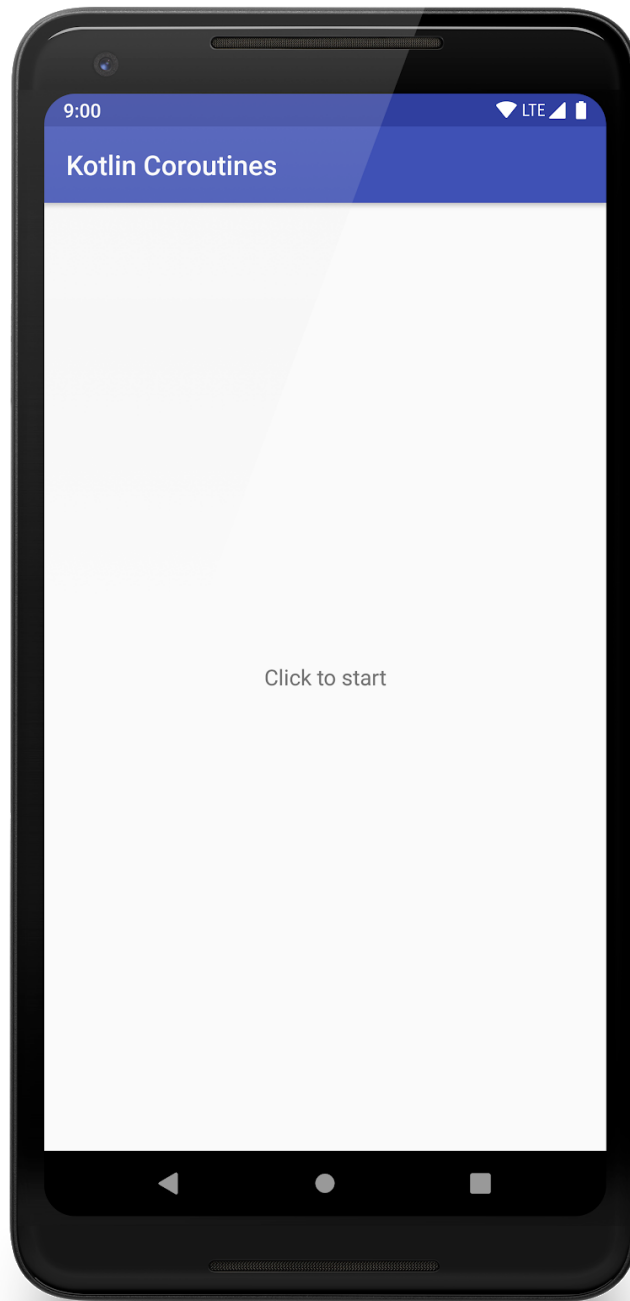
[How do I install Android Studio?](#)

[How do I set up a device for development?](#)

3. Run the starting sample app

First, let's see what the starting sample app looks like. Follow these instructions to open the sample app in Android Studio.

1. If you downloaded the `kotlin-coroutines` zip file, unzip the file.
2. Open the `kotlin-coroutines-start` project in Android Studio.
3. Click the  **Run** button, and either choose an emulator or connect your Android device, which must be capable of running Android Lollipop (the minimum SDK supported is 21). The Kotlin Coroutines screen should appear:



This starter app uses threads to show a [Snackbar](#) one second after you press anywhere on the screen. Give it a try now, and you should see *"Hello, from threads!"* after a short delay. In the first part of this codelab you'll convert this application to use coroutines.

This app uses Architecture Components to separate the UI code in `MainActivity` from the application logic in `MainViewModel`. Take a moment to familiarize yourself with the structure of the project.

1. `MainActivity` displays the UI, registers click listeners, and can display a `Snackbar`. It passes events to `MainViewModel` and updates the screen based on `LiveData` in `MainViewModel`.
2. `MainViewModel` handles events in `onMainViewClicked` and will communicate to `MainActivity` using `LiveData`.
3. `Executors` defines `BACKGROUND`, which can run things on a background thread.
4. `MainViewModelTest` defines a test for `MainViewModel`.

Adding coroutines to a project

To use coroutines in Kotlin, you must include the `coroutines-core` library in the `build.gradle (Module: app)` file of your project. The codelab projects have already done this for you, so you don't need to do this to complete the codelab.

Coroutines on Android are available as a core library, and Android specific extensions:

- **kotlinx-coroutines-core** — Main interface for using coroutines in Kotlin
- **kotlinx-coroutines-android** — Support for the Android Main thread in coroutines

The starter app already includes the dependencies in `build.gradle`. When creating a new app project, you'll need to open `build.gradle (Module: app)` and add the coroutines dependencies to the project.

```
dependencies {  
    ...  
    implementation "org.jetbrains.kotlin:kotlinx-coroutines-core:x.x.x"  
    implementation "org.jetbrains.kotlin:kotlinx-coroutines-android:x.x.x"  
}
```

Coroutines and RxJava

If you're using [RxJava](#) in your current codebase, you can use coroutines with RxJava by using a [kotlin-coroutines-rx](#) library.

4. Coroutines in Kotlin

On Android, it's essential to avoid blocking the main thread. The main thread is a single thread that handles all updates to the UI. It's also the thread that calls all click handlers and other UI callbacks. As such, it has to run smoothly to guarantee a great user experience.

For your app to display to the user without any visible pauses, the main thread has to update the screen [every 16ms or more often](#), which is about 60 frames per second. Many common tasks take longer than this, such as parsing large JSON datasets, writing data to a database, or fetching data from the network. Therefore, calling code like this from the main thread can cause the app to pause, stutter, or even freeze. And if you block the main thread for too long, the app may even crash and present an **Application Not Responding** dialog.

Watch the video below for an introduction to how coroutines solves this problem for us on Android by introducing main-safety.

The callback pattern

One pattern for performing long-running tasks without blocking the main thread is callbacks. By using callbacks, you can start long-running tasks on a background thread. When the task completes, the callback is called to inform you of the result on the main thread.

Take a look at an example of the callback pattern.

```
// Slow request with callbacks
@UiThread
fun makeNetworkRequest() {
    // The slow network request runs on another thread
    slowFetch { result ->
        // When the result is ready, this callback will get the result
        show(result)
    }
    // makeNetworkRequest() exits after calling slowFetch without waiting for the result
}
```

Because this code is annotated with [@UiThread](#), it must run fast enough to execute on the main thread. That means, it needs to return very quickly, so that the next screen update is not delayed. However, since `slowFetch` will take seconds or even minutes to complete, the main thread can't wait for the result. The `show(result)` callback allows `slowFetch` to run on a background thread and return the result when it's ready.

Using coroutines to remove callbacks

Callbacks are a great pattern, however they have a few drawbacks. Code that heavily uses callbacks can become hard to read and harder to reason about. In addition, callbacks don't allow the use of some language features, such as exceptions.

Kotlin coroutines let you convert callback-based code to sequential code. Code written sequentially is typically easier to read, and can even use language features such as exceptions.

In the end, they do the exact same thing: wait until a result is available from a long-running task and continue execution. However, in code they look very different.

The keyword `suspend` is Kotlin's way of marking a function, or function type, available to coroutines. When a coroutine calls a function marked `suspend`, instead of blocking until that function returns like a normal function call, it **suspends** execution until the result is ready then it **resumes** where it left off with the result. While it's suspended waiting for a result, it **unblocks the thread that it's running on** so other functions or coroutines can run.

For example in the code below, `makeNetworkRequest()` and `slowFetch()` are both suspend functions.

```
// Slow request with coroutines
@UiThread
suspend fun makeNetworkRequest() {
    // slowFetch is another suspend function so instead of
    // blocking the main thread makeNetworkRequest will `suspend` until the result is
    // ready
    val result = slowFetch()
    // continue to execute after the result is ready
    show(result)
}

// slowFetch is main-safe using coroutines
suspend fun slowFetch(): SlowResult { ... }
```

Just like with the callback version, `makeNetworkRequest` must return from the main thread right away because it's marked `@UiThread`. This means that usually it could not call blocking methods like `slowFetch`. Here's where the `suspend` keyword works its magic.

Important: The `suspend` keyword doesn't specify the thread code runs on. Suspend functions may run on a background thread or the main thread.

Compared to callback-based code, coroutine code accomplishes the same result of unblocking the current thread with less code. Due to its sequential style, it's easy to chain several long running tasks without creating multiple callbacks. For example, code that fetches a result from two network endpoints and saves it to the database can be written as a function in coroutines with no callbacks. Like so:

```
// Request data from network and save it to database with coroutines

// Because of the @WorkerThread, this function cannot be called on the
// main thread without causing an error.
@WorkerThread
suspend fun makeNetworkRequest() {
    // slowFetch and anotherFetch are suspend functions
    val slow = slowFetch()
    val another = anotherFetch()
    // save is a regular function and will block this thread
    database.save(slow, another)
}

// slowFetch is main-safe using coroutines
suspend fun slowFetch(): SlowResult { ... }
// anotherFetch is main-safe using coroutines
suspend fun anotherFetch(): AnotherResult { ... }
```

Coroutines by another name

The pattern of `async` and `await` in other languages is based on coroutines. If you're familiar with this pattern, the `suspend` keyword is similar to `async`. However in Kotlin, `await()` is implicit when calling a suspend function.

Kotlin has a method `Deferred.await()` that is used to wait for the result from a coroutine started with the `async` builder.

You will convert the start sample app to use coroutines in the next section.

5. Controlling the UI with coroutines

In this exercise you will write a coroutine to display a message after a delay. To get started, make sure you have the project `kotlin-coroutines-start` open in Android Studio.

Add a coroutine scope to MainViewModel

In Kotlin, all coroutines run inside a [CoroutineScope](#). A scope controls the lifetime of coroutines through its job. When you cancel the job of a scope, it cancels all coroutines started in that scope. On Android, you can use a scope to cancel all running coroutines when, for example, the user navigates away from an Activity or Fragment. Scopes also allow you to specify a default dispatcher. A dispatcher controls which thread runs a coroutine.

To start coroutines in `MainViewModel.kt`, the scope would be created like this:

```
private val viewModelJob = Job()

private val uiScope = CoroutineScope(Dispatchers.Main + viewModelJob)
```

In our example, `uiScope` will start coroutines in `Dispatchers.Main` which is the main thread on Android. A coroutine started on the main won't block the main thread while suspended. Since a `ViewModel` coroutine almost always updates the UI on the main thread, starting coroutines on the main thread is a reasonable default. As we'll see later in this codelab, a coroutine can switch dispatchers any time after it's started. For example, a coroutine can start on the main dispatcher then use another dispatcher to parse a large JSON result off the main thread.

CoroutineContext

A `CoroutineScope` can take a [CoroutineContext](#) as a parameter. The `CoroutineContext` is a set of attributes that configures the coroutine. It can define the threading policy, exception handler, etc.

In the example above, we're using the `CoroutineContext` plus operator to define the threading policy ([Dispatchers.Main](#)) and the job (`viewModelJob`). The resulting `CoroutineContext` is a combination of both contexts.

Cancel the scope when ViewModel is cleared

`onCleared` is called when the `ViewModel` is no longer used and will be destroyed. This typically happens when the user navigates away from the Activity or Fragment that was using the `ViewModel`. If we wanted to cancel the scope that we saw in the previous section, you'd have to include the following code.

```
override fun onCleared() {
    super.onCleared()
    viewModelJob.cancel()
}
```

Since `viewModelJob` is passed as the job to `uiScope`, when `viewModelJob` is cancelled every coroutine started by `uiScope` will be cancelled as well. It's important to cancel any coroutines that are no longer required to avoid unnecessary work and memory leaks.

Important: You must pass `CoroutineScope` a `Job` in order to cancel all coroutines started in the scope. If you don't, the scope will run until your app is terminated. If that's not what you intended, you will be leaking memory.

Scopes created with the `CoroutineScope` constructor add an implicit job, which you can cancel using `uiScope.coroutineContext.cancel()`

Use viewModelScope to avoid boilerplate code

We could include the code above in every `ViewModel` we have in our project to have a scope bound to it. However, that would be a lot of boilerplate code. That's why we use the `AndroidX lifecycle-viewmodel-ktx` library. To use this library, you must include it in the `build.gradle (Module: app)` file of your project. That step is already done in the codelab projects.

```
dependencies {
    ...
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:x.x.x"
}
```

The library adds a `viewModelScope` as an extension function of the `ViewModel` class. This scope is bound to `Dispatchers.Main` and will automatically be cancelled when the `ViewModel` is cleared. Instead of having to create a new scope in every `ViewModel`, you can just use `viewModelScope` and the library will take care of setting up and clearing the scope for you.

This is how you can use the `viewModelScope` to launch a coroutine that will make a network request in a background thread.

```
class MainViewModel : ViewModel() {
    // Make a network request without blocking the UI thread
    private fun makeNetworkRequest() {
        // launch a coroutine in viewModelScope
        viewModelScope.launch(Dispatchers.IO) {
            // slowFetch()
        }
    }

    // No need to override onCleared()
}
```

Switch from threads to coroutines

In `MainViewModel.kt` find the next `TODO` along with this code:


```
/**
 * Wait one second then display a snackbar.
 */
fun onMainViewClicked() {
    // TODO: Replace with coroutine implementation
    BACKGROUND.submit {
        Thread.sleep(1_000)
        // use postValue since we're in a background thread
        _snackBar.postValue("Hello, from threads!")
    }
}
```

This code uses `BACKGROUND` to run in a background thread. Since `sleep` blocks the current thread it would freeze the UI if it were called on the main thread. One second after the user clicks the main view, it requests a snackbar.

Replace `onMainViewClicked` with this coroutine based code that does the same thing. You will have to import `launch` and `delay`.

```
/**
 * Wait one second then display a snackbar.
 */
fun onMainViewClicked() {
    // launch a coroutine in viewModelScope
    viewModelScope.launch {
        // suspend this coroutine for one second
        delay(1_000)
        // resume in the main dispatcher
        // _snackbar.value can be called directly from main thread
        _snackBar.value = "Hello, from coroutines!"
    }
}
```

This code does the same thing, waiting one second before showing a snackbar. However, there are some important differences:

1. `viewModelScope.launch` will start a coroutine in the `viewModelScope`. This means when the job that we passed to `viewModelScope` gets canceled, all coroutines in this job/scope will be cancelled. If the user left the Activity before `delay` returned, this coroutine will automatically be cancelled when `onCleared` is called upon destruction of the `ViewModel`.
2. Since `viewModelScope` has a default dispatcher of `Dispatchers.Main`, this coroutine will be launched in the main thread. We'll see later how to use different threads.
3. The function `delay` is a `suspend` function. This is shown in Android Studio by the  icon in the left gutter. Even though this coroutine runs on the main thread, `delay` won't block the thread for one second. Instead, the dispatcher will schedule the coroutine to resume in one second at the next statement.

Go ahead and run it. When you click on the main view you should see a snackbar one second later.

In the next section we'll consider how to test this function.

6. Testing coroutines through behavior

In this exercise you'll write a test for the code you just wrote. This exercise shows you how to test coroutines in the same style as tests written for code using threads. Later in this codelab you'll implement a test that interacts with coroutines directly.

The [kotlinx-coroutines-test](#) library was recently launched that provides many utilities to simplify testing coroutines on Android. The library currently has a status of `@ExperimentalCoroutinesApi` and may change before final release.

The library provides a way to set `Dispatchers.Main` for running tests off-device, as well as a testing dispatcher that allows test code to control the execution of coroutines.

It offers you the ability to:

1. Auto-advance of time for regular suspend functions
2. Explicitly control time for testing multiple coroutines
3. Eagerly execute the bodies of `launch` or `async` code blocks
4. Pause, manually advance, and restart the execution of coroutines in a test
5. Report uncaught exceptions as test failures

To learn more, read the documentation for [kotlinx-coroutines-test](#).

Since the library is currently marked experimental, this codelab will show you how to write tests using stable APIs until it's stable.

At the end of this section, you can find the test code rewritten using `kotlinx-coroutines-test`.

Review the existing test

Open `MainViewModelTest.kt` in the `androidTest` folder.

```
@RunWith(JUnit4::class)
class MainViewModelTest {

    /**
     * In this test, LiveData will immediately post values without switching threads.
     */
    @get:Rule
    val instantTaskExecutorRule = InstantTaskExecutorRule()

    lateinit var subject: MainViewModel

    /**
     * Before the test runs initialize subject
     */
    @Before
    fun setup() {
        subject = MainViewModel()
    }
}
```

Two things happen before each test:

1. A rule is a way to run code before and after the execution of a test in JUnit. `InstantTaskExecutorRule` is a JUnit rule that configures LiveData to immediately post to the main thread while a test is run.
2. During `setup()` of the test the field `subject` is initialized to a new `MainViewModel`.

After this test setup, one test is defined:



```
@Test
fun whenMainViewModelClicked_showSnackbar() {
    runBlocking {
        subject.snackbar.captureValues {
            subject.onMainViewClicked()
            assertSendsValues(2_000, "Hello, from threads!")
        }
    }
}
```

This test calls `onMainViewClicked` then waits for a snackbar using the test helper `assertSendsValues` which waits for up to two seconds for a value to be sent to a `LiveData`. You won't need to read the function to complete this codelab.

This test only depends on the public API of `ViewModel`: When `onMainViewClicked` is called, *"Hello, from threads!"* will be sent to the snackbar.

We didn't change the public API, the method call still updates the snackbar, so changing the implementation to use coroutines won't break the test.

Run the existing test

1. Right click on the class name `MainViewModelTest` in your editor to open a context menu.
2. In the context menu choose  **Run 'MainViewModelTest'**
3. For future runs you can select this test configuration in the configurations next to the  button in the toolbar. By default, the configuration will be called **MainViewModelTest**.

When you run the test you will get an assertion failure if you implemented the previous exercise

```
expected: Hello, from threads!
but was : Hello, from coroutines!
```

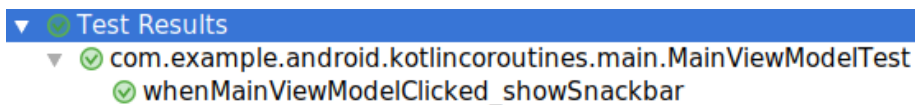
Update failing test to pass

This test is failing because we changed the behavior of our method. Instead of saying "Hello, from threads!" it says "Hello, from coroutines!"

Update the test to the new behavior by changing the assertion.

```
@Test
fun whenMainViewModelClicked_showSnackbar() {
    runBlocking {
        subject.snackbar.captureValues {
            subject.onMainViewClicked()
            assertSendsValues(2_000, "Hello, from coroutines!")
        }
    }
}
```

When you run the test again using the  in the toolbar, the test will pass



By testing only the public API, we were able to change our test from a background thread to a coroutine without any changes to the structure of our test.

In the next exercise you'll learn how to convert existing callback APIs to work with coroutines.

What about that delay?

This test still has one major problem. It takes an entire second to run! This is because the call to `delay(1_000)` is hardcoded in `onMainViewClicked`.

Tests should run as fast as possible, and this one can definitely run faster.

The `TestCoroutineDispatcher` provided by `kotlinx-coroutines-test` allows you to control "virtual time" and call a function with a one second delay without actually waiting for a second.

Here's the above test rewritten to use the experimental [TestCoroutineDispatcher](https://kotlinlang.org/api/latest/jvm/stdlib/kotlinx.coroutines-test/test-coroutine-dispatcher/).

```
/**
 * Example of the same test written using the experimental test
 * kotlinx-coroutines-test API.
 */

@RunWith(JUnit4::class)
class MainViewModelTest {

    /**
     * In this test, LiveData will immediately post values without switching threads.
     */
    @get:Rule
    val instantTaskExecutorRule = InstantTaskExecutorRule()

    /**
     * This dispatcher will let us progress time in the test.
     */
    var testDispatcher = TestCoroutineDispatcher()

    lateinit var subject: MainViewModel

    /**
     * Before the test runs initialize subject
     */
    @Before
    fun setup() {
        // set Dispatchers.Main, this allows our test to run
        // off-device
        Dispatchers.setMain(testDispatcher)
        subject = MainViewModel()
    }

    @After
    fun teardown() {
        // reset main after the test is done
        Dispatchers.resetMain()
        // call this to ensure TestCoroutineDispatcher doesn't
    }
}
```

```
// accidentally carry state to the next test
dispatcher.cleanupTestCoroutines()
}

// note the use of runBlockingTest instead of runBlocking
// this gives the test the ability to control time.
@Test
fun whenMainViewModelClicked_showSnackBar() = testDispatcher.runBlockingTest {
    subject.snackbar.observeForTesting {
        subject.onMainViewClicked()
        // progress time by one second
        advanceTimeBy(1_000)
        // value is available immediately without making the test wait
        Truth.assertThat(subject.snackbar.value)
            .isEqualTo("Hello, from coroutines!")
    }
}

// helper method to allow us to get the value from a LiveData
// LiveData won't publish a result until there is at least one observer
private fun <T> LiveData<T>.observeForTesting(
    block: () -> Unit) {
    val observer = Observer<T> { Unit }
    try {
        observeForever(observer)
        block()
    } finally {
        removeObserver(observer)
    }
}
}
```

7. Converting existing callback APIs with coroutines

In this exercise you'll convert an existing callback-based API to use coroutines.

To get started open the project `kotlin-coroutines-repository` in Android Studio.

This app uses Architecture Components and extends the previous project to implement a data layer that uses both a network and a local database. When the main view is clicked, it fetches a new title from the network, saves it to the database, and displays it on screen. Take a moment to familiarize yourself with the new classes.

1. `MainDatabase` implements a database using Room that saves and loads a `Title`.
2. `MainNetwork` implements a network API that fetches a new title. It uses a fake network library defined in `FakeNetworkLibrary.kt` to fetch titles. The network library will randomly return errors.
3. `TitleRepository` implements a single API for fetching or refreshing the title by combining data from the network and database.
4. `MainViewModelTest` defines a test for `MainViewModel`.
5. `FakeNetworkCallAwaitTest` is a test we will complete later in this codelab.

Explore the existing callback API

Open `MainNetwork.kt` to see the declaration of `fetchNewWelcome()`

```
// MainNetwork.kt

fun fetchNewWelcome(): FakeNetworkCall<String>
```

Open `TitleRepository.kt` to see how `fetchNewWelcome` is used to make a network call using the callback pattern.

This function returns a `FakeNetworkCall`, which allows callers to register a listener for this network request. Calling `fetchNewWelcome` will start a long running network request on another thread and return an object that exposes `addOnResultListener`. Your code passes a callback to `addOnResultListener` which will be called when the request completes or errors.

```
// TitleRepository.kt

fun refreshTitle(/* ... */) {
    val call = network.fetchNewWelcome()
    call.addOnResultListener { result ->
        // callback called when network request completes or errors
        when (result) {
            is FakeNetworkSuccess<String> -> {
                // process successful result
            }
            is FakeNetworkError -> {
                // process network error
            }
        }
    }
}
```

Convert the existing callback API to a suspend function

The function `refreshTitle` is currently implemented using callbacks on `FakeNetworkCall`. The goal of this exercise is to expose our network API as a suspend function so that `refreshTitle` can be rewritten as a coroutine.

To do that Kotlin provides a function `suspendCoroutine` that's used to convert callback-based APIs to suspend functions.

Calling `suspendCoroutine` will immediately suspend the current coroutine. `suspendCoroutine` will give you a `continuation` object that you can use to resume the coroutine. A `continuation` does what it sounds like: it holds all the context needed to continue, or resume, a suspended coroutine.

The `continuation` that `suspendCoroutine` provides has two functions: `resume` and `resumeWithException`. Calling either function will cause `suspendCoroutine` to resume immediately.

You can use `suspendCoroutine` to suspend before waiting for a callback. Then, after the callback is called call `resume` or `resumeWithException` to resume with the result of the callback.

An example of `suspendCoroutine` looks like this:

```
// Example of suspendCoroutine

/**
 * A class that passes strings to callbacks
 */
class Call {
    fun addCallback(callback: (String) -> Unit)
}

/**
 * Exposes callback based API as a suspend function so it can be used in coroutines.
 */
suspend fun convertToSuspend(call: Call): String {
    // 1: suspendCoroutine and will immediately *suspend*
    // the coroutine. It can be only *resumed* by the
    // continuation object passed to the block.
    return suspendCoroutine { continuation ->
        // 2: pass a block to suspendCoroutine to register callbacks

        // 3: add a callback to wait for the result
        call.addCallback { value ->
            // 4: use continuation.resume to *resume* the coroutine
            // with the value. The value passed to resume will be
            // the result of suspendCoroutine.
            continuation.resume(value)
        }
    }
}
```

This example shows how to use `suspendCoroutine` to convert a callback-based API on `Call` into a suspend function. You can now use `Call` directly in coroutine based code, for example

```
// Example of using convertToSuspend to use a callback API in coroutines

suspend fun exampleUsage() {
    val call = makeLongRunningCall()
    convertToSuspend(call) // suspends until the long running call completes
}
```

You can use this pattern to expose a suspend function on `FakeNetworkCall` that allows you to use the callback-based network API in coroutines.

What about cancellation?

[suspendCoroutine](#) is a good choice when you don't need to support cancellation. Typically, however, cancellation is a concern and you can use [suspendCancellableCoroutine](#) to propagate cancellation to libraries that support cancellation to a callback based API.

Use suspendCoroutine to convert a callback API to coroutines

Scroll to the bottom of `TitleRepository.kt` and find the TODO to implement an extension function.

```
/**
 * Suspend function to use callback-based [FakeNetworkCall] in coroutines
 *
 * @return network result after completion
 * @throws Throwable original exception from library if network request fails
 */
// TODO: Implement FakeNetworkCall<T>.await() here
```

Replace that TODO with this extension function on `FakeNetworkCall<T>`

```
suspend fun <T> FakeNetworkCall<T>.await(): T {
    return suspendCoroutine { continuation ->
        addOnResultListener { result ->
            when (result) {
                is FakeNetworkSuccess<T> -> continuation.resume(result.data)
                is FakeNetworkError -> continuation.resumeWithException(result.error)
            }
        }
    }
}
```

This extension function uses `suspendCoroutine` to convert a callback based API to a suspend function. Coroutines can call `await` and will immediately suspend until the network result is ready. The network result is the return value of `await`, and errors will throw an exception.

You can use it like this:

```
// Example usage of await

suspend fun exampleAwaitUsage() {
    try {
        val call = network.fetchNewWelcome()
        // suspend until fetchNewWelcome returns a result or throws an error
        val result = call.await()
        // resume will cause await to return the network result
    } catch (error: FakeNetworkException) {
        // resumeWithException will cause await to throw the error
    }
}
```

It's worth taking a second to read the function signature for `await`. The `suspend` keyword tells Kotlin this is available to coroutines. As a result it can call other suspend functions such as `suspendCoroutine`. The rest of the declaration, `fun <T> FakeNetworkCall<T>.await()`, defines an extension function called `await` on

any `FakeNetworkCall` . It doesn't actually modify the class, but when calling from Kotlin it appears as a public method. The return type of `await` is `T` which is specified after the function name.

What is an extension function?

If you're new to Kotlin extension functions may be a new concept. Extension functions don't modify the class, instead they introduce a new function that takes `this` as the first argument.

```
fun <T> await(this: FakeNetworkCall<T>): T
```

Inside the body of `await` function, `this` is bound to the `FakeNetworkCall<T>` passed. That's how `await` calls `addOnResultListener` . It uses the implicit `this` just like a member method.

All together, this signature means we're adding a `suspend` function, called `await()` , to a class that was not originally built for coroutines. This approach can be used to update a callback based API to support coroutines without changing the implementation.

In the next exercise you'll write tests for `await()` and learn how to call coroutines directly from tests.

8. Testing coroutines directly

In this exercise you'll write a test that calls a `suspend` function directly.

Since `await` is exposed as a public API it will be tested directly, showing how to call coroutines functions from tests.

Here's the `await` function you implemented in the last exercise:

```
// TitleRepository.kt

suspend fun <T> FakeNetworkCall<T>.await(): T {
    return suspendCoroutine { continuation ->
        addOnResultListener { result ->
            when (result) {
                is FakeNetworkSuccess<T> -> continuation.resume(result.data)
                is FakeNetworkError -> continuation.resumeWithException(result.error)
            }
        }
    }
}
```

Write a test that calls a suspend function

Open `FakeNetworkCallAwaitTest.kt` in the `androidTest` folder which has two `TODOS`.

Try to call `await` from the second test `whenFakeNetworkCallFailure_throws`.

```
@Test(expected = FakeNetworkException::class)
fun whenFakeNetworkCallFailure_throws() {
    val subject = makeFailureCall(FakeNetworkException("the error"))

    subject.await() // Compiler error: Can't call outside of coroutine
}
```

Since `await` is a `suspend` function Kotlin doesn't know how to call it except from a coroutine or another `suspend` function, and you will get a compiler error like, *"Suspend function 'await' should be called only from a coroutine or another suspend function."*

The test runner doesn't know anything about coroutines so we can't make this test a `suspend` function. We could launch a coroutine using a `CoroutineScope` like in a `ViewModel`, however tests need to run coroutines to completion before they return. Once a test function returns, the test is over. Coroutines started with `launch` are asynchronous code, which will complete at some point in the future. Therefore to test that asynchronous code, you need some way to tell the test to wait until your coroutine completes. Since `launch` is a non-blocking call, that means it returns right away and can continue to run a coroutine after the function returns, it can't be used in tests. For example:

```
// Example of launch in a test (always fails)

@Test(expected = FakeNetworkException::class)
fun whenFakeNetworkCallFailure_throws() {
    val subject = makeFailureCall(FakeNetworkException("the error"))

    // launch starts the coroutine and then returns immediately
    GlobalScope.launch {
        // since this is asynchronous code, this may be called *after* the test completes
        subject.await()
    }
}
```

```

    }
    // test function returns immediately, and
    // doesn't get the exception raised by await()
}

```

This test will **always** fail. The call to `launch` will return immediately and end the test case. The exception from `await()` may happen before or after the test ends, but the exception will not be thrown in the test call stack. It will instead be thrown into `scope`'s uncaught exception handler.

Kotlin has the `runBlocking` function that blocks while it calls suspend functions. When `runBlocking` calls a suspend function, instead of suspending like normal it will block just like a function. You can think of it as a way to convert suspend functions into normal function calls.

Since `runBlocking` runs coroutines just like a normal function, it will also throw exceptions just like a normal function.

Important: The function `runBlocking` will always block the caller, just like a regular function call. The coroutine will run synchronously on the same thread. You should avoid `runBlocking` in your application code and prefer `launch` which returns immediately.

`runBlocking` should only be used from APIs that expect blocking calls like tests.

The recently launched [kotlinx-coroutines-test](#) library provides `runBlockingTest` which replaces the use of `runBlocking` for tests.

Since the library is currently marked experimental, this codelab will show you how to write tests using stable APIs until it's stable.

If you include that library you should use `runBlockingTest` in the test code here wherever `runBlocking` is used.

Wrap the call to `await` with `runBlocking`.

```

@Test(expected = FakeNetworkException::class)
fun whenFakeNetworkCallFailure_throws() {
    val subject = makeFailureCall(FakeNetworkException("the error"))

    runBlocking {
        subject.await()
    }
}

```

And implement the first test using `runBlocking` as well.

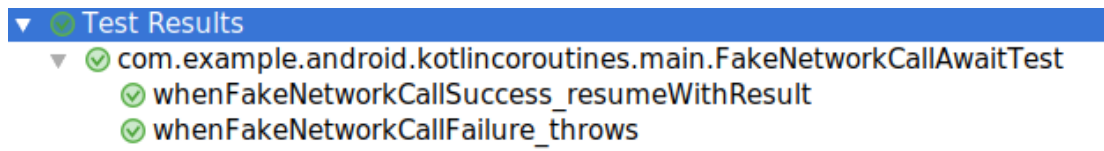
```

@Test
fun whenFakeNetworkCallSuccess_resumeWithResult() {
    val subject = makeSuccessCall("the title")

    runBlocking {
        Truth.assertThat(subject.await()).isEqualTo("the title")
    }
}

```

Run the test. When you run the tests you should see all tests pass!



In the next exercise you'll learn how to use coroutines to fetch data in a `Repository` and `ViewModel` .

9. Using coroutines on a worker thread

In this exercise you'll learn how to switch the thread a coroutine runs on in order to finish implementing `TitleRepository`.

Review the existing callback code in `refreshTitle`

Open `TitleRepository.kt` and review the existing callback-based implementation.

```
// TitleRepository.kt

fun refreshTitle(onStateChanged: TitleStateListener) {
    // 1: Communicate the network request is starting
    onStateChanged>Loading)
    val call = network.fetchNewWelcome()
    // 2: Register a callback to get notified when the network result completes or errors
    call.addOnResultListener { result ->
        when (result) {
            is FakeNetworkSuccess<String> -> {
                // 3: Save the new title on a background thread
                BACKGROUND.submit {
                    // run insertTitle on a background thread
                    titleDao.insertTitle>Title(result.data))
                }
                // 4: Tell the caller the request is successful
                onStateChanged>Success)
            }
            is FakeNetworkError -> {
                // 5: tell the caller the request errored
                onStateChanged(
                    Error>TitleRefreshError(result.error)))
            }
        }
    }
}
```

In `TitleRepository.kt` the method `refreshTitle` is implemented with a callback to communicate the loading and error state to the caller. With two callbacks working together, the code can be hard to read. Here's what's going on:

1. Before the request starts, the callback is notified that the request is `Loading`.
2. To wait for the network result, another callback is registered on the `FakeNetworkCall`.
3. When a new title is returned by the network, it is saved to the database on a background thread.
4. The caller is told that the request has been completed (it is no longer `Loading`).
5. If the network request fails, the caller is told that the request has errored (again it is no longer `Loading`).

Open `MainViewModel.kt` to see how this API is used to control the UI

```
// MainViewModel.kt

fun refreshTitle() {
    // pass a state listener as a lambda to refreshTitle
    repository.refreshTitle { state ->
        when (state) {
            is Loading -> _spinner.postValue(true)
            is Success -> _spinner.postValue(false)
        }
    }
}
```

```

is Error -> {
    _spinner.postValue(false)
    _snackBar.postValue(state.error.message)
}
}
}
}
}

```

On the side calling `refreshTitle`, the code is less complicated. It passes a callback to `repository.refreshTitle` that is called repeatedly with either `Loading`, `Success`, or `Error`. In each case, the UI is updated with the appropriate `LiveData`.

Replace callback code with coroutines in TitleRepository

Open `TitleRepository.kt` and replace `refreshTitle` with a coroutine based implementation. This code doesn't use `RefreshState` or `TitleStateListener`, so you can delete them now.

```

// TitleRepository.kt

suspend fun refreshTitle() {
    withContext(Dispatchers.IO) {
        try {
            val result = network.fetchNewWelcome().await()
            titleDao.insertTitle(Title(result))
        } catch (error: FakeNetworkException) {
            throw TitleRefreshError(error)
        }
    }
}

// delete class RefreshState and typealias TitleStateListener

```

This code uses the `await` function defined earlier to convert `fetchNewWelcome` to a `suspend` function. Since `await` returns the value of the network request as the result when it resumes, it can be assigned directly to `result` without creating a callback. If the network request errors, `await` will throw an exception (because it called `resumeWithException`) and the error is available in a regular try/catch block.

The function `withContext` is used to ensure the database insert is executed on a background thread. This is important because `insertTitle` is a blocking function. Even though it's running in a coroutine, it will block the thread the coroutine runs on until it returns. If you call `insertTitle` from the main thread, even in a coroutine, it will cause your app to freeze during the database write.

Using `withContext` causes the coroutine to execute the block passed in the dispatcher specified. Here it specifies `Dispatchers.IO` which is a large thread pool that's tuned specifically for handling IO operations such as database writes. When `withContext` finishes, the coroutine will continue on the dispatcher that was specified previously. This is a good way to switch threads briefly to handle expensive operations such as disk IO or CPU intensive tasks that shouldn't be run on the main thread.

We don't need scopes here because this `suspend` function doesn't launch a coroutine. This function will run in the scope of the caller coroutine.

You may have noticed that this code no longer explicitly passes a loading state. When we change `MainViewModel` to call this `suspend` function next you'll see that it does not need to be explicit in the coroutine implementation.

Prefer suspend functions for Kotlin APIs

The extension `await` defined here is a good way to bridge an existing API to coroutines. However, it relies upon the caller remembering to call `await`. When designing new APIs for use in Kotlin, it is better to provide results directly through `suspend` functions.

`suspend fun fetchNewWelcome: String`

If `fetchNewWelcome` were redefined as a suspend function like this callers wouldn't need to remember to call `await` every time they used it.

Use suspend function in MainViewModel

Open `MainViewModel.kt` and replace `refreshTitle` with a coroutine based implementation.

```
// MainViewModel.kt
```

```
fun refreshTitle() {  
    viewModelScope.launch {  
        try {  
            _spinner.value = true  
            repository.refreshTitle()  
        } catch (error: TitleRefreshError) {  
            _snackBar.value = error.message  
        } finally {  
            _spinner.value = false  
        }  
    }  
}
```


This implementation uses normal flow control to capture errors. Since repository's `refreshTitle` is a `suspend` function, when it throws an exception it'll be exposed via `try/catch`.

The logic for showing the spinner is also straightforward. Since `refreshTitle` will suspend this coroutine until the refresh is complete, there is no need to pass the loading state explicitly via a callback. Instead, the loading spinner can be controlled by the `ViewModel` and hidden in a `finally` block.

What happens to uncaught exceptions

Uncaught exceptions in a coroutine scope are similar to uncaught exceptions in non-coroutine code. By default, they'll cancel the job that was passed to the scope and pass the exception to the `uncaughtExceptionHandler`.

You can customize this behavior by providing a [CoroutineExceptionHandler](#).

Run the application again by selecting the **app** configuration then pressing , you should see a loading spinner when you tap anywhere. The title will be updated from the room database, or in the case of error a snackbar will be shown.

In the next exercise you'll refactor this to use a general data loading function.

10. Using coroutines in higher order functions

In this exercise you'll refactor `refreshTitle` in `MainViewModel` to use a general data loading function. This will teach you how to build higher order functions that use coroutines.

The current implementation of `refreshTitle` works, but we can create a general data loading coroutine that always shows the spinner. This might be helpful in a codebase that loads data in response to several events, and wants to ensure the loading spinner is consistently displayed.

Reviewing the current implementation every line except `repository.refreshTitle()` is boilerplate to show the spinner and display errors.

```
// MainViewModel.kt

fun refreshTitle() {
    viewModelScope.launch {
        try {
            _spinner.value = true
            // this is the only part that changes between sources
            repository.refreshTitle()
        } catch (error: TitleRefreshError) {
            _snackBar.value = error.message
        } finally {
            _spinner.value = false
        }
    }
}
```

Important: Even though we only use `viewModelScope` in this codelab, generally it's fine to add a scope anywhere that it makes sense. Don't forget to cancel it if it's no longer needed.

For example, you might declare one in a `RecyclerView Adapter` to do `DiffUtil` operations.

Using coroutines in higher order functions

Find the `TODO` to implement `launchDataLoad` in `MainViewModel.kt`:

```
// MainViewModel.kt

/**
 * Helper function to call a data load function with a loading spinner, errors will trigger a
 * snackbar.
 *
 * By marking `block` as `suspend` creates a suspend lambda which can call suspend
 * functions.
 *
 * @param block lambda to actually load data. It is called in the viewModelScope. Before calling the
 * lambda the loading spinner will display, after completion or error the loading
 * spinner will stop
 */

// TODO: Add launchDataLoad here then refactor refreshTitle to use it
```

Replace this `TODO` with this implementation

```
// MainViewModel.kt
```

```
private fun loadDataLoad(block: suspend () -> Unit): Job {
    return viewModelScope.launch {
        try {
            _spinner.value = true
            block()
        } catch (error: TitleRefreshError) {
            _snackBar.value = error.message
        } finally {
            _spinner.value = false
        }
    }
}
```

Now refactor `refreshTitle()` to use this higher order function.

```
// MainViewModel.kt
```

```
fun refreshTitle() {
    loadDataLoad {
        repository.refreshTitle()
    }
}
```

By abstracting the logic around showing a loading spinner and showing errors, we've simplified our actual code needed to load data. Showing a spinner or displaying an error is something that's easy to generalize to any data loading, while the actual data source and destination needs to be specified every time.

To build this abstraction, `loadDataLoad` takes an argument `block` that is a suspend lambda. A suspend lambda allows you to call suspend functions. That's how Kotlin implements the coroutine builders `launch` and `runBlocking` we've been using in this codelab.

```
// suspend lambda
```

```
block: suspend () -> Unit
```

To make a suspend lambda, start with the `suspend` keyword. The function arrow and return type `Unit` complete the declaration.

You don't often have to declare your own suspend lambdas, but they can be helpful to create abstractions like this that encapsulate repeated logic!

In the next exercise you'll learn how to call coroutine based code from `WorkManager`.

11. Using coroutines with WorkManager

In this exercise you'll learn how to use coroutine based code from WorkManager.

What is WorkManager

There are many options on Android for deferrable background work. This exercise shows you how to integrate [WorkManager](#) with coroutines. WorkManager is a compatible, flexible and simple library for deferrable background work. WorkManager is the recommended solution for these use cases on Android.

WorkManager is part of [Android Jetpack](#), and an [Architecture Component](#) for background work that needs a combination of opportunistic and guaranteed execution. Opportunistic execution means that WorkManager will do your background work as soon as it can. Guaranteed execution means that WorkManager will take care of the logic to start your work under a variety of situations, even if you navigate away from your app.

Because of this, WorkManager is a good choice for tasks that must complete eventually.

Some examples of tasks that are a good use of WorkManager:

- Uploading logs
- Applying filters to images and saving the image
- Periodically syncing local data with the network

To learn more about WorkManager check out the [documentation](#).

Using coroutines with WorkManager

WorkManager provides different implementations of its base `ListenableWorker` class for different use cases.

The simplest `Worker` class allows to have some synchronous operation executed by WorkManager. However, having worked so far to convert our codebase to use coroutines and suspend functions, the best way to use WorkManager is through the `CoroutineWorker` class that allows to define our `doWork()` function as a suspend function:

```
class RefreshMainDataWork(context: Context, params: WorkerParameters) :  
    CoroutineWorker(context, params) {  
  
    override suspend fun doWork(): Result {  
        val database = getDatabase(applicationContext)  
        val repository = TitleRepository(MainNetworkImpl, database.titleDao)  
  
        return try {  
            repository.refreshTitle()  
            Result.success()  
        } catch (error: TitleRefreshError) {  
            Result.failure()  
        }  
    }  
}
```

Note that `CoroutineWorker.doWork()` is a suspending function. Unlike the simpler `Worker` class, this code does NOT run on the `Executor` specified in your WorkManager configuration.

Testing our CoroutineWorker

No codebase should be complete without testing.

WorkManager makes available a couple of different ways to test your Worker classes, to learn more about the original testing infrastructure, you can read the [documentation](#).

WorkManager v2.1 introduces a new set of APIs to support a simpler way to test ListenableWorker classes and, as a consequence, CoroutineWorker. In our code we're going to use one of these new API: [TestListenableWorkerBuilder](#).

To add our new test, create a new kotlin source file with the name RefreshMainDataWorkTest under the androidTest folder. The full path of the file will be: `app/src/androidTest/java/com/example/android/kotlincoroutines/main/RefreshMainDataWorkTest.kt`.

The content of the file is:

```
package com.example.android.kotlincoroutines.main

import android.content.Context
import androidx.test.core.app.ApplicationProvider
import androidx.work.ListenableWorker.Result
import androidx.work.testing.TestListenableWorkerBuilder
import com.example.android.kotlincoroutines.util.DefaultErrorDecisionStrategy
import com.example.android.kotlincoroutines.util.ErrorDecisionStrategy
import org.hamcrest.CoreMatchers.`is`
import org.junit.Assert.assertThat
import org.junit.Before
import org.junit.Test
import org.junit.runner.RunWith
import org.junit.runners.JUnit4

@RunWith(JUnit4::class)
class RefreshMainDataWorkTest {
    private lateinit var context: Context

    @Before
    fun setup() {
        context = ApplicationProvider.getApplicationContext()

        DefaultErrorDecisionStrategy.delegate =
            object: ErrorDecisionStrategy {
                override fun shouldError() = false
            }
    }

    @Test
    fun testRefreshMainDataWork() {
        // Get the ListenableWorker
        val worker = TestListenableWorkerBuilder<RefreshMainDataWork>(context).build()

        // Start the work synchronously
        val result = worker.startWork().get()

        assertThat(result, `is`(Result.success()))
    }
}
```

In the setup function we configure the decision strategy in our simulated network connection to never fail (otherwise the test will occasionally fail with the error threshold we've in the default decision strategy).

The test itself uses the `TestListenableWorkerBuilder` to create our worker that we can then run calling the `startWork()` method.

WorkManager is just one example of how coroutines can be used to simplify APIs design.

12. Where to learn more

In this codelab we have covered the basics you'll need to start using coroutines in your app!

We covered how to integrate coroutines to Android apps from both the UI and WorkManager jobs to simplify asynchronous programming. And how to use coroutines inside a `ViewModel` to fetch data from the network and save it to a database without blocking the main thread as well as how to cancel all coroutines when the `ViewModel` is finished.

For testing coroutine based code, we covered both by testing behavior as well as directly calling `suspend` functions from tests. We also converting how to convert existing callback-based APIs to coroutines using `suspendCoroutine`.

Kotlin coroutines have many features that weren't covered by this codelab. If you're interested in learning more about Kotlin coroutines, read the [coroutines guides](#) published by JetBrains.