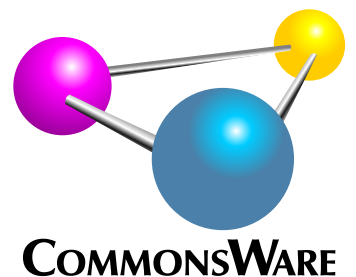# Elements of Kotlin Coroutines

20
Kc

Mark L. Murphy

Elements of Kotlin Coroutines

by Mark L. Murphy

Elements of Kotlin Coroutines
by Mark L. Murphy

Table of Contents

Headings formatted in *bold-italic* have changed since the last version.

Preface

Thanks!

First, thanks for your interest in Kotlin! Right now, Kotlin is Google's primary language for Android app development, with Java slowly being relegated to second-tier status. Kotlin is popular outside of Android as well, whether you are building Web apps, desktop apps, or utility programs. Coroutines represent an important advancement in the Kotlin ecosystem, but it is a fairly complex topic.

And, to that end... thanks for picking up this book! Here, you will learn what problems coroutines solve and how to implement them, in isolation and in the context of larger apps.

The Book's Prerequisites

This book assumes that you have basic familiarity with Kotlin syntax, data types, and core constructs (e.g., lambda expressions).

If you are new to Kotlin, please read *Elements of Kotlin* or another Kotlin book first, then return here. Don't worry — we'll wait for you.

.

.

.

OK! At this point, you're an experienced Kotlin developer, ready to take the plunge into coroutines!

(and if you are still a bit new to Kotlin… that's OK — we won't tell anyone!)

About the Updates

Once this book reaches Version 1.0, it will be updated sporadically. JetBrains – the firm that created Kotlin — is fairly aggressive about improving the language. New minor versions of Kotlin (e.g., Kotlin 1.4) come out every year or two, and this book will be updated to reflect those changes.

If you obtained this book through the Warescription, you will be able to download updates as they become available, for the duration of your subscription period.

If you obtained this book through other channels… um, well, it's still a really nice book!

Each release has notations to show what is new or changed compared with the immediately preceding release:

- The Table of Contents in the ebook formats (PDF, EPUB, MOBI/Kindle) shows sections with changes in **_bold-italic_** font
- Those sections have changebars on the right to denote specific paragraphs that are new or modified

And, there is the "What's New" section, just below this paragraph.

What's New in Version 0.3?

There is a new chapter on creating custom scopes.

Also, the material was updated for version 1.4.2 of the coroutines libraries. This includes a substantial revision to the chapter on `SharedFlow` and `StateFlow`. Also material on channels has been more firmly placed in a secondary role, compared to flows… though channels still have their uses.

And, as usual, there were lots of bug fixes.

Warescription

If you purchased the Warescription, read on! If you obtained this book from other

channels, feel free to jump ahead.

The Warescription entitles you, for the duration of your subscription, to digital editions of this book and its updates, in PDF, EPUB, and Kindle (MOBI/KF8) formats, plus the ability to read the book online at the Warescription Web site. You also have access to other books that CommonsWare publishes during that subscription period.

Each subscriber gets personalized editions of all editions of each title. That way, your books are never out of date for long, and you can take advantage of new material as it is made available.

However, you can only download the books while you have an active Warescription. There is a grace period after your Warescription ends: you can still download the book until the next book update comes out after your Warescription ends. After that, you can no longer download the book. Hence, **please download your updates as they come out**. You can find out when new releases of this book are available via:

1. The CommonsBlog
2. The CommonsWare Twitter feed
3. Opting into emails announcing each book release — log into the Warescription site and choose Configure from the nav bar
4. Just check back on the Warescription site every month or two

Subscribers also have access to other benefits, including:

- "Office hours" — online chats to help you get answers to your Android application development questions. You will find a calendar for these on your Warescription page.
- A Stack Overflow "bump" service, to get additional attention for a question that you have posted there that does not have an adequate answer.
- A discussion board for asking arbitrary questions about Android app development.

Book Bug Bounty

Find a problem in the book? Let CommonsWare know!

Be the first to report a unique concrete problem in the current edition, and

CommonsWare will extend your Warescription by six months as a bounty for helping CommonsWare deliver a better product.

By "concrete" problem, we mean things like:

1. Typographical errors
2. Sample applications that do not work as advertised, in the environment described in the book
3. Factual errors that cannot be open to interpretation

By "unique", we mean ones not yet reported. Be sure to check the book's errata page, though, to see if your issue has already been reported. One coupon is given per email containing valid bug reports.

We appreciate hearing about "softer" issues as well, such as:

1. Places where you think we are in error, but where we feel our interpretation is reasonable
2. Places where you think we could add sample applications, or expand upon the existing material
3. Samples that do not work due to "shifting sands" of the underlying environment (e.g., changed APIs with new releases of an SDK)

However, those "softer" issues do not qualify for the formal bounty program.

Questions about the bug bounty, or problems you wish to report for bounty consideration, should be sent to bounty@commonsware.com.

Source Code and Its License

The source code in this book is licensed under the Apache 2.0 License, in case you have the desire to reuse any of it.

Copying source code directly from the book, in the PDF editions, works best with Adobe Reader, though it may also work with other PDF viewers. Some PDF viewers, for reasons that remain unclear, foul up copying the source code to the clipboard when it is selected.

Creative Commons and the Four-to-Free (42F)

Guarantee

Each CommonsWare book edition will be available for use under the [Creative Commons Attribution-Noncommercial-ShareAlike 3.0](#) license as of the fourth anniversary of its publication date, or when 4,000 copies of the edition have been sold, whichever comes first. That means that, once four years have elapsed (perhaps sooner!), you can use this prose for non-commercial purposes. That is our Four-to-Free Guarantee to our readers and the broader community. For the purposes of this guarantee, new Warescriptions and renewals will be counted as sales of this edition, starting from the time the edition is published.

This edition of this book will be available under the aforementioned Creative Commons license on *1 March 2025*. Of course, watch the CommonsWare Web site, as this edition might be relicensed sooner based on sales.

For more details on the Creative Commons Attribution-Noncommercial-ShareAlike 3.0 license, visit [the Creative Commons Web site](#)

Note that future editions of this book will become free on later dates, each four years from the publication of that edition or based on sales of that specific edition. Releasing one edition under the Creative Commons license does not automatically release *all* editions under that license.

Acknowledgments

The author would like to thank JetBrains for their development of Kotlin. In particular, the author would like to thank Roman Elizarov and the rest of the coroutines development team.

The Rudiments of Coroutines

Introducing Coroutines

Kotlin is an ever-evolving language, with a steady stream of new releases. 2018 saw the release of Kotlin 1.3, and perhaps the pre-eminent feature in that release was the coroutines system.

While coroutines do not change the *language* very much, they will change the *development practices* of Kotlin users substantially. For example, Google is supporting coroutines in some of the Android Jetpack Kotlin extension libraries ("Android KTX"). Other Jetpack libraries, like Paging 3, are coroutines-centric.

With all that in mind, you may be wondering what the fuss is all about.

The Problems

As experienced programmers know, software development is a series of problems occasionally interrupted by the release of working code.

Many of those problems share common themes, and one long-standing theme in software development is "threading sucks".

Doing Work Asynchronously

We often need to do work asynchronously, and frequently that implies the use of threads.

For example, for the performance of a Web browser to be reasonable, we need to download assets (images, JavaScript files, etc.) in parallel across multiple threads. The bottleneck tends to be the network, so we parallelize a bunch of requests, queue up the remainder, and get whatever other work done that we can while we wait for

the network to give us our data.

Getting Results on a "Magic Thread"

In many environments, one or more threads are special with respect to our background work:

- In Android, JavaFx, and other GUI environments, updates to the UI are single-threaded, with a specific thread being responsible for those updates. Typically, we *have* to do any substantial work on background threads, so we do not tie up the UI thread and prevent it from updating the UI. However, in some cases, we also need to ensure that we only try to update the UI from the UI thread. So, while the long-running work might need to be done on a background thread, the UI effects of that work need to be done on the UI thread.
- In Web app development, traditional HTTP verbs (e.g., GET, PUT, POST) are synchronous requests. The Web app forks a thread to respond to a request, and it is up to the code executing on that thread to construct and return the response. Even if that thread delegates work to some other thread (e.g., a processing thread pool) or process (e.g., a microservice), the Web request thread needs to block waiting for the results (or for some sort of timeout), so that thread can build the appropriate response.
- And so on

Doing So Without Going to Hell

Software developers, as a group, are fond of mild profanity. In particular, we often describe things as being some form of "hell", such as "DLL hell" for the problems of cross-software shared dependencies, as illustrated by DLL versioning in early versions of Microsoft Windows.

Similarly, the term "callback hell" tends to be used in the area of asynchronous operations.

Callbacks are simply objects representing chunks of code to be invoked when a certain condition occurs. In the case of asynchronous operations, the "certain condition" often is "when the operation completes, successfully or with an error". In some languages, callbacks could be implemented as anonymous functions or lambda expressions, while in other languages they might need to be implementations of certain interfaces.

"Callback hell" occurs when you have lots of nested callbacks, such as this Java snippet:

```java
doSomething(new Something.Callback() {
  public void whenDone() {
    doTheNextThing(new NextThing.Callback() {
      public void onSuccess(List<String> stuff) {
        doSomethingElse(stuff, new SomethingElse.Callback() {
          public void janeStopThisCrazyThing(String value) {
            // TODO
          }
        });
      }

      public void onError(Throwable t) {
        // TODO
      }
    });
  }
});
```

Here, `doSomething()`, `doTheNextThing()`, and `doSomethingElse()` might all arrange to do work on background threads, calling methods on the callbacks when that work completes.

For cases where we need the callbacks to be invoked on some *specific* thread, such as a UI thread, we would need to provide some means for the background code to do that. For example, in Android, the typical low-level solution is to pass in a `Looper`;

```java
doSomething(Looper.getMainLooper(), new Something.Callback() {
  public void whenDone() {
    doTheNextThing(Looper.getMainLooper(), new NextThing.Callback() {
      public void onSuccess(List<String> stuff) {
        doSomethingElse(stuff, Looper.getMainLooper(), new SomethingElse.Callback() {
          public void janeStopThisCrazyThing(String value) {
            // TODO
          }
        });
      }

      public void onError(Throwable t) {
        // TODO
      }
    });
  }
});
```

The `doSomething()`, `doTheNextThing()`, and `doSomethingElse()` methods could then use that `Looper` (along with a `Handler` and some `Message` objects) to get the callback calls to be made on the thread tied to the `Looper`.

This is a bit ugly, and it only gets worse as our scenarios get more complex.

What Would Be Slick

The ugliness comes from the challenges in following the execution flow through layers upon layers of callbacks. Ideally, the *syntax* for our code would not be inextricably tied to the *threading model* of our code.

If `doSomething()`, `doTheNextThing()`, and `doSomethingElse()` could all do their work on the current thread, we would have something more like this:

```
doSomething();

try {
  String result = doSomethingElse(doTheNextThing());

  // TODO
}
catch (Throwable t) {
  // TODO
}
```

Or, in Kotlin syntax:

```
doSomething()

try {
  val result = doSomethingElse(doTheNextThing())

  // TODO
} catch (t: Throwable) {
  // TODO
}
```

What we want is to be able to do something like this, while still allowing those functions to do their work on other threads.

Actual Coroutine Syntax

As it turns out, coroutines does just that. If `doSomething()`, `doTheNextThing()`, and `doSomethingElse()` all employ coroutines, our code invoking those functions could look something like this:

```kotlin
someCoroutineScope.launch {
  doSomething()

  try {
    val result = doSomethingElse(doTheNextThing())

    // TODO
  } catch (t: Throwable) {
    // TODO
  }
}
```

There is a *lot* of "plumbing" in Kotlin — both in the language and in libraries — that makes this simple syntax possible. We, as users of Kotlin, get to enjoy the simple syntax.

Trying Out Coroutines In the Klassbook

This book will have lots of sample snippets of Kotlin code demonstrating the use of coroutines. You may want to try running those yourself, with an eye towards changing the snippets and experimenting with the syntax, options, and so on.

Many of the samples are in the Klassbook.

The Klassbook is an online Kotlin sandbox, pre-populated with hundreds of different snippets (or "lessons", as Klassbook terms them). Many of the Kotlin code snippets shown in this book are available in the Klassbook, with links below the snippet leading you to the specific Klassbook lesson.

For example, here is a bit of Kotlin

```kotlin
  println("hello, world!")
```

(from ["Hello, World! (In a Function)" in the Klassbook](#))

The "Hello, World! (In a Function)" link beneath the snippet leads you to the

corresponding Klassbook page:

Figure 1: Klassbook Sample, As Initially Displayed
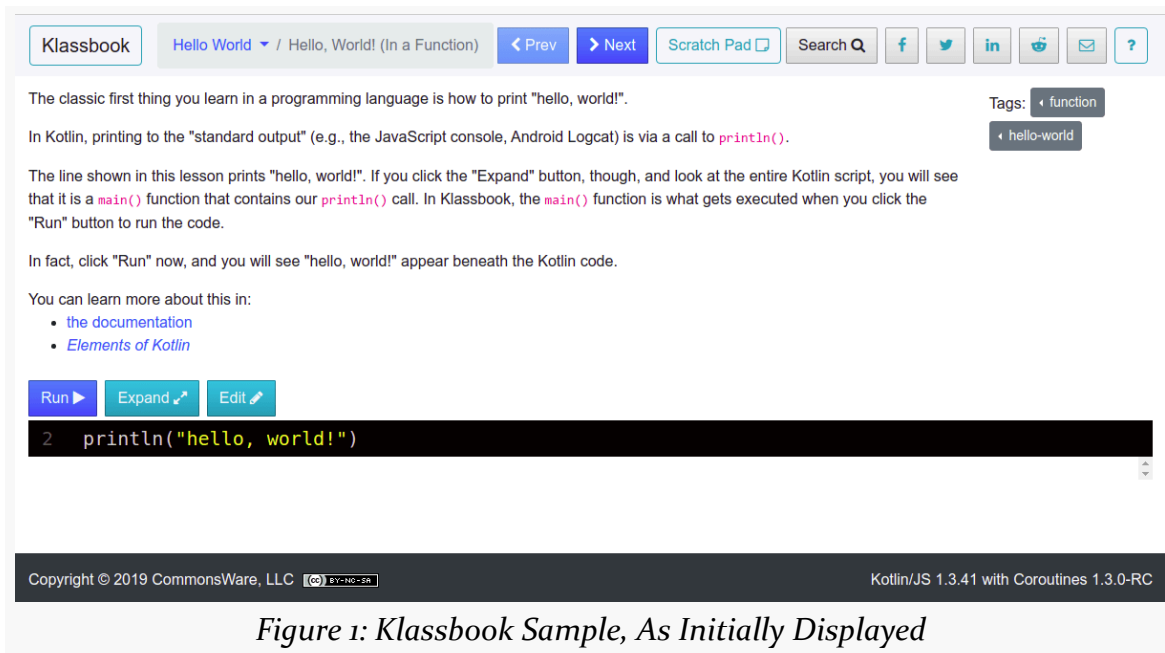
Clicking the "Run" button will execute that Kotlin code and show you the results beneath the code:

Figure 2: Klassbook Sample, After Running the Snippet

Frequently, the snippet shown in the book will be a subset of the actual Kotlin code, such as skipping the main() function. The Klassbook page will mimic the book content, but you can click the "Expand" button to see the entire Kotlin content:
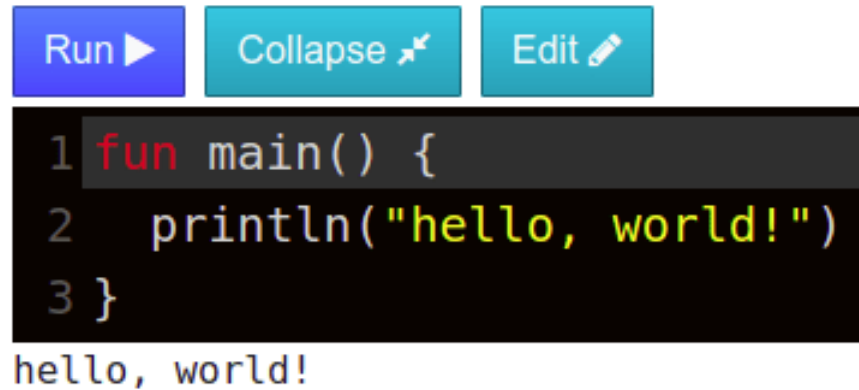
Figure 3: Klassbook Sample, with Expanded Snippet

If you want to play around with the code, click the "Edit" button. That will automatically expand the editor (if you had not clicked "Expand" already) and make the editor read-write. You can modify the Kotlin and run that modified code. Note that it will take a bit longer to run your own custom code, as it needs to be transpiled to JavaScript first.

You can learn more about how to navigate the Klassbook on the Klassbook site.

Key Pieces of Coroutines

Let's look at the following use of coroutines:

```kotlin
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the delay")
    stallForTime()
    println("This is executed after the delay")
  }

  println("This is executed immediately")
}
```

```
suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

(from "A Simple Coroutine Sample" in the Klassbook)

If you run this, you will see "This is executed immediately" and "This is executed before the delay" show up, then "This is executed after the delay" will appear after a two-second delay.

There are several pieces that make up the overall coroutines system that we will be focusing on over the next several chapters.

The Dependencies

While one element of coroutines (the suspend keyword) is part of the language, the rest comes from libraries. You will need to add these libraries to your project in order to be able to use coroutines. Exactly *how* you add these libraries will depend a lot on your project type and the build system that you are using. This book will focus on projects built using Gradle, such as Android projects in Android Studio or Kotlin/Multiplatform projects in IntelliJ IDEA.

This book focuses mostly on the 1.4.x versions of these dependencies (e.g., 1.4,2). Note that these versions are somewhat independent of the overall Kotlin version (1.4).

Kotlin/JVM and Android

For an Android project, you would want to add a dependency on org.jetbrains.kotlinx:kotlinx-coroutines-android. This has transitive dependencies to pull in the core coroutines code, plus it has Android-specific elements.

If you are using Kotlin/JVM for ordinary Java code, though, the Android code is of no use to you. Instead, add a dependency on org.jetbrains.kotlinx:kotlinx-coroutines-core.

Kotlin/JS

If you are working in Kotlin/JS, such as the Klassbook, you would want to add a

dependency on `org.jetbrains.kotlinx:kotlinx-coroutines-core-js`.

Overall, Kotlin/JS has the least support for coroutines among the major Kotlin variants. Mostly, that is because JavaScript itself does not offer first-class threads, but instead relies on `Promise`, web workers, and similar structures. Over time, Kotlin/JS may gain better coroutines support, and this book will help to point out some of the places where you have more options in Kotlin/JVM than you do in Kotlin/JS.

Kotlin/Native

Coroutines support for Kotlin/Native, right now, is roughly on par with that of Kotlin/JS. There is an `org.jetbrains.kotlinx:kotlinx-coroutines-core-native` dependency that you would use for such modules.

Note that this book will be focusing on Kotlin/JVM and Kotlin/JS, with very little material unique to Kotlin/Native.

Kotlin/Common

In a Kotlin/Multiplatform project, you can depend upon `org.jetbrains.kotlinx:kotlinx-coroutines-core-common` in any modules that are adhering to the Kotlin/Common subset.

The Scope

All coroutine work is managed by a `CoroutineScope`. In the sample code, we are using `GlobalScope`. As the name suggests, `GlobalScope` is a global instance of a `CoroutineScope`.

Primarily, a `CoroutineScope` is responsible for canceling and cleaning up coroutines when the `CoroutineScope` is no longer needed. `GlobalScope` will be set up to support the longest practical lifetime: the lifetime of the process that is running the Kotlin code.

However, while `GlobalScope` is reasonable for book samples like this one, more often you will want to use a scope that is a bit smaller in... well... scope. For example, if you are using coroutines in an Android app, and you are doing I/O to populate a UI, if the user navigates away from the activity or fragment, you may no longer need that coroutine to be doing its work. This is why Android, through the Jetpack, offers a range of `CoroutineScope` implementations that will clean up coroutines when they

are no longer useful.

We will explore `CoroutineScope` more [in an upcoming chapter](), and we will explore Android-specific scopes more [later in the book]().

The Builder

The `launch()` function that we are calling on `GlobalScope` is a coroutine builder. Coroutine builder functions take a lambda expression and consider it to be the actual work to be performed by the coroutine.

We will explore coroutine builders more [in an upcoming chapter]().

The Dispatcher

Part of the configuration that you can provide to a coroutine builder is a dispatcher. This indicates what thread pool (or similar structure) should be used for executing the code inside of the coroutine.

Our code snippet refers to two of these:

- `Dispatchers.Default` represents a stock pool of threads, useful for general-purpose background work
- `Dispatchers.Main` is a dispatcher that is associated with the "main" thread of the environment, such as Android's main application thread

By default, a coroutine builder will use `Dispatchers.Default`, though that default can be overridden in different circumstances, such as in different implementations of `CoroutineScope`.

We will explore the various dispatcher editions [in an upcoming chapter]().

Our `main()` function uses the `launch()` coroutine builder to indicate a block of code that should run on the main application thread. This means that we will call `stallForTime()` and the second `println()` function on the main application thread. What those functions do, though, might involve other threads, as we will see [a bit later in this chapter]().

The `suspend` Function

The coroutine builders set up blocks of code to be executed by certain thread pools.

Java developers might draw an analogy to handing a `Runnable` over to some `Executor`. For `Dispatchers.Main`, Android developers might draw an analogy to handing a `Runnable` over to `post()` on a `View`, to have that `Runnable` code be executed on the Android main application thread.

However, those analogies are not quite complete.

In those `Runnable` scenarios, the unit of work for the designated thread (or thread pool) is the `Runnable` itself. Once the thread starts executing the code in that `Runnable`, that thread is now occupied. So, if elsewhere we try handing other `Runnable` objects over, those will wait until the first `Runnable` is complete.

In Kotlin, though, we can mark functions with the `suspend` keyword. This tells the coroutines system (particularly the current dispatcher) that it is OK to suspend execution of the current block of code *and to feel free to run other code from another coroutine builder* if there is any such code to run.

Our `stallForTime()` function has the `suspend` keyword. So, when Kotlin starts executing the code that we provided to `launch()`, when it comes time to call `stallForTime()`, Kotlin could elect to execute other coroutines scheduled for `Dispatchers.Main`. However, Kotlin will not execute the `println()` statement on the line after the `stallForTime()` call until `stallForTime()` returns.

Any function marked with `suspend` needs to be called either from inside of a coroutine builder or from another function marked with the `suspend` keyword. `stallForTime()` is OK, because we are calling it from code being executed by a coroutine builder (`launch()`). As it turns out, `delay()` — called inside `stallForTime()` — also is marked with `suspend`. In our case, that is still OK, because `stallForTime()` has the `suspend` keyword, so it is safe to call `suspend` functions like `delay()` from within `stallForTime()`. `delay()`, as you might imagine, delays for the requested number of milliseconds.

We will explore `suspend` functions much more in [an upcoming chapter](#).

The Context

The dispatcher that we provide to a coroutine builder is part of a `CoroutineContext`. As the name suggests, a `CoroutineContext` provides a context for executing a coroutine. The dispatcher is part of that context, but there are other elements of a `CoroutineContext`, such as a `Job`, as we will discuss in [an upcoming chapter](#).

The `withContext()` top-level function is a `suspend` function, so it can only be executed from inside of another `suspend` function or from inside of a code block executed by a coroutine builder. `withContext()` takes a different block of code and executes it with a modified `CoroutineContext`.

In our snippet, we use `withContext()` to switch to a different dispatcher. Our coroutine starts executing on the main application thread (`Dispatchers.Main`). In `stallForTime()`, though, we execute our `delay()` call on `Dispatchers.Default`, courtesy of the `withContext()` call. `withContext()` will block until the code completes, but since it is a `suspend` function, Kotlin could start work on some other `Dispatchers.Main` coroutine while waiting for our `withContext()` call to end.

Suspending `main()`

You do not need to use `GlobalScope.launch()` inside your `main()` function. Instead, you can just put the `suspend` keyword on `main()` itself:

```
import kotlinx.coroutines.*

suspend fun main() {
  println("This is executed before the delay")
  stallForTime()
  println("This is executed after the delay")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

(from ["Suspending main()" in the Klassbook](#))

Now, you can call other `suspend` functions from `main()` without having to fuss with using `GlobalScope` or some other coroutine scope.

The examples in this book — other than the one shown above — will not use this. The Klassbook samples all have `main()` functions, but that is simply how the Klassbook works. Little production code is used directly from a `main()` function. And your choice of coroutine scope and coroutine builder are fairly important concepts. So, while the `suspend main()` approach works, you will not see it used much here.

The Timeline of Events

Let's look at two variations of the previous sample and use them to examine how coroutines handle sequential statements and parallel work.

Sequential Statements

```kotlin
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the first delay")
    stallForTime()
    println("This is executed after the first delay")
    println("This is executed before the second delay")
    stallForTime()
    println("This is executed after the second delay")
  }

  println("This is executed immediately")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

(from ["A Sequential Coroutine Sample" in the Klassbook](#))

Here, we have one coroutine, one that triggers two `delay()` calls (by way of two `stallForTime()` calls).

If you run this sample, you will see that the output looks like:

```
This is executed immediately
This is executed before the first delay
This is executed after the first delay
This is executed before the second delay
This is executed after the second delay
```

Within a coroutine, each statement is executed sequentially.

Concurrent Coroutines

Now, let's divide that work into two separate coroutines, though both are tied to `Dispatchers.Main`:

```kotlin
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the first delay")
    stallForTime()
    println("This is executed after the first delay")
  }

  GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the second delay")
    stallForTime()
    println("This is executed after the second delay")
  }

  println("This is executed immediately")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

If you execute *this* sample, the results are different:

```
This is executed immediately
This is executed before the first delay
This is executed before the second delay
This is executed after the first delay
This is executed after the second delay
```

We enqueue two coroutines. Kotlin starts executing the first one, but when it hits the `stallForTime()` call, it knows that it can suspend execution of that first coroutine, if desired. In the sequential example, we only have one coroutine, so Kotlin just waits for our first `stallForTime()` call to complete before proceeding. Now, though, we have two coroutines, so when Kotlin encounters our first `stallForTime()`, Kotlin *can start executing the second coroutine,* even though both

coroutines are tied to the same thread (the single "magic" thread associated with `Dispatchers.Main`). So Kotlin runs the first `println()` from the second coroutine, then hits `stallForTime()`. At this point, Kotlin has no unblocked coroutines to run, so it waits for one of the `suspend` functions to complete. It can then resume execution of that coroutine.

So, while within a single coroutine, all statements are sequential, multiple coroutines for the same dispatcher can be executed concurrently. Kotlin can switch between coroutines when it encounters a `suspend` function.

Cheap, Compared to Threads

In thread-centric programming, we worry about creating too many threads. Each thread consumes a chunk of heap space. Plus, context-switching between threads consumes CPU time on top of the actual code execution. This is why we have scaling algorithms for sizing thread pools (e.g., "twice the number of CPU cores, plus one").

However, in Kotlin, coroutines do not declare what *thread* they run on — they declare what *dispatcher* they run on. The dispatcher determines the threading rules, which can be a single thread a or constrained thread pool.

As a result, we can execute a lot of coroutines fairly cheaply:

```kotlin
import kotlinx.coroutines.*

fun main() {
  for (i in 1..100) {
    GlobalScope.launch(Dispatchers.Main) {
      println("This is executed before delay $i")
      stallForTime()
      println("This is executed after delay $i")
    }
  }
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

(from ["Massively Parallel Coroutines" in the Klassbook](#))

Here, we execute 100 coroutines, each delaying for two seconds. If we were using threads for each of those blocks, we would have 100 threads, which is far too many. Instead, we have as many threads as `Dispatchers.Main` uses, and `Dispatchers.Main` uses only one thread. Yet, we can do work on the coroutines concurrently, with Kotlin switching between them when it encounters `suspend` functions. So, our code will wind up printing all 100 "before delay" messages before printing any of the "after delay" messages, as we should be able to print 100 messages before the first two-second delay expires.

The History of Coroutines

While Kotlin may be the most popular use of coroutines today, coroutines themselves have been around as a programming construct for quite some time. They fell out of favor for much of that time, though, in favor of a thread-based concurrency model.

And all of this comes back to "multitasking".

Preemptive and Cooperative

The original use of the term "multitasking" in programming was in regards to how programs would appear to perform multiple things ("tasks") at once. Two major approaches for multitasking evolved: cooperative and preemptive.

Most modern programmers think in terms of preemptive multitasking, as that is the form of multitasking offered by processes and threads. Here, the code that implements a task is oblivious (mostly) to the fact that there is any sort of multitasking going on. Instead, arranging to have tasks run or not is a function of the operating system. The OS schedules processes and threads to run on CPU cores, and the OS switches cores to different processes and threads to ensure that everything that is supposed to be executing gets a chance to do a little bit of work every so often.

However, some of the original multitasking work was performed on a cooperative basis. Here, code needs to explicitly "yield" and indicate that if there is some other task needing CPU time, this would be a good point in which to switch to that task. Some framework — whether part of an OS or within an individual process — can then switch between tasks at their yield points. 16-bit Windows programs and the original Mac OS used cooperative multitasking at their core.

More often nowadays, cooperative multitasking is handled by some framework inside of a process. The OS itself uses preemptive multitasking, to help manage misbehaving processes. Within a process, cooperative multitasking might be used. For example, 32-bit Windows moved to a preemptive multitasking approach overall, but added fibers as a cooperative multitasking option that could be used within a process.

Green and Red

Sometimes, a framework can provide the illusion of preemptive multitasking while still using a more-or-less cooperative multitasking model.

Perhaps the most famous example of this is Java. In the beginning, all "threads" were managed within the JVM, with the virtual machine switching code execution between those threads. Threads did not have to explicitly yield, but because the JVM was in charge of code execution, the JVM could switch between its "threads" on its own. From the OS' standpoint, this was cooperative multitasking, but from the Java programmers standpoint, it felt like preemptive multitasking.

This so-called "green threads" approach was replaced by "red threads", where `Thread` mapped to an OS thread. This was particularly important as Java started moving from its original use cases (browser applets and Swing desktop apps) to powering Web apps, where a lot more true parallel processing needed to be performed to take advantage of server power.

The Concept of Coroutines

Coroutines are [significantly older](#) than is the author of this book. In other words, coroutines are *really* old.

Coroutines originated as the main vehicle for cooperative multitasking. A coroutine had the ability to "yield" to other coroutines, indicating to the coroutines system that if there is another coroutine that is ready to run, it can do so now. Also, a coroutine might block, or suspend, waiting on some other coroutine to do some work.

Classic implementations of coroutines might literally use a `yield` keyword or statement to indicate "this is a good time to switch to some other coroutine, if needed". In Kotlin's coroutines, mostly that is handled automatically, at the point of calling a `suspend` function. So, from a programming standpoint, we do not explicitly think about yielding control, but likely points of doing so come "for free" as we set

up our coroutines.

Coroutines, as with any form of cooperative multitasking, requires cooperation. If a coroutine does not yield, then other coroutines cannot run during that period of time. This makes coroutines a poor choice of concurrency model between apps, as each app's developers have a tendency to think that their app is more important than is any other app. However, *within* an app, developers have to cooperate if their coroutines misbehave, or else their app will crash or encounter other sorts of bugs. Kotlin's coroutines are purely an in-app solution for concurrency, and Kotlin relies upon the OS and its preemptive multitasking (processes and threads) for helping to mediate CPU access between several apps.

Coroutines, There and Back Again

Once multi-threaded programming became the norm, coroutines faded into obscurity. With the rise in complexity of multi-threaded programming — coupled with other architectural changes, such as a push for reactive programming — coroutines have started to make a comeback. However, Kotlin's implementation of coroutines is one of the most prominent use of coroutines in the modern era.

Kotlin coroutines share some elements with the original coroutines. However, Kotlin coroutines also share some elements with Java's "green/red" threads system. Coroutines decouple the units of work from the means by which that work gets scheduled to run. You will be able to indicate threads or thread pools that coroutines can work on, with an eye towards dealing with challenges like Android's main application thread restrictions.

But whether the coroutines are really running purely on individual OS threads, or whether they are swapped around like Java's green threads or Windows fibers, is up to the coroutine library, not the authors of the coroutines themselves. In practice, coroutines share threads and thread pools, so we may have many more coroutines than we have threads. The coroutines cooperatively multitask, using the threads to add some level of concurrency, depending on the availability of CPU cores and the like.

Introducing Flows and Channels

A coroutine centered around simple suspend functions works great when either:

- You need asynchronous work to be done, but you do not need to receive any sort of "result" from that work; or
- You need asynchronous work to be done, and you are expecting a single object that serves as the result

However, there are many occasions in programming where you need a stream of results, not just a single result. An ordinary suspend function does not offer that. Instead, Kotlin's coroutines system offers channels and flows for streams of results.

For that, we have flows and channels. Of the two, flows are the primary streaming API for coroutines, though channels have some specialized uses.

Life is But a Stream

Quite a bit of asynchronous work can be modeled as no-result or single-result operations:

- Database transactions
- Web service calls
- Downloading or reading an image file
- And so on

Basically, anything that is transactional in nature — where each result is triggered by a distinct request — can be modeled as a no-result or single-result operation. Those work great with ordinary suspend functions.

However, it is also common to have a single routine needing to return a series of results over time:

- Durable network connections, such as WebSockets or XMPP, where the server can send down content without a fresh client request
- GPS readings
- Sensor readings from accelerometers, thermometers, etc.
- Data received from external devices via USB, Bluetooth, etc.
- And so on

Some programming environments or frameworks might have their own streams. In Android, for example, we can get several results over time from Room (as we change the contents of a database), a `ContentObserver` (for finding out about changes in a `ContentProvider`), and a `BroadcastReceiver`, among others.

Flows and channels do a much better job than simple `suspend` functions for these sorts of cases.

You're Hot and You're Cold

In programming terms, a "hot" stream is one where events are available on the stream regardless of whether anyone is paying attention to them. By contrast, a "cold" stream is one where events are available on the stream only when there is at least one consumer of the stream.

The determination of whether a stream is hot or cold can depend on where you look. For example, if you think of GPS:

- GPS satellites emit their signals regardless of whether any GPS receiver on Earth is powered on. Hence, the satellites have a hot stream of signals.
- On a smartphone, the GPS radio is usually powered down, to save on battery. It is only powered up when one or more apps request GPS fixes. Hence, the GPS subsystem on a phone has a cold stream of GPS fixes, as it only tries to emit those when there is somebody interested in them.

With Kotlin, a simple flow usually models a cold stream, while a channel usually models a hot stream. There are special types of flows that are hot (`SharedFlow` and `StateFlow`), and we will examine those [later in the book](#).

`Flow` **Basics**

Simple flows are represented in the form of a `Flow` object.

One typical way to create a `Flow` is to use the `flow()` top-level function. `flow()` is fairly simple: you supply a lambda expression, and that expression calls `emit()` for each item that you want to publish on the stream.

One typical way to consume a `Flow` is to call `collect()` on it. This takes another lambda expression, and it is passed each item that the `Flow` emits onto its stream. `collect()` is a suspend function, and so we need to call it from inside of another suspend function or from a coroutine builder like `launch()`.

So, let's print some random numbers:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    randomPercentages(10, 200).collect { println(it) }
    println("That's all folks!")
  }

  println("...and we're off!")
}

fun randomPercentages(count: Int, delayMs: Long) = flow {
  for (i in 0 until count) {
    delay(delayMs)
    emit(Random.nextInt(1,100))
  }
}
```

(from ["A Simple Flow" in the Klassbook](#))

Here, `randomPercentages()` creates a `Flow` using `flow()`. It loops a specified number of times, delays for a specified number of milliseconds for each pass, and for each pass emits a random number. Of note:

- `emit()` is a suspend function. `flow()` sets up a coroutine for you to use, so you do not need to worry about doing that yourself. But it does mean that `emit()` might trigger a switch to another coroutine, and that `emit()` might

block for a bit.
- When you exit the lambda expression, the flow is considered to be closed.

Then, inside of a launched coroutine, we call `collect()` on that `Flow`, printing each number. This will give us something like:

```
...and we're off!
41
29
6
98
49
91
15
62
40
76
That's all folks!
```

(though the numbers that you get from running the sample are very likely to be different than these)

So, our `Flow` emits objects, and our `collect()` function — which sets up a `FlowCollector` — receives them.

Channel Basics

Similarly, a simple channel is modeled as a `Channel` object.

Setting up a simple `Channel` looks a lot like setting up a simple `Flow`:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    randomPercentages(10, 200).consumeEach { println(it) }
    println("That's all folks!")
  }

  println("...and we're off!")
}
```

```kotlin
fun CoroutineScope.randomPercentages(count: Int, delayMs: Long) = produce {
  for (i in 0 until count) {
    delay(delayMs)
    send(Random.nextInt(1,100))
  }
}
```

(from ["A Simple Channel" in the Klassbook](#))

This is the same basic pattern that we used above for a Flow. There are a few differences:

- We use produce() to create the Channel, instead of flow() to create a Flow. Like flow(), produce() takes a lambda expression; when that expression completes, the channel will be closed. However, whereas flow() is a top-level function, produce() is defined on CoroutineScope. One convention for this is to put the produce() code in an extension function for CoroutineScope, then call that function from inside of the coroutine builder (e.g., launch()).
- We use send() rather than emit() to put a value onto the channel's stream.
- We use consumeEach() rather than collect() to receive the values from the channel. Like collect(), consumeEach() is a suspend function and needs to be called from within another suspend function or from within a coroutine builder like launch().

And, we get the same basic results that we did from Flow:

```
...and we're off!
69
18
21
51
74
60
57
14
49
12
That's all folks!
```

Hot and Cold Impacts

send() on a Channel, like emit() on a Flow, is a blocking call. It will not return until

something is in position to receive the item that we are placing on the stream.

Channel, though, also has offer(). offer() will try to put the item on the stream, but if it cannot, it does not block.

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    val channel = randomPercentages(10, 200)

    delay(1000)
    channel.consumeEach { println(it) }
    println("That's all folks!")
  }

  println("...and we're off!")
}

fun CoroutineScope.randomPercentages(count: Int, delayMs: Long) = produce {
  for (i in 0 until count) {
    delay(delayMs)
    offer(Random.nextInt(1,100))
  }
}
```

(from ["Hot Channels Via offer()" in the Klassbook](#))

Here, our consumer code delays a bit before calling consumeEach(). With a send()-based Channel, or with a Flow, we still wind up getting all 10 items, despite the delay, because send() and emit() block until something can receive their items. In this case, though, we are using offer(), so a few of the items will be dropped because nobody is consuming when we make our offer. As a result, we wind up with six or so items in our output, rather than the full set of 10.

Exploring Builders and Scopes

As we saw in <u>an earlier chapter</u>, there are five major pieces to coroutines:

- Coroutine scopes
- Coroutine builders
- Dispatchers
- suspend functions
- Coroutine context

In this chapter, we will focus on the first two of those, diving deeper into how we use builders and scopes.

Builders Build Coroutines

All coroutines start with a coroutine builder. The block of code passed to the builder, along with anything called from that code block (directly or indirectly), represents the coroutine. Anything else is either part of some other coroutine or is just ordinary application code.

So, you can think of a coroutine as a call tree and related state for those calls, rooted in the lambda expression supplied to the coroutine builder.

The Basic Builders

There are two coroutine builders that we will focus on in this book: `launch()` and `async()`. There are a few others, though, that you might use in specific circumstances.

launch()

`launch()` is the "fire and forget" coroutine builder. You pass it a lambda expression to form the root of the coroutine, and you want that code to be executed, but you are not looking to get a result directly back from that code. Instead, that code has only side effects, updating other data structures within your application.

`launch()` returns a `Job` object, which we can use for managing the ongoing work, such as canceling it. We will explore `Job` in detail [later in the book](#).

We saw a few examples of `launch()` previously, such as:

```kotlin
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the delay")
    stallForTime()
    println("This is executed after the delay")
  }

  println("This is executed immediately")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

(from ["A Simple Coroutine Sample" in the Klassbook](#))

async()

`async()` also creates a coroutine and also returns a type of `Job`. However, the specific type that `async()` returns is `Deferred`, a sub-type of `Job`.

`async()` also receives a lambda expression to serve as the root of the coroutine, and `async()` executes that lambda expression. However, while `launch()` ignores whatever the lambda expression returns, `async()` will deliver that to callers via the `Deferred` object. You can call `await()` on a `Deferred` object to block until that lambda expression result is ready.

Hence, `async()` is used for cases where you do want a direct result from the

coroutine.

The catch is that `await()` is itself a `suspend` function, so you need to call it inside of some other coroutine:

```kotlin
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    val deferred = GlobalScope.async(Dispatchers.Default) {
      delay(2000L)
      println("This is executed after the delay")
      1337
    }

    println("This is executed after calling async()")

    val result = deferred.await()

    println("This is the result: $result")
  }

  println("This is executed immediately")
}
```

(from <u>"async()" in the Klassbook</u>)

Here, we use `async()` to kick off some long calculation, with the "long" aspect simulated by a call to `delay()`. We then `await()` the result and use it. All of that is done inside of a coroutine kicked off by `launch()`. We get:

```
This is executed immediately
This is executed after calling async()
This is executed after the delay
This is the result: 1337
```

If you run that snippet in the Klassbook, you will see that the first two lines of output appear nearly immediately, while the latter two lines appear after the two-second delay.

We will explore `Deferred` more <u>later in the book</u>.

runBlocking()

Sometimes, you will find yourself with a desire to call a `suspend` function from

outside of a coroutine. For example, you might want to reuse a suspend function for some code that is called by something outside of your control, such as an Android framework method, where you are not in position to set up a coroutine of your own.

For that, you can use runBlocking()... if you are using Kotlin/JVM or Kotlin/Native.

As the name suggests, runBlocking() is a blocking coroutine launcher. runBlocking() will execute its lambda expression as a coroutine — so you can call suspend functions — but it will not return until the block itself completes:

```kotlin
import kotlinx.coroutines.*

fun main() {
  println("This is executed immediately")

  runBlocking {
    stallForTime()
    println("This is executed after the delay")
  }

  println("This is executed after runBlocking returns")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

(from ["runBlocking()" in the Klassbook](#))

However, runBlocking() **is not supported by Kotlin/JS**. Attempting to use it — such as attempting to run this code snippet in the Klassbook, results in a compile error.

promise()

JavaScript has Promise for asynchronous work. Kotlin/JS wraps the JavaScript Promise in its own Promise class, and the promise() coroutine builder bridges Kotlin coroutines with the JavaScript Promise system:

```kotlin
import kotlinx.coroutines.*

fun main() {
  GlobalScope.promise(Dispatchers.Default) {
```

```
    delay(2000L)
    println("This is executed after the delay")
    1337
  }.then { result ->
    println("This is the result: $result")
  }

  println("This is executed after calling promise()")
}
```

(from ["promise()" in the Klassbook](#))

In the end, `promise()` works a bit like `async()`, in that you can get the result from the lambda expression that serves as the root of the coroutine. To get that result, you can call `then()` on the `Promise` returned by `promise()`, passing in another lambda expression that will receive the result. So, we get:

```
This is executed after calling promise()
This is executed after the delay
This is the result: 1337
```

The latter two lines are executed after the two-second delay.

However, since `then()` is really deferring to JavaScript, `then()` does *not* need to be called from inside of a coroutine itself, the way `await()` does on the `Deferred` object returned by `async()`. This is why we can call `then()` directly inside of our `main()` function, without a `launch()` for some coroutine.

`promise()` **only works on Kotlin/JS**. You cannot use it on Kotlin/JVM or Kotlin/Native.

Scope = Control

A coroutine scope, as embodied in a `CoroutineScope` implementation, exists to control coroutines. Or, [as Sean McQuillan put it](#):

> A `CoroutineScope` keeps track of all your coroutines, and it can cancel all of the coroutines started in it.

In particular, in the current implementation of coroutines, a coroutine scope exists to offer "structured concurrency" across multiple coroutines. In particular, if one coroutine in a scope crashes, all coroutines in the scope are canceled. We will explore structured concurrency more [in an upcoming chapter](#).

Where Scopes Come From

Lots of things in the coroutine system create scopes. We have created a few scopes already, without even realizing it.

GlobalScope

As the name suggests, `GlobalScope` itself is a `CoroutineScope`. Specifically, it is a singleton instance of a `CoroutineScope`, so it exists for the lifetime of your process.

In sample code, such as books and blog posts, `GlobalScope` gets used a fair bit, because it is easy to access and is always around. In general, you will not use it much in production development — in fact, it should be considered to be a code smell. Most likely, there is some other focused `CoroutineScope` that you should be using (or perhaps creating).

Coroutine Builders

The `launch()` and `async()` functions that we called on `GlobalScope` create a `Job` object (in the case of `async()`, a `Deferred` subclass of `Job`). By default and convention, creating a `Job` creates an associated `CoroutineScope` for that job. So, calling a coroutine builder creates a scope.

In particular, the scope associated with a job is used when we nest coroutines, as will be explored [in an upcoming chapter](#).

Framework-Supplied Scopes

A programming environment that you are using might have scopes as part of their API. In particular, things in a programming environment that have a defined lifecycle and have an explicit "canceled" or "destroyed" concept might have a `CoroutineScope` to mirror that lifecycle.

For example, in Android app development, the `androidx.lifecycle:lifecycle-viewmodel-ktx` library adds a `viewModelScope` extension property to `ViewModel`. `ViewModel` is a class whose instances are tied to some activity or fragment. A `ViewModel` is "cleared" when that activity or fragment is destroyed for good, not counting any destroy-and-recreate cycles needed for configuration changes. The `viewModelScope` is canceled when its `ViewModel` is cleared. As a result, any coroutines created by coroutine builders (e.g., `launch()`) on `viewModelScope` get

canceled when the ViewModel is cleared.

Having viewModelScope as an extension property means that it is "just there" for your use. For example, you might have some sort of repository that exposes suspend functions that in turn use withContext() to arrange for work to be performed on a background thread. Your ViewModel can then call those repository functions using viewModelScope and its coroutine builders, such as:

```kotlin
fun save(pageUrl: String) {
  viewModelScope.launch(Dispatchers.Main) {
    _saveEvents.value = try {
      val model = BookmarkRepository.save(getApplication(), pageUrl)

      Event(BookmarkResult(model, null))
    } catch (t: Throwable) {
      Event(BookmarkResult(null, t))
    }
  }
}
```

We will explore Android's use of coroutines later in the book.

Other programming frameworks (e.g., for desktop apps, for Web apps) may offer their own similar scopes — you will need to check the documentation for the framework to see how it integrates with Kotlin coroutines.

withContext()

The withContext() function literally creates a new CoroutineContext to govern the code supplied in the lambda expression. The CoroutineScope is an element of a CoroutineContext, and withContext() creates a new CoroutineScope for its new CoroutineContext. So, calling withContext() creates a CoroutineScope.

So, if we go back to our original example:

```kotlin
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the delay")
    stallForTime()
    println("This is executed after the delay")
  }
```

```
  println("This is executed immediately")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

(from ["A Simple Coroutine Sample" in the Klassbook](#))

GlobalScope is a CoroutineScope. Then, launch() creates another CoroutineScope. withContext() creates yet another CoroutineScope. These are all nested, to support "structured concurrency", as we will examine [in an upcoming chapter](#).

coroutineScope()

withContext() usually is used to change the dispatcher, to have some code execute on some other thread pool.

If you just want a new CoroutineScope for structured concurrency, you can use coroutineScope() and keep your current dispatcher. We will explore coroutineScope() more [in an upcoming chapter](#).

supervisorScope()

The default behavior of a CoroutineScope is if one coroutine fails with an exception, the scope cancels all coroutines in the scope. Frequently, that is what we want. If we are doing N coroutines and need the results of all N of them to proceed, as soon as one crashes, we know that we do not need to waste the time of doing the rest of the work.

However, sometimes that is not what we want. For example, suppose that we are uploading N images to a server. Just because one image upload fails does not necessarily mean that we want to abandon uploading the remaining images. Instead, we might want to complete the rest of the uploads, then find out about the failures and handle them in some way (e.g., retry policy).

supervisorScope() is very similar to coroutineScope(), except that it skips the default failure rule. The failure of one coroutine due to an exception has no impact on the other coroutines executed by this scope. Instead, there are ways that you can set up your own rule for how to deal with such failures.

We will explore exceptions and coroutines more [in an upcoming chapter](#).

Test Scopes

Kotlin's coroutines system comes with a `kotlinx-coroutine-test` library. That library offers a `TestCoroutineScope` that provides greater control over how coroutines executed inside of that scope work. This is part of a larger set of classes and functions provided to help you test your coroutines-based projects.

We will explore testing coroutines [later in the book](#).

Your Own Custom Scopes

Perhaps you are creating a library that has objects with a defined lifecycle. You might elect to have them offer a custom `CoroutineScope` tied to their lifecycle, just as Android's `ViewModel` does.

This is a relatively advanced technique, one that we will explore in greater detail [later in the book](#).

Choosing a Dispatcher

Much of the focus on using coroutines is for doing work in parallel across multiple threads. That is really under the control of the dispatcher. Coroutine builders set up the coroutines, but exactly what thread they run on is handled by your chosen dispatcher.

So, in this chapter, we will explore dispatcher a bit more.

Concurrency != Parallelism

Most of what we do with coroutines is to set up concurrency. We want to say that certain blocks of code can run concurrently with other blocks of code.

However, a lot of developers equate concurrency with parallelism. They are not really the same thing.

Parallelism is saying that two things run simultaneously, using multiple threads (or processes) and multiple CPU cores in modern OS and hardware architectures.

Concurrency says that certain blocks of code are independent and *could* be running in parallel. This is why we have to worry about concurrent access to shared memory when we work with multi-threaded apps: it is quite possible that our concurrent code blocks will both try accessing that shared memory at the same time, if they happen to be running in parallel. Conversely, though, if we are in a situation where concurrent code is not running in parallel, we can "get away with" unsafe access to shared memory, because while the concurrent code is running independently, only one will access the shared memory at a time if they are not running in parallel.

For example, in the early days of Android, devices had single-core CPUs. An

`AsyncTask` would set up concurrent execution of code using a pool of 128 threads. With a single-core CPU, though, there is no real parallel execution: only one thread runs at a time. Hence, unsafe shared memory access *usually* was fine, as it was unlikely that one task's work would be interrupted by another task in the middle of that unsafe memory access. But, in 2011, we started getting multi-core CPUs in Android devices. Now, our concurrent `AsyncTask` code was more likely to run in parallel, and our unsafe shared memory access was significantly more risky. This caused Google to elect to have `AsyncTask` use a single-thread thread pool by default, instead of the 128-thread thread pool that it used to use, to help save developers from their unsafe shared memory access.

Dispatchers: Controlling Where Coroutines Run

A dispatcher is an object that knows how to arrange for a coroutine to actually run. Most dispatchers are tied to a thread or thread pool and arrange for the coroutine to run on a thread from the pool.

Coroutines != Threads

A coroutine is tied to a particular dispatcher. That dispatcher (usually) is tied to a thread pool. Indirectly, therefore, the coroutine is tied to the thread pool.

However, coroutines are cooperative. At suspension points like `suspend` function calls, `withContext()` calls, and the like, Kotlin can elect to stop execution of one coroutine and pick up execution of another one.

As a result, a coroutine might execute on different threads at different points in time. It might start on one thread from a dispatcher's thread pool, hit a suspension point, be suspended, then pick up execution on a *different* thread from the thread pool.

For example, suppose we launch a coroutine on a dispatcher that has a few other suspended coroutines and a pool of five threads:
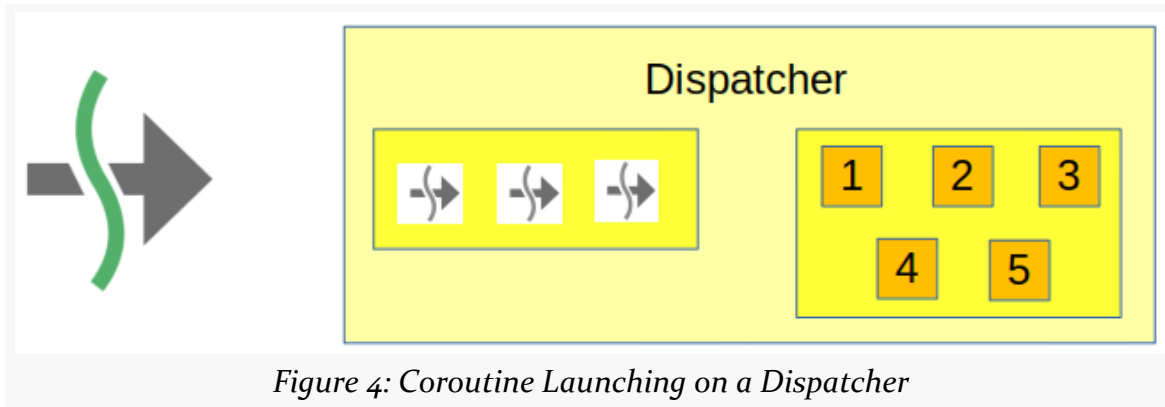
Figure 4: Coroutine Launching on a Dispatcher

Perhaps our coroutine is ready to run immediately, so the dispatcher starts executing its code, using thread 1 from the pool:
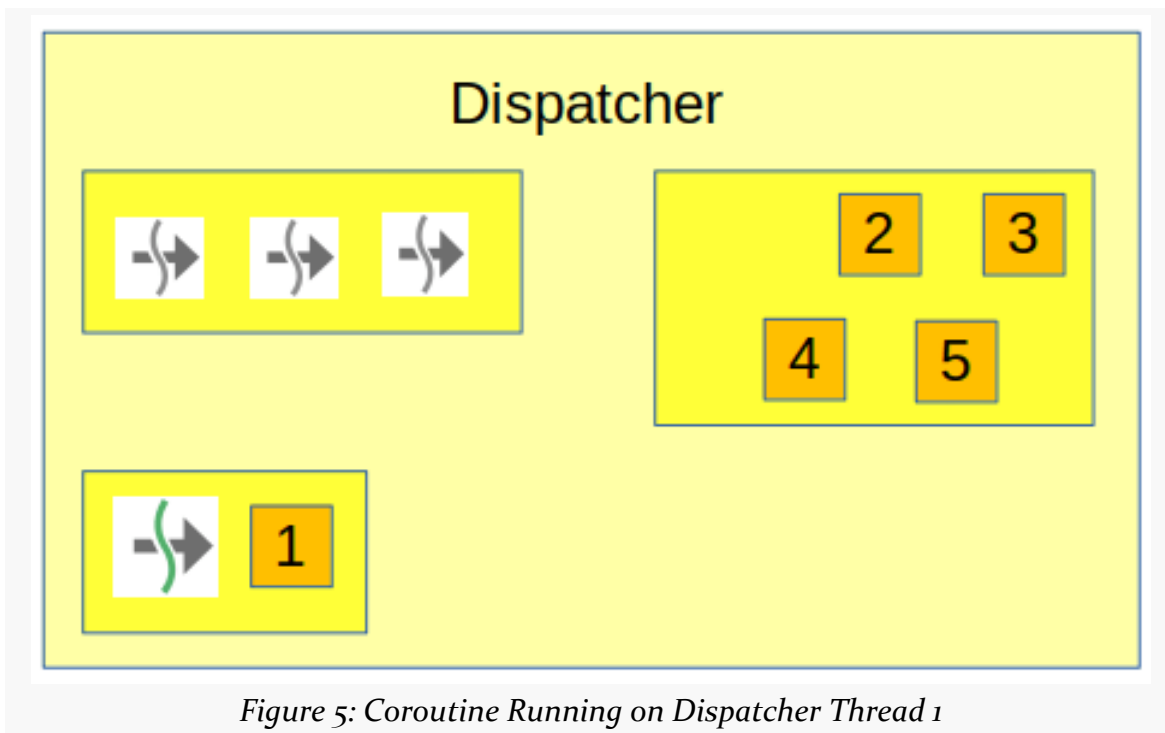
Figure 5: Coroutine Running on Dispatcher Thread 1

Eventually, that coroutine hits a suspend point, perhaps one that is blocking on some other dispatcher, so the coroutine goes into the bucket of other suspended coroutines:
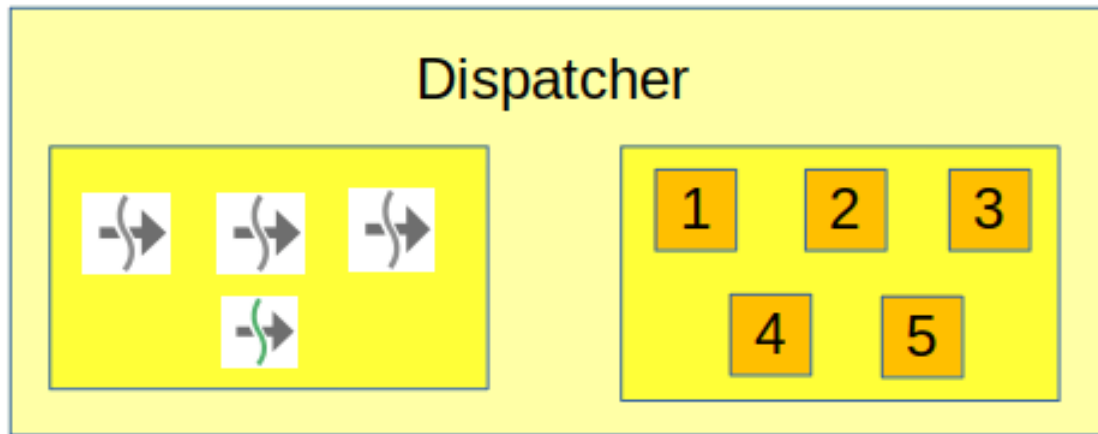
Figure 6: Suspended Coroutine

Later on, that coroutine gets unblocked. The dispatcher sets up that coroutine to run again, but this time, it happens to choose thread 2 from the pool:
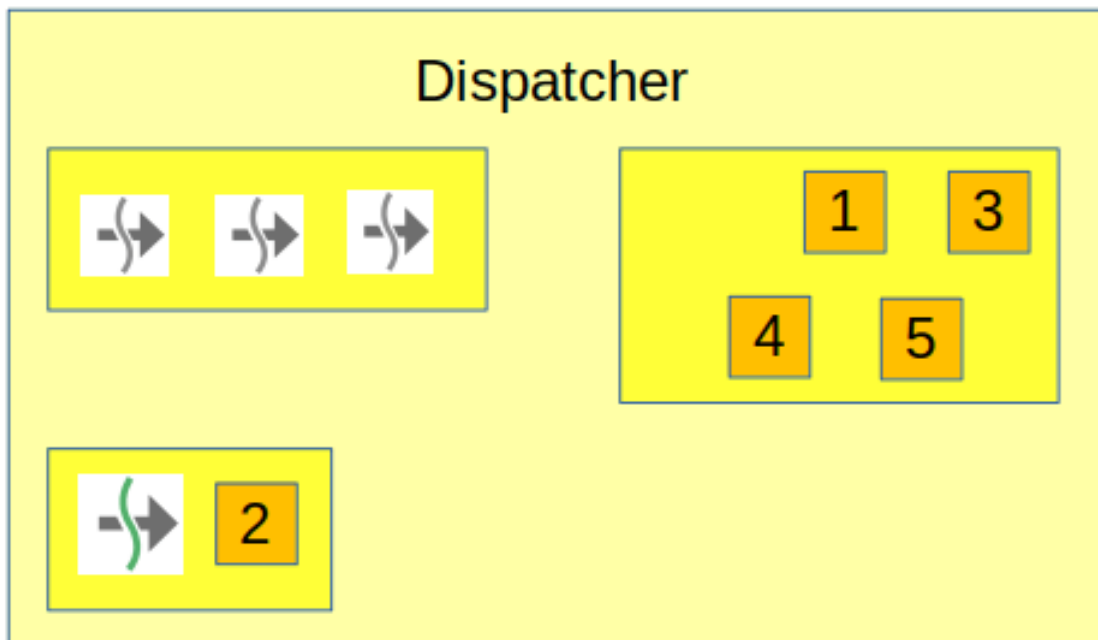
Figure 7: Coroutine Running on Dispatcher Thread 2

When we write our coroutine, we do not think of these thread switches — hiding those details is the point behind coroutines. We just need to ensure that our coroutine does not care which thread it runs on, given that the thread that it runs on can change at any suspend point.

Similarly, a thread is not dedicated to a single coroutine, unless you take steps to make that happen (e.g., a custom single-thread dispatcher that you only ever use for that one coroutine).

The idea is to make coroutines cheap — so concurrency is cheap — and to give you the ability to manage the thread pool(s) to manage parallelism to a degree.

Common Dispatchers

There are four main dispatchers (or sources of dispatchers) that you will find yourself using... if you are working in Kotlin/JVM.

If you are using Kotlin/JS, there are fewer options, owing to the restricted threading capabilities in browser-based JavaScript.

If you are using Kotlin/Native, there are fewer options as well, though that should improve over time. The limitations here are more a function of the relative youth of both coroutines and Kotlin/Native, as opposed to fundamental limitations of the platform.

`Dispatchers.Default`

If you do not provide a dispatcher to the stock implementations of `launch()` or `async()`, your dispatcher will be `Dispatchers.Default`. This dispatcher is for generic background work.

The size and nature of the thread pool backing any given dispatcher may vary based on platform. In Kotlin/JVM, this is backed by a thread pool with at least two threads that can scale upwards to 128 threads for each CPU core.

`Dispatchers.IO`

If you are building for Kotlin/JVM, you also have access to a separate dispatcher named `Dispatchers.IO`. This is designed for background work that may potentially block, such as disk I/O or network I/O.

In Kotlin/JVM, this dispatcher does not have its own thread pool. Instead, it shares a thread pool with `Dispatchers.Default`. However, `Dispatchers.IO` has different logic at the level of the dispatcher for using that thread pool, taking into account the limited actual parallelism that may be going on due to the blocking nature of the work.

So, for Kotlin/JVM, use `Dispatchers.IO` for I/O bound work or other work that tends to block, and use `Dispatchers.Default` for work that tends not to block.

`Dispatchers.Main`

`Dispatchers.Main` exists for UI work. In some environments, there is a "magic thread" that you need to use for such UI work. `Dispatchers.Main` will run its coroutines on that magic thread.

In some cases, there is no such magic thread. So, on Kotlin/JS and Kotlin/Native, `Dispatchers.Main` presently "is equivalent to" `Dispatchers.Default`, though it is not completely clear what this means.

In Kotlin/JVM, you need a specific library to get the implementation of `Dispatchers.Main` that is appropriate for a given environment:

Environment	Library
Android	`org.jetbrains.kotlinx:kotlinx-coroutines-android`
JavaFx	`org.jetbrains.kotlinx:kotlinx-coroutines-javafx`
Swing	`org.jetbrains.kotlinx:kotlinx-coroutines-swing`

If you lack such a library, and you attempt to use `Dispatchers.Main`, you will fail with an exception. If you are working in Kotlin/JVM outside of any of the environments listed in the table shown above, avoid `Dispatchers.Main`.

`asCoroutineDispatcher()`

In Kotlin/JVM, you may want to share a thread pool between coroutines and other libraries. Frequently, such a thread pool comes in the form of an `Executor` implementation, where you can supply your own `Executor` to the library, perhaps overriding a default `Executor` that it might use. For example, with OkHttp — a very

popular HTTP client library for Java — you can wrap an `ExecutorService` in an OkHttp `Dispatcher` and provide that to your `OkHttpClient.Builder`, to control the thread pool that OkHttp uses.

To use that same thread pool with coroutines, you can use the `asCoroutineDispatcher()` extension function on `Executor`. This wraps the `Executor` in a coroutine dispatcher that you can use with `launch()`, `async()`, `withContext()`, etc.

Note that this is only available for Kotlin/JVM.

Uncommon Dispatchers

Kotlin/JVM presently offers `newFixedThreadPoolContext()`, which creates a dispatcher wrapped around a dedicated thread pool with a fixed number of threads. Kotlin/JVM also presently offers `newSingleThreadContext()`, which is basically `newFixedThreadPoolContext(1)`. However, neither are recommended and both [are planned to be replaced in the future](). Use `asCoroutineDispatcher()` for your own custom `Executor` if you need a tailored thread pool right now.

If you look at the documentation for `Dispatchers`, you will see a `Dispatchers.Unconfined` object. This is a dispatcher that does not use a separate thread. Instead, its behavior is somewhat unpredictable, as your coroutine will execute on different threads depending on the other dispatchers used in your code. As the documentation indicates, `Dispatchers.Unconfined` "should not be normally used in code".

In principle, you could create your own custom `CoroutineDispatcher` subclass that implements dispatching in some custom fashion. This is well outside the scope of this book, as few developers should need to do this.

Deciding on a Dispatcher

Choosing a dispatcher with Kotlin coroutines is reminiscent of choosing a thread or thread pool for other environments (e.g., choosing a `Scheduler` for RxJava).

In general:

- Use `Dispatchers.Main` for anything that is updating your UI, for platforms where `Dispatchers.Main` has special meaning (Android, JavaFx, Swing, and perhaps others in the future)

- If you have an existing thread pool in Kotlin/JVM that you want to share with coroutines, use `asCoroutineDispatcher()`
- Use `Dispatchers.IO` for other I/O-bound work on Kotlin/JVM
- Use `Dispatchers.Default` for anything else

Specify Your Dispatcher

Coroutine builders have a default dispatcher. Technically, you only need to specify the dispatcher when the default is not what you want.

However, you may not necessarily know what the default is for a coroutine builder for a particular `CoroutineScope`. For example, you have to rummage through the Android SDK documentation to discover that some of their custom `CoroutineScope` objects, such as `viewModelScope` on a `ViewModel`, use `Dispatchers.Main` as the default dispatcher.

All else being equal, it is safest — and most self-documenting — to declare your dispatcher for your coroutine builders.

Testing and Dispatchers

The dispatcher that you want for production use may not be the dispatcher that you want for testing. You might want to use a different dispatcher that you can control better, such as the `TestCoroutineDispatcher` that is offered by the `org.jetbrains.kotlinx:kotlinx-coroutines-test` library.

There are two ways of going about this:

- `kotlinx-coroutines-test` offers the ability to override `Dispatchers.Main` and supply your own dispatcher to use instead of the default one... but this is not available for the other standard dispatchers
- Use dependency inversion or similar techniques to define the dispatchers that you are using, eliminating direct references to any standard dispatcher, so you can swap in different dispatchers for your tests

We will explore these techniques more .

`launch()` is Asynchronous

The lambda expression supplied to `launch()` runs asynchronously with respect to

the caller, even if the caller is on the thread identified by the dispatcher that you provided to launch().

Let's go back to our original coroutines sample:

```
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the delay")
    stallForTime()
    println("This is executed after the delay")
  }

  println("This is executed immediately")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

(from <u>"A Simple Coroutine Sample" in the Klassbook</u>)

Now suppose that instead of main(), we were using the above code in an Android activity:

```
class MainActivity : AppCompatActivity() {

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

    GlobalScope.launch(Dispatchers.Main) {
      println("This is executed before the delay")
      stallForTime()
      println("This is executed after the delay")
    }

    println("This is executed immediately")
  }

  private suspend fun stallForTime() {
    withContext(Dispatchers.Default) {
      delay(2000L)
```

```
    }
  }
}
```

`onCreate()` is called on the main application thread, which `Dispatchers.Main` also uses. You might think that "This is executed before the delay" would get printed before "This is executed immediately", since we are launching a coroutine onto the same thread that we are on.

That is not the case. "This is executed immediately" is printed first, followed by "This is executed before the delay", just as it is in the Klassbook.

…Except When It Is Not

Android developers can think of `launch(Dispatchers.Main)` as being a bit like `post()` on `Handler` or `View`. Those methods take a `Runnable` and add it to the work queue for the main application thread. So, even if you are on the main application thread when you call `post()`, your `Runnable` is still run later, not immediately. Similarly, the coroutine specified by `launch(Dispatchers.Main)` is run later, not immediately.

There is a specialized dispatcher, `Dispatchers.Main.immediate`, that may be available in your environment. If the caller happens to be on the main application thread at the time of launching the coroutine, the coroutine is executed immediately, rather than being put into a bucket of coroutines to run later.

Android developers can think of `launch(Dispatchers.Main.immediate)` as being a bit like `runOnUiThread()` on `Activity`. This *looks* like it works like `post()`, but there is a subtle implementation distinction in Android:

- `post()` always puts the `Runnable` on the work queue.
- `runOnUiThread()` sees if we are currently executing on the main application thread and runs the `Runnable` immediately if that is the case. If we are currently executing on some other thread, `runOnUiThread()` works like `post()` and puts the `Runnable` on the work queue.

Similarly:

- `Dispatchers.Main` always runs the coroutine later
- `Dispatchers.Main.immediate` might run the coroutine synchronously, if you happen to be on the main application thread at the time of launching

the coroutine

`Dispatchers.Main.immediate` **is only available on Kotlin/JVM**. You will not be able to use it in Kotlin/JS or Kotlin/Native.

Some Context for Coroutines

Let's go back to the first coroutine example:

```kotlin
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the delay")
    stallForTime()
    println("This is executed after the delay")
  }

  println("This is executed immediately")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

(from ["A Simple Coroutine Sample" in the Klassbook](#))

Our call to `launch()` passes in a dispatcher, specifically `Dispatcher.Main`. That is a shorthand mechanism for passing in a `CoroutineContext`.

A `CoroutineContext` describes the environment for the coroutine that we are looking to create.

Contents of `CoroutineContext`

A `CoroutineContext` holds a series of elements. The two that we tend to focus on are:

- The dispatcher that is used for executing the coroutine
- The `Job` that represents the coroutine itself

In reality, a `CoroutineContext` has a key-value store of `Element` objects, keyed by a class. This allows `CoroutineContext` to be extended by libraries without having to create custom subclasses — a library can just add its own `Element` objects to the context as needed.

Where Contexts Come From

A couroutine builder creates a new `CoroutineContext` as part of launching the coroutine. The coroutine builder inherits an existing context (if there is one) from some outer scope, then overrides various elements. For example, if you pass a dispatcher to `launch()`, that dispatcher goes into the new context.

The `withContext()` function creates a new `CoroutineContext` for use in executing its code block, overriding whatever elements (e.g., dispatcher) that you provide.

Why Do We Care?

Most likely the details of a `CoroutineContext` do not matter to you. They are just part of the overall coroutines system, and so long as you provide the right dispatcher at the right time, the rest takes care of itself.

If you wish to *extend* the coroutines system, though, you might care about `CoroutineContext`.

For example, if you are creating a new type of `CoroutineScope` — as we will explore in a later chapter — you will wind up working a bit with `CoroutineContext`.

Suspending Function Guidelines

When using coroutines, many of your functions will wind up with the suspend keyword. If you are using an IDE like Android Studio or IntelliJ IDEA, the IDE will warn you when a suspend function is and is not needed, which helps a lot for determining where suspend is needed.

However, at some point, you are going to need to think a bit about where suspend belongs and does not belong. This chapter outlines some basic guidelines to consider.

DO: Suspend Where You Need It

suspend will be needed when you are calling some other function that is marked with suspend, and you are not making that call within a coroutine builder's lambda expression.

Failing to add the suspend keyword where one is required will result in a compiler error, and inside of your IDE that may give you an in-editor warning about the mistake:

```
    fun getCurrentObservation() = api.getCurrentObservation(STATION)
}
              Suspend function 'getCurrentObservation' should be called only from a coroutine or another suspend function
```

Figure 8: Android Studio Error About Missing suspend Keyword

Conversely, having suspend on a function that does not need one may result in a compiler warning and IDE "lint" message:
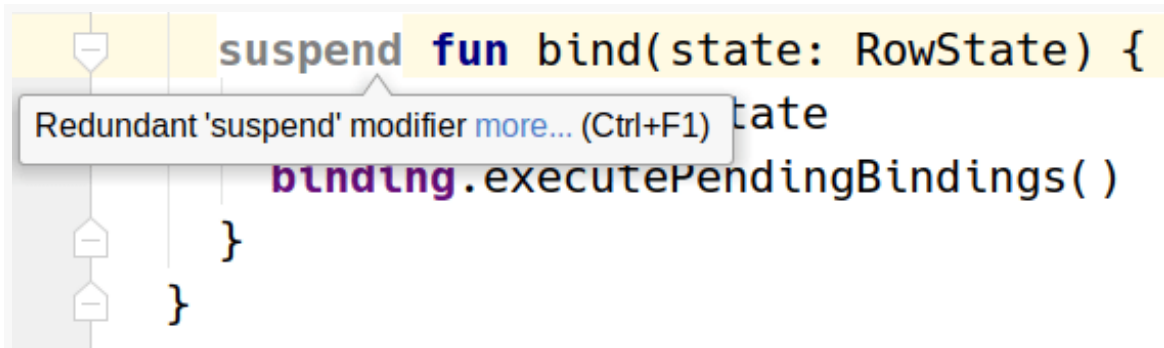
Figure 9: Android Studio Warning About Pointless suspend Keyword

DON'T: Create Them "Just Because"

It is easy to think that suspend makes the function run on a background thread.

It doesn't.

If you have code that you want to run on a background thread, typically you wrap it with withContext() and pass in a dispatcher indicating the thread/thread pool on which you want that code to run. withContext() is a suspend function, and so when you use it in your own function, that function will need to be marked with suspend or wrap the withContext() parts in a coroutine builder.

Focus on the coroutine builders and withContext() for controlling where your code runs. Then, add suspend as needed to make the compiler happy.

DO: Track Which Functions Are Suspending

Sometimes, you can set things up such that a particular class has an API that is completely suspend functions or is completely ordinary functions. In other cases, you will have an API that is a mix of ordinary and suspend functions. For example, you might have some sort of a repository object that usually offers a suspend API, but you have some regular functions as well:

- To support legacy code that is not moved over to coroutines yet
- To integrate with some Java-based library that will not be set up to use

coroutines
- To support synchronous calls made from callbacks from some framework where the background thread is already established (e.g., `ContentProvider` in Android)

To the developer of the API, the difference is obvious. To the consumer of the API, it may not be quite that obvious.

If everybody is using an IDE that offers up good in-editor warnings about missing `suspend` modifiers, you may be able to rely on that to help guide API consumers as to what is needed. Basically, they try calling your function, and if they get an error about a missing `suspend`, they then start thinking about where the coroutine will come from that will eventually control that `suspend` function call.

The problem comes from developers accidentally using the synchronous API and then winding up doing some unfortunate work on some unfortunate thread.

Consider using a naming convention to identify long-running non-`suspend` functions, to help consumers of those functions realize their long-running nature. For example, instead of `doWork()`, use `doWorkBlocking()` or `doWorkSynchronous()`.

DON'T: Block

A function marked with `suspend` should be safe to call from any thread. If that function has long-running work, it should be executed inside of a `withContext()` block and specify the dispatcher to use.

This does not mean that a function marked with `suspend` has to do *everything* on a designated background dispatcher. But a `suspend` function should return in microseconds, limiting its current-thread work to cheap calculations and little more.

This way, calling a `suspend` function from a coroutine set for `Dispatchers.Main` is safe. The `suspend` function is responsible for ensuring slow work is performed in the background; the coroutine is responsible for using the results of that background work on the "magic thread" associated with `Dispatchers.Main`.

Managing Jobs

In some cases, we need to control our coroutines a bit after we create them using a coroutine builder. For that, we have jobs: the output of a coroutine builder and our means of inspecting and managing the work while it is ongoing.

You Had One Job (Per Coroutine Builder)

Coroutine builders — `launch()` and `async()` — return a `Job`. The exact type will vary, as `Job` is an interface. In the case of `async()`, the return type is `Deferred`, an interface that extends from `Job`.

For `launch()`, there is no requirement to bother looking at this `Job`. You can treat `launch()` as a "fire and forget" coroutine builder. However, you can elect to hold onto this `Job` and use it for learning about the state of the coroutine that it represents.

Conversely, the only reason to use `async()` over `launch()` is to hold onto the `Deferred` that `async()` returns, so you can get the result of the coroutine — we will explore this more in the next chapter.

Contexts and Jobs

A `Job` is also an element of a `CoroutineContext`. When you call a coroutine builder, it sets up a new `CoroutineContext` and creates a new `Job` at the same time.

You can get the `Job` for your current `CoroutineContext` by using `coroutineContext[Job]` from the lambda expression passed to the coroutine builder or `withContext()`:

```
import kotlinx.coroutines.*

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main) {
    println("Job inside launch() context: ${coroutineContext[Job]}")
    stallForTime()
    println("This is executed after the delay")
  }

  println("Job returned by launch(): ${job}")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    println("Job inside withContext() context: ${coroutineContext[Job]}")
    delay(2000L)
  }
}
```

<div align="right">(from <u>"Contexts and Jobs" in the Klassbook</u>)</div>

This results in:

```
Job returned by launch(): StandaloneCoroutine{Active}@1
Job inside launch() context: StandaloneCoroutine{Active}@1
Job inside withContext() context: DispatchedCoroutine{Active}@2
This is executed after the delay
```

Though the actual object instance numbers after the @ may vary, you should find that the first two `println()` calls refer to the same object (a `StandaloneCoroutine`), while the third will refer to a different object (a `DispatchedCoroutine`).

Parents and Jobs

As contexts change, jobs can change as well. So, our `withContext()` call is not only adjusting the dispatcher that we are using, but it also sets up a separate `Job`.

This works because jobs are hierarchical. Each job can have children. `withContext()` is setting up a child job that manages the code block supplied to `withContext()`.

You can access a `children` property on `Job` that is a `Sequence` of the current children of that job:

```
import kotlinx.coroutines.*

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main) {
    println("Job inside launch() context: ${coroutineContext[Job]}")
    stallForTime(coroutineContext[Job]!!)
    println("Job after stallForTime(): ${coroutineContext[Job]}")
    println("This is executed after the delay")
  }

  println("Job returned by launch(): ${job}")
}

suspend fun stallForTime(parent: Job) {
  withContext(Dispatchers.Default) {
    println("Job inside withContext() context: ${coroutineContext[Job]}")
    println("Parent job children: ${parent.children.joinToString()}")
    delay(2000L)
  }
}
```

(from <u>"Jobs and Parents" in the Klassbook</u>)

This gives us:

```
Job returned by launch(): StandaloneCoroutine{Active}@1
Job inside launch() context: StandaloneCoroutine{Active}@1
Job inside withContext() context: DispatchedCoroutine{Active}@2
Parent job children: DispatchedCoroutine{Active}@2
Job after stallForTime(): StandaloneCoroutine{Active}@1
This is executed after the delay
```

Being Lazy on the Job

By default, when you use a coroutine builder, the Job that you are creating is "active". Kotlin and the underlying platform will begin doing the work defined by the lambda expression you provided to the coroutine builder, as soon as conditions warrant (e.g., there is a thread available in the dispatcher).

However, you can elect to pass CoroutineStart.LAZY to the coroutine builder. This will set up the Job but not yet make it active, so its work will not get dispatched until some time later.

Requesting Lazy Operation

Coroutine builders like launch() have an optional start parameter. If you pass CoroutineStart.LAZY as the start value, the coroutine will be set up in lazy mode:

```kotlin
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main, start = CoroutineStart.LAZY) {
    stallForTime()
    println("This is executed after the delay")
  }

  println("This is executed immediately")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

(from ["Lazy Coroutines" in the Klassbook](#))

If you run this sample, you will not see "This is executed after the delay", because that coroutine is never executed.

Making a Lazy Job Be Active

Usually, though, we want to eventually run the coroutine. Otherwise, why bother setting it up?

One way to start a lazy coroutine is to call start() on the Job:

```kotlin
import kotlinx.coroutines.*

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main, start = CoroutineStart.LAZY) {
    stallForTime()
    println("This is executed after the delay")
  }

  println("This is executed immediately")

  job.start()
```

```
  println("This is executed after starting the job")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

This time, we hold onto the Job returned by launch() and call start() on it later. The coroutine will then become active and, in short order, run and print its message.

The State of Your Job

A Job has three properties related to the state of execution of the Job:

- isCompleted will be true when the Job has completed all of its processing, successfully or unsuccessfully
- isCancelled will be true if the Job has been canceled, either directly or due to some exception
- isActive will be true while the Job is running and has not yet been canceled or completed (and has already been advanced to the active state after having been lazy, if applicable)

We will explore cancellation more in an upcoming section and exception handling later in the book.

Waiting on the Job to Change

As we saw with async(), await() is a blocking call on the Deferred that we got from async(). We get the result of the coroutine's lambda expression from await():

```
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    val deferred = GlobalScope.async(Dispatchers.Default) {
      delay(2000L)
      println("This is executed after the delay")
      1337
```

```
    }

    println("This is executed after calling async()")

    val result = deferred.await()

    println("This is the result: $result")
  }

  println("This is executed immediately")
}
```

<div align="right">(from <u>"async()" in the Klassbook</u>)</div>

Job itself does not have await(). Job does have join(). join() is also a blocking call, not returning until after the job has completed or been canceled. So, we can use join() if we need to eliminate the concurrency and tie up the current coroutine waiting for the launched block to complete:

```
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    val job = GlobalScope.launch(Dispatchers.Default) {
      delay(2000L)
      println("This is executed after the delay")
    }

    println("This is executed after calling launch()")

    job.join()

    println("This is executed after join()")
  }

  println("This is executed immediately")
}
```

<div align="right">(from <u>"join()" in the Klassbook</u>)</div>

Here, our output is:

```
This is executed immediately
This is executed after calling launch()
This is executed after the delay
This is executed after join()
```

Cancellation

Sometimes, we just do not want to do any more work.

With coroutines, we can try to cancel a `Job` and get out of doing its work. Whether this succeeds or not depends a bit on the coroutine, as we will see.

Canceling Jobs

There are many ways in which we can cancel a job, or have a job be canceled for us by the coroutines engine.

…Via `cancel()`

The typical proactive way to cancel a job is to call `cancel()` on it:

```
import kotlinx.coroutines.*

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the first delay")
    stallForTime()
    println("This is executed after the first delay")
  }

  GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the second delay")
    job.cancel()
    stallForTime()
    println("This is executed after the second delay")
  }

  println("This is executed immediately")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

(from ["Canceling a Job" in the Klassbook](#))

Here, our second coroutine calls `cancel()` on the job from the first coroutine. The

order of events is something like this:

- We `launch()` the first coroutine on the `Main` dispatcher
- We `launch()` the second coroutine on the `Main` dispatcher
- We print "This is executed immediately" and exit out of the `main()` function
- The `Main` dispatcher starts executing the first coroutine
- In that coroutine, we call `stallForTime()`, which is a suspend function, so the `Main` dispatcher elects to start executing the second coroutine
- The second coroutine calls `cancel()` on the first coroutine's `Job`, before it too calls `stallForTime()`
- The `Main` dispatcher switches back to the first coroutine
- `withContext()` or `delay()` detects that the coroutine was canceled and skips execution of that work, so that coroutine completes via cancellation
- The second coroutine continues through to completion

We will explore that second-to-last bullet, and the concept of "cooperative" cancellation, a bit more later in the chapter.

...Via `cancelAndJoin()`

A variation of `cancel()` is `cancelAndJoin()`. This cancels a job, but then blocks waiting for that job to complete its cancellation process:

```kotlin
import kotlinx.coroutines.*

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the first delay")
    stallForTime()
    println("This is executed after the first delay")
  }

  GlobalScope.launch(Dispatchers.Main) {
    println("This is executed at the start of the second coroutine")
    job.cancelAndJoin()
    println("This is executed before the second delay")
    stallForTime()
    println("This is executed after the second delay")
  }

  println("This is executed immediately")
}

suspend fun stallForTime() {
```

```
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

In this particular case, things work pretty much as they did in the previous example. That is because `withContext()` or `delay()` in `stallForTime()` will detect the cancellation and wrap up immediately.

But, pretend that `stallForTime()` would always run for two seconds, regardless of our cancellation request. In that case, the `cancelAndJoin()` call means that the second coroutine will take four seconds to complete:

- Two seconds blocking on waiting for the first coroutine to complete its cancellation
- Two seconds for its own delay

By contrast, the simple `cancel()` call does not block the caller, so the earlier example would not slow down the second coroutine.

…Via `cancel()` on the Parent

If you cancel a job, all of its child jobs get canceled as well.

In this snippet, we use `coroutineContext[Job]` to get the `Job` associated with our `launch()` coroutine builder, and we pass that `Job` to `stallForTime()`:

```kotlin
import kotlinx.coroutines.*

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the delay")
    stallForTime(coroutineContext[Job]!!)
    println("This is executed after the delay")
  }

  println("This is executed immediately")
}

suspend fun stallForTime(parent: Job) {
  withContext(Dispatchers.Default) {
    println("This is executed at the start of the child job")
```

```
    parent.cancel()
    println("This is executed after canceling the parent")
    delay(2000L)
    println("This is executed at the end of the child job")
  }
}
```

`stallForTime()` then calls `cancel()` on that parent `Job` before the `delay()` call.

What we get is:

```
This is executed immediately
This is executed before the delay
This is executed at the start of the child job
This is executed after canceling the parent
```

When we cancel the parent, both that job and our `withContext()` child job are canceled. When we call `delay()` in the `withContext()` job, the coroutines system sees that our job was canceled, so it abandons execution, so we never get the "This is executed at the end of the child job". Similarly, the coroutines system sees that the parent job was canceled while it was blocked waiting on `stallForTime()` to return, so it abandons execution of that job too, so we never see the "This is executed after the delay" message.

Note that the inverse is not true: canceling a child job does *not* cancel its parent.

…Via an Exception

If the code in a coroutine has an unhandled exception, its job gets canceled:

```kotlin
import kotlinx.coroutines.*

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the first delay")
    stallForTime()
    println("This is executed after the first delay")
  }

  GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the second delay")

    var thisIsReallyNull: String? = null

    println("This will result in a NullPointerException: ${thisIsReallyNull!!.length}")
```

```
    stallForTime()
    println("This is executed after the second delay")
  }

  println("This is executed immediately")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

Here, we have two independent jobs. The second job generates a
NullPointerException, so it will be canceled at that point, skipping the rest of its
work. The first job, though, is unaffected, since it is not related to the second job.
Hence, we get:

```
This is executed immediately
This is executed before the first delay
This is executed before the second delay
NullPointerException
This is executed after the first delay
```

(where NullPointerException gets logged at the point of our exception)

…Via an Exception on a Child

In reality, what happens is that an unhandled exception cancels the *parent* job,
which in turn cancels any children. It so happens that there was no parent job in the
preceding example, so only the one job was affected.

In this sample, though, we have a parent and a child, using the same basic code that
we saw previously:

```
import kotlinx.coroutines.*

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the delay")
    stallForTime()
    println("This is executed after the delay")
  }

  println("This is executed immediately")
}

suspend fun stallForTime() {
```

61

```
  withContext(Dispatchers.Default) {
    println("This is executed at the start of the child job")

    var thisIsReallyNull: String? = null

    println("This will result in a NullPointerException: ${thisIsReallyNull!!.length}")

    delay(2000L)
    println("This is executed at the end of the child job")
  }
}
```

<div align="right">(from <u>"Parent Jobs Cancel When a Child Crashes" in the Klassbook</u>)</div>

This time, rather than the child canceling the parent directly, the child crashes with a `NullPointerException`. However, the effect is more or less the same as if the child job had canceled the parent: the parent job is canceled, which in turn cancels the child job. And our results are the same as before, just with the extra `NullPointerException` log message from the exception:

```
This is executed immediately
This is executed before the delay
This is executed at the start of the child job
NullPointerException
```

…Via `withTimeout()`

Sometimes, we need to stop long-running work because it is running too long.

For that, we can use `withTimeout()`. `withTimeout()` creates a child job, and it cancels that job if it takes too long. You provide the timeout period in milliseconds to the `withTimeout()` call.

```
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    withTimeout(2000L) {
      println("This is executed before the delay")
      stallForTime()
      println("This is executed after the delay")
    }

    println("This is printed after the timeout")
  }

  println("This is executed immediately")
}
```

```
suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(10000L)
  }
}
```

(from ["Timeouts" in the Klassbook](#))

Here, we time out after two seconds, for a `delay()` of ten seconds. Ten seconds exceeds our timeout period, so the job created by `withTimeout()` is canceled, and we never see our "This is executed after the delay" message.

We also do not see the "This is printed after the timeout" message, though.

The reason for that is because `withTimeout()` throws a `TimeoutCancellationException`. That exception gets handled inside of `launch()` by default. Since cancellation is considered a normal thing to do, the `TimeoutCancellationException` does not trigger a crash. However, since `withTimeout()` threw an exception, the rest of our code inside of `launch()` gets skipped.

You can elect to handle the `TimeoutCancellationException` yourself, if you need to perform some post-timeout cleanup:

```
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    try {
      withTimeout(2000L) {
        println("This is executed before the delay")
        stallForTime()
        println("This is executed after the delay")
      }
    } catch (e: TimeoutCancellationException) {
      println("We got a timeout exception")
    }

    println("This is printed after the timeout")
  }

  println("This is executed immediately")
}
```

```
suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(10000L)
  }
}
```

In this case, not only do we see the "We got a timeout exception" message, but we also see the "This is printed after the timeout" message. We handled the raised exception, so execution can proceed normally after our try/catch logic.

Another variation on withTimeout() is withTimeoutOrNull(). This will return either:

- The result of the lambda expression, if it completes before the timeout
- null, if the work took too long

```
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    val result = withTimeoutOrNull(2000L) {
      println("This is executed before the delay")
      stallForTime()
      println("This is executed after the delay")
      "hi!"
    }

    println("This is the result: $result")
  }

  println("This is executed immediately")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(10000L)
  }
}
```

So, in this case, since our work exceeds our timeout period, we get a null result. If you raise the timeout limit or reduce the duration of the delay() call, such that the work can complete before the timeout period elapses, you will get "hi" as the result.

Supporting Cancellation

One of the key points behind coroutines is that they are cooperative. Whereas threads get control yanked from them when the OS feels like it, coroutines only lose control at points where it is deemed safe: when suspend functions get called.

Similarly, coroutines are cooperative with respect to cancellation. It is entirely possible to create a coroutine that is oblivious to a cancellation request, but that is not a particularly good idea. Instead, you want your coroutines to handle cancellation quickly and gracefully most of the time. There are a few ways of accomplishing this.

As an example of "not a particularly good idea", welcome to busyWait():

```kotlin
import kotlinx.coroutines.*
import kotlin.js.Date

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the first delay")
    stallForTime()
    println("This is executed after the first delay")
  }

  println("This is executed immediately")
}

suspend fun busyWait(ms: Int) {
  val start = Date().getTime().toLong()

  while ((Date().getTime().toLong() - start) < ms) {
    // busy loop
  }
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    coroutineContext[Job]!!.cancel()
    println("This is executed before the busyWait(2000) call")
    busyWait(2000)
    println("This is executed after the busyWait(2000) call")
  }
}
```

(from ["Canceling and Cooperation" in the Klassbook](#))

Here, we replace delay() with busyWait(). As the function name suggests, it implements a busy-wait, iterating through a tight while loop until the desired amount of time has elapsed. This is not a particularly good idea on pretty much every platform and language, let alone Kotlin/JS running in the browser via the Klassbook.

(note that busyWait() relies on kotlin.js.Date, so this snippet will not run on Kotlin/JVM or Kotlin/Native without modification)

Not only is busyWait() very inefficient from a CPU utilization standpoint, it does not cooperate with cancellation. This snippet's stallForTime() function cancels its own job immediately after starting it. However, cancel() itself does not throw a CancellationException, so execution continues in our lambda expression. Ideally, something in that lambda expression detects the cancellation request in a timely fashion... but that will not happen here. Neither println() nor busyWait() are paying any attention to coroutines, and so our output is:

```
This is executed immediately
This is executed before the first delay
This is executed before the busyWait(2000) call
This is executed after the busyWait(2000) call
```

The "This is executed after the first delay" message is not displayed because withContext() realizes (too late) that the job was canceled and throws the CancellationException, so we skip past the final println() in our launch() code.

So, with the anti-pattern in mind, let's see how we can improve busyWait(), at least a little bit.

Explicitly Yielding

A simple solution is to call yield() periodically:

```kotlin
import kotlinx.coroutines.*
import kotlin.js.Date

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the first delay")
    stallForTime()
    println("This is executed after the first delay")
  }
```

```
  println("This is executed immediately")
}

suspend fun busyWait(ms: Int) {
  val start = Date().getTime().toLong()

  while ((Date().getTime().toLong() - start) < ms) {
    yield()
  }
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    coroutineContext[Job]!!.cancel()
    println("This is executed before the busyWait(2000) call")
    busyWait(2000)
    println("This is executed after the busyWait(2000) call")
  }
}
```

(from ["Cooperation by Yielding" in the Klassbook](#))

Here we have the same code as before, except that inside the busy loop, we call yield() instead of nothing.

yield() does two things:

1. Since it is a suspend function, it signals to the coroutines system that it is safe to switch to another coroutine from this dispatcher, if there is one around that is ready to run
2. It throws CancellationException if the job was canceled

In our case, we only have this one coroutine, so the first feature is unused. However it also means that busyWait() will throw CancellationException as soon as the job is canceled. And, since the job was canceled before the call to busyWait(), busyWait() will go one pass through the while loop, then throw CancellationException, courtesy of the yield() call.

Checking for Cancellation

Another option for code with access to a CoroutineScope is to check the isActive property:

```kotlin
import kotlinx.coroutines.*
import kotlin.js.Date

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main) {
    println("This is executed before the first delay")
    stallForTime()
    println("This is executed after the first delay")
  }

  println("This is executed immediately")
}

suspend fun busyWait(ms: Int) {
  val start = Date().getTime().toLong()

  while ((Date().getTime().toLong() - start) < ms) {
    // busy loop
  }
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    coroutineContext[Job]!!.cancel()

    if (isActive) {
      println("This is executed before the busyWait(2000) call")
      busyWait(2000)
      println("This is executed after the busyWait(2000) call")
    }
  }
}
```

(from ["Cooperation by Checking isActive" in the Klassbook](#))

Here, `stallForTime()` checks `isActive` before making any calls to `println()` or `busyWait()`. Since we canceled the job, `isActive` will be `false`. `isActive` is available to us directly because the lambda expression passed to `withContext()` is executed with the `CoroutineScope` as its receiver, so `this` is the `CoroutineScope`.

Preventing Cancellation

Sometimes, you have code that simply cannot afford to be canceled. This should be the exception, not the rule… which is why the best example of this scenario is a `catch` or `finally` block. If you are cleaning things up from a crash, or are freeing resources from some work in a `try` block, you want all of your work to proceed, even

68

if the job you are in had been canceled already.

To do this, wrap your code in a `withContext(NonCancellable)` block. This ignores any prior cancellations.

```kotlin
import kotlinx.coroutines.*

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main) {
    try {
      println("This is executed before the first delay")
      stallForTime()
      println("This is executed after the first delay")
    }
    finally {
      withContext(NonCancellable) {
        println("This is executed before the finally block delay")
        stallForTime()
        println("This is executed after the finally block delay")
      }
    }
  }

  GlobalScope.launch(Dispatchers.Main) {
    println("This is executed at the start of the second coroutine")
    job.cancelAndJoin()
    println("This is executed before the second delay")
    stallForTime()
    println("This is executed after the second delay")
  }

  println("This is executed immediately")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(2000L)
  }
}
```

(from ["Non-Cancellable Coroutines" in the Klassbook](#))

This is similar to the `cancelAndJoin()` sample from before. This time, though, we wrap the first coroutine code in a `try/finally` structure, and we have another `stallForTime()` in the `finally`... but wrapped in `withContext(NonCancellable)`.

This gives us:

```
This is executed immediately
This is executed before the first delay
This is executed at the start of the second coroutine
This is executed before the finally block delay
This is executed after the finally block delay
This is executed before the second delay
This is executed after the second delay
```

The second coroutine calls `cancelAndJoin()` on the `Job` from the first coroutine. When that `Job` is canceled, `delay()` throws a `CancellationException`, so the `finally` block is executed. If we did not have the `withContext(NonCancellable)` bit in the `finally` block, the second `stallForTime()` would fail fast, once `delay()` sees that the job was canceled. However, `withContext(NonCancellable)` suppresses that behavior, so the `stallForTime()` in the `finally` block takes the full two seconds. And, since that `finally` block is part of the first coroutine and its job, the second coroutine blocks on its `cancelAndJoin()` call until the two-second delay completes.

Finding Out About Cancellation

If you need to know whether your job has been canceled from some specific piece of code inside of the job, you can catch the `CancellationException`. Alternatively, you could catch a specific subclass, such as `TimeoutCancellationException`, as we did in one of the `withTimeout()` samples:

```kotlin
import kotlinx.coroutines.*

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    try {
      withTimeout(2000L) {
        println("This is executed before the delay")
        stallForTime()
        println("This is executed after the delay")
      }
    } catch (e: TimeoutCancellationException) {
      println("We got a timeout exception")
    }

    println("This is printed after the timeout")
  }

  println("This is executed immediately")
```

```
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(10000L)
  }
}
```

However, that is only practical within the coroutine code itself. If you need to react from outside of the job, you can call invokeOnCompletion() on the Job and register a CompletionHandler, typically in the form of a lambda expression:

```
import kotlinx.coroutines.*

fun main() {
  val job = GlobalScope.launch(Dispatchers.Main) {
    withTimeout(2000L) {
      println("This is executed before the delay")
      stallForTime()
      println("This is executed after the delay")
    }
  }

  job.invokeOnCompletion { cause -> println("We were canceled due to $cause") }

  println("This is executed immediately")
}

suspend fun stallForTime() {
  withContext(Dispatchers.Default) {
    delay(10000L)
  }
}
```

You will be passed a cause value that will be:

- null if the job completed normally
- a CancellationException (or subclass) if the job was canceled
- some other type of exception if the job failed

In this case, we timed out, so we get a TimeoutCancellationException:

```
This is executed immediately
This is executed before the delay
We were canceled due to TimeoutCancellationException: Timed out waiting for 2000 ms
```

However, the `invokeOnCompletion()` lambda expression is in a somewhat precarious state:

- If it throws an exception, it might screw stuff up
- It should not take significant time
- It might be executed on any thread

So, mostly, this is for lightweight cleanup or possibly some form of error logging (if you get a `cause` that is something other than `null` or a `CancellationException`).

Deferring Results

This chapter will added in a future edition of this book (honest!).

Intermediate Coroutines

Structuring Concurrency

This chapter will be added in a future version of this book!

Working with Flows

We were introduced to flows [earlier in the book](), but we have lots more to learn about `Flow` behavior. So, this chapter will delve more deeply into `Flow` objects, including how we get them and how we leverage them.

Getting a `Flow`

The three main ways to get a `Flow` are:

- Calling a top-level function from the Kotlin coroutines system,
- Converting a `Channel` to a `Flow`, or
- Getting a `Flow` from a third-party library

We already saw `flow()` [in an earlier chapter](), so let's explore some of the other options.

flowOf()

`flowOf()` takes a `vararg` number of parameters and emits them one at a time for you. So, you just provide the objects:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    randomPercentages().collect { println(it) }
    println("That's all folks!")
  }
```

```
  println("...and we're off!")
}

fun randomPercentages() =
  flowOf(Random.nextInt(1,100), Random.nextInt(1,100), Random.nextInt(1,100))
```

<div align="right">(from <u>"flowOf()" in the Klassbook</u>)</div>

Here, we pre-generate three random numbers, create a `Flow` of those via `flowOf()`, and `collect()` that `Flow`.

`flowOf()` might seem pointless. After all, if we already have the objects, why do we need to bother with a `Flow`? However, there are a few cases where that is indeed what we want, such as:

- Testing, where we need to provide a `Flow` and we already have our test data to provide
- Error conditions, where we have our error object already, but the API that we are implementing requires a `Flow` (perhaps of a `sealed` class representing loading/content/error states)

emptyFlow()

`emptyFlow()` is basically `flowOf()` with no parameters. It returns a flow that is already closed, so a `collect()` call will just return immediately.

This is a bit more explicit than is `flowOf()`, for cases where you need a `Flow` for API purposes but will never emit any objects. But, like `flowOf()`, mostly it is for scenarios like testing.

Channels and Flows

A certain type of channel, called a `BroadcastChannel`, can be converted into a `Flow` via `asFlow()`. We will explore that more <u>later in the book</u>.

The `callbackFlow()` and `channelFlow()` top-level functions create `Flow` objects whose items come from a `produce()`-style `Channel`. These are designed specifically to support bridging streaming callback-style APIs into the world of coroutines. We will explore those more <u>in an upcoming chapter</u>.

Library-Supplied Flows

In many cases, you will not create a `Flow` yourself. Instead, you will get it from some library that offers a `Flow`-based API.

For example, in Android app development, Room supports `Flow` as a return type for a `@Query`-annotated function, just as it supports `LiveData`, `Observable` and `Flowable` from RxJava, etc.

Consuming a Flow

We have seen that one way to consume the `Flow` is to call `collect()` on it. While that will be fairly common, it is not your only option.

Functions like `collect()` that consume a flow are called "terminal operators" — we will see non-terminal operators [later in the chapter](). So, let's look at some other terminal operators.

`single()` **and** `singleOrNull()`

Sometimes, you may wind up with a `Flow` where you are only expecting one object to be emitted... and perhaps not even that. This would be the case where perhaps a library's API always returns a `Flow`, but you know that the situation will never result in a stream of multiple objects.

`single()` returns the first object emitted by the `Flow`. If the `Flow` is closed before emitting an object, or if the `Flow` is not closed after emitting the one-and-only expected object, `single()` throws an exception.

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    println(randomPercentage().single())
    println("That's all folks!")
  }

  println("...and we're off!")
}
```

```kotlin
fun randomPercentage() = flow {
  emit(Random.nextInt(1,100))
}
```

Here, our revised `Flow` emits just one random number, which we print. `single()` is a suspend function, which is why we are calling it from within a `launch()` lambda expression.

A slightly safer version of `single()` is `singleOrNull()`. This:

- Returns `null` for an empty `Flow` (one that closes without emitting anything)
- Returns the emitted object for a single-object `Flow`
- Throws an exception if the `Flow` does not close after emitting its first object

However, `singleOrNull()` will return a nullable type. If you have a `Flow<String>`, `singleOrNull()` returns `String?`, not `String`.

first()

`first()` is reminiscent of `single()` and `singleOrNull()`, in that it returns one object from the `Flow`. However, it then stops observing the `Flow`, so it is safe to use with a `Flow` that might return more than one value.

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    println("received ${randomPercentages(100, 200).first()}")
    println("That's all folks!")
  }

  println("...and we're off!")
}

fun randomPercentages(count: Int, delayMs: Long) = flow {
  for (i in 0 until count) {
    delay(delayMs)

    val value = Random.nextInt(1,100)

    println("emitting $value")
```

```
    emit(value)
  }
}
```

Here, we request 100 random numbers... but only take one via `first()`, then abandon the `Flow`. We print each of our emissions, and if you run this, you will see that `emit()` only winds up being called once:

```
...and we're off!
emitting 62
received 62
That's all folks!
```

When we stop observing the `Flow`, the `Flow` is considered to be closed, so it cancels the job that our `flow()` lambda expression runs in, and we exit cleanly.

`toList()` and `toSet()`

Calling `toList()` or `toSet()` collects all of the objects emitted by the `Flow` and returns them in a `List` or `Set`:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    println(randomPercentages(10, 200).toList())
    println("That's all folks!")
  }

  println("...and we're off!")
}

fun randomPercentages(count: Int, delayMs: Long) = flow {
  for (i in 0 until count) {
    delay(delayMs)
    emit(Random.nextInt(1,100))
  }
}
```

Here, we collect our ten random numbers in a `List`, then print that result:

```
...and we're off!
[15, 12, 31, 28, 34, 11, 65, 85, 71, 78]
That's all folks!
```

These functions handle all bounded flows: zero items, one item, or several items. However, they only work for flows that will close themselves based on some internal criterion. These functions will not work well for flows that might emit objects indefinitely.

As with `single()` and the other terminal operators, `toList()` and `toSet()` are `suspend` functions.

Flows and Dispatchers

Typically, a `Flow` will take some amount of time to do its work, and there may be additional external delays. For example, a `Flow` that is emitting objects based on messages received over an open WebSocket has no idea how frequently the server will send those messages.

So far, we have not dealt with this. We have launched our collectors using `Dispatchers.Main`, and we have not otherwise specified a dispatcher. Flows do not wind up on some other dispatcher by magic. Instead, they use the same dispatcher mechanism as we use for the rest of coroutines.

suspend **Function and** withContext()

If your `Flow` calls a `suspend` function, that function is welcome to use `withContext()` to switch to a different dispatcher:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    flow {
      for (i in 0 until 10) {
        emit(randomPercentage(200))
      }
    }.collect { println(it) }
    println("That's all folks!")
  }

  println("...and we're off!")
}
```

```
suspend fun randomPercentage(delayMs: Long) = withContext(Dispatchers.Default) {
  delay(delayMs)
  Random.nextInt(1,100)
}
```

<div align="right">(from <u>"suspend Functions and Flows" in the Klassbook</u>)</div>

Here, we move the "slow" work to a simple randomPercentage() function that does that work on Dispatchers.Default. Our Flow can call randomPercentage() and emit() the values that it returns without issue.

withContext() **in** flow()

You could get rid of the suspend function and just use withContext() from inside the flow() lambda expression itself:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    flow {
      for (i in 0 until 10) {
        val randomValue = withContext(Dispatchers.Default) {
          delay(200)
          Random.nextInt(1,100)
        }
        emit(randomValue)
      }
    }.collect { println(it) }
    println("That's all folks!")
  }

  println("...and we're off!")
}
```

This was not supported originally with Flow, but it appears to work with coroutines version 1.3.5 and newer.

flowOn()

Another option is to use flowOn() to customize the Flow:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    flow {
      for (i in 0 until 10) {
        delay(200)
        emit(Random.nextInt(1,100))
      }
    }
    .flowOn(Dispatchers.Default)
    .collect { println(it) }

    println("That's all folks!")
  }

  println("...and we're off!")
}
```

(from ["flowOn()" in the Klassbook](#))

Now, the lambda expression will run on `Dispatchers.Default`, but our collection of the results will be performed on `Dispatchers.Main` (courtesy of the `launch()` dispatcher).

Flows and Actions

You can attach listeners — frequently in the form of lambda expressions — to a `Flow` to find out about events in the flow's lifecycle.

Two of these are for monitoring the ongoing operation of the `Flow`:

- `onEach()`, to be notified about each object that is emitted, separately from what the collector collects
- `onCompletion()`, to be notified when the `Flow` is closed and will not emit any more objects

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
```

```kotlin
    randomPercentages(10, 200)
      .collect { println(it) }
  }

  println("...and we're off!")
}

fun randomPercentages(count: Int, delayMs: Long) = flow {
  for (i in 0 until count) {
    delay(delayMs)
    emit(Random.nextInt(1,100))
  }
}
  .onEach { println("about to collect $it") }
  .onCompletion { println("That's all folks!") }
```

(from ["Flow Actions" in the Klassbook](#))

This gives us:

```
...and we're off!
about to collect 24
24
about to collect 20
20
about to collect 67
67
about to collect 13
13
about to collect 77
77
about to collect 79
79
about to collect 36
36
about to collect 51
51
about to collect 3
3
about to collect 37
37
That's all folks!
```

Another, `onStart()`, allows you to inject new values before the "real" content emitted by the `Flow`:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    randomPercentages(10, 200)
      .onStart { emit(1337) }
      .collect { println(it) }
    println("That's all folks!")
  }

  println("...and we're off!")
}

fun randomPercentages(count: Int, delayMs: Long) = flow {
  for (i in 0 until count) {
    delay(delayMs)
    emit(Random.nextInt(1,100))
  }
}
```

(from ["Flow onStart()" in the Klassbook](#))

The lambda expression (or other function type) supplied to onStart() gets a
FlowCollector as a parameter. This is the same object supplied to flow(), and it has
the emit() function that we have been using. Anything you emit() from onStart()
will be emitted by the Flow first. So, in this sample, the ten random numbers are
preceded by a non-random value of 1337:

```
...and we're off!
1337
5
94
32
71
97
60
51
21
99
33
That's all folks!
```

Flows and Other Operators

Flow has a bunch of methods that are reminiscent of Sequence or List, that perform operations on what the Flow emits. These include:

- filter()
- filterNot()
- filterNotNull()
- fold()
- map()
- mapNotNull()
- reduce()

Flow also supports a number of operators that help you deal with more than one Flow, such as:

- flatMapConcat() and flatMapMerge(), where you supply a function or lambda expression that turns each item from the original Flow into a new Flow (e.g., making a Web service request for each item retrieved from a local database)
- zip() to combine the outputs of two Flow objects into a single Flow

Flows and Exceptions

It is entirely possible that something will go wrong with our Flow processing, resulting in an exception.

Default Behavior

The default behavior of Flow mirrors what happens with a suspend function that throws an exception: you can just catch it:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    val myFlow = randomPercentages(10, 200)

    try {
```

```
      myFlow.collect { println(it) }
    } catch (ex: Exception) {
      println("We crashed! $ex")
    }

    println("That's all folks!")
  }

  println("...and we're off!")
}

fun randomPercentages(count: Int, delayMs: Long) = flow {
  for (i in 0 until count) {
    delay(delayMs)
    emit(Random.nextInt(1,100))
    throw RuntimeException("ick!")
  }
}
```

(from ["Flows and Exceptions" in the Klassbook](#))

The collect() call will throw the exception raised by the Flow, and you can use an ordinary try/catch structure to handle it. So, here, we get:

```
...and we're off!
18
We crashed! RuntimeException: ick!
That's all folks!
```

A lot of the time, this approach is sufficient.

retry() **and** retryWhen()

Sometimes, though, we want to retry the work being done by the Flow in case of an exception. For example, with network I/O, some exceptions represent random failures, such as the user of a mobile device switching from WiFi to mobile data mid-way through a download. We might want to retry the download a couple of times, and if it still does not succeed, then report the problem to the user.

For this, Flow offers retry() and retryWhen().

Roughly speaking, there are three main patterns for using those operators.

retry(N), for a given value of N, will just blindly retry the Flow that number of times. If it continues to fail after the N tries, then the exception proceeds normally:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    val myFlow = randomPercentages(10, 200)

    try {
      myFlow
        .retry(3)
        .collect { println(it) }
    } catch (ex: Exception) {
      println("We crashed! $ex")
    }

    println("That's all folks!")
  }

  println("...and we're off!")
}

fun randomPercentages(count: Int, delayMs: Long) = flow {
  for (i in 0 until count) {
    delay(delayMs)
    emit(Random.nextInt(1,100))
    throw RuntimeException("ick!")
  }
}
```

(from ["retry()" in the Klassbook](#))

Here, we request three retries. Since we emit() an object each time before we throw the exception, this means that a total of four objects are emitted (one for the initial failure and one for each of the three retries):

```
...and we're off!
70
91
45
31
We crashed! RuntimeException: ick!
That's all folks!
```

You can optionally pass a lambda expression to retry(). This receives the exception as a parameter. Now the exception will continue to the FlowCollector if either:

- We have exceeded the maximum number of retries, or
- The lambda expression returns `false`

For example, you might want to retry if an `IOException` occurs (as that is an expected problem) but not retry for any other type of exception.

`retryWhen()` also takes a lambda expression, but it does not take a retry count. The lambda expression is passed both the exception and the current count of retries. If the lambda expression returns `true`, the `Flow` work is retried. If the lambda expression returns `false`, the exception proceeds to the `FlowCollector`:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    val myFlow = randomPercentages(10, 200)

    try {
      myFlow
        .retryWhen { ex, count -> count < 3 }
        .collect { println(it) }
    } catch (ex: Exception) {
      println("We crashed! $ex")
    }

    println("That's all folks!")
  }

  println("...and we're off!")
}

fun randomPercentages(count: Int, delayMs: Long) = flow {
  for (i in 0 until count) {
    delay(delayMs)
    emit(Random.nextInt(1,100))
    throw RuntimeException("ick!")
  }
}
```

(from ["retryWhen()" in the Klassbook](#))

This particular lambda expression happens to only look at the count, but a typical `retryWhen()` invocation would have a lambda expression that looked at the exception and made decisions from there.

catch()

Another operator is catch(). Like the catch in try/catch, catch() catches exceptions. However, it simply consumes them and stops the flow:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    val myFlow = randomPercentages(10, 200)

    myFlow
      .catch { println("We crashed! $it") }
      .collect { println(it) }

    println("That's all folks!")
  }

  println("...and we're off!")
}

fun randomPercentages(count: Int, delayMs: Long) = flow {
  for (i in 0 until count) {
    delay(delayMs)
    emit(Random.nextInt(1,100))
    throw RuntimeException("ick!")
  }
}
```

(from)

Opting Into `SharedFlow` **and** `StateFlow`

Originally, coroutines just had `suspend` functions.

Then, coroutines added channels, and later, flows, for inter-coroutine communications.

More recently, coroutines added a pair of specific types of flows — `SharedFlow` and `StateFlow` — to handle specific communications patterns. These get consumed like an ordinary `Flow`, but they are created differently and have distinct characteristics.

These are long-term replacements for things like `ConflatedBroadcastChannel` (which we will explore in [the next chapter](#)). `StateFlow` also likely will be a long-term replacement for `LiveData` in Android app development, as we will explore [later in the book](#).

So, in this chapter, we will take a look at what `SharedFlow` and `StateFlow` offer and how we use them.

Hot and Cold, Revisited

Let's go back to the original `Flow` example, using `flow()` to build the `Flow`:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    randomPercentages(10, 200).collect { println(it) }
    println("That's all folks!")
```

```
  }

  println("...and we're off!")
}

fun randomPercentages(count: Int, delayMs: Long) = flow {
  for (i in 0 until count) {
    delay(delayMs)
    emit(Random.nextInt(1,100))
  }
}
```

(from ["A Simple Flow" in the Klassbook](#))

flow() creates a cold Flow. Nothing happens until something starts collecting the output of the Flow — in this case, the collect() call in main(). The events only exist if something is collecting them.

Moreover, flow() creates a unique cold Flow. If we call flow() three times, we get three distinct Flow objects. And each Flow created by flow() can only be collected once — we cannot have multiple collectors.

These are fairly severe limitations.

In particular, a lot of the time, we want to consume events that are created regardless of whether or not anything is around to observe them:

- User input events, such as keypresses or button clicks
- Clock ticks
- Sensor input events: accelerometer readings, temperatures, GPS fixes, etc.
- And so on

flow() is ill-suited for this. The Flow API is fine, but we need some other way of creating flows that can model this case better.

That is where SharedFlow and StateFlow come into play.

Introducing SharedFlow

SharedFlow is a hot flow that can have multiple consumers, with each consumer seeing the same sequence of emitted events as any other active consumer at the time.

Creating and Consuming a `SharedFlow`

The stock underlying implementation of `SharedFlow` is `MutableSharedFlow`. There is a `MutableSharedFlow()` factory function that creates instances of `MutableSharedFlow`. You can call `emit()` on the `MutableSharedFlow` to send an object to any current subscribers — if there are no subscribers, that object is simply ignored. `emit()` is a suspend function, so you need to call it from inside of a coroutine.

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

fun randomSharedFlow(): Flow<Int> {
  val sharedFlow = MutableSharedFlow<Int>()

  GlobalScope.launch(Dispatchers.Default) {
    for (i in 0 until 10) {
      sharedFlow.emit(Random.nextInt(1,100))
      delay(200)
    }
  }

  return sharedFlow
}

fun main() {
  val sharedFlow = randomSharedFlow()

  GlobalScope.launch(Dispatchers.Main) {
    sharedFlow.collect { println("Collector A: $it") }
    println("That's all folks!")
  }

  GlobalScope.launch(Dispatchers.Main) {
    sharedFlow.collect { println("Collector B: $it") }
    println("That's all folks!")
  }

  println("...and we're off!")
}
```

(from ["A Simple SharedFlow" in the Klassbook](#))

Here, `randomSharedFlow()` creates a `MutableSharedFlow`, then loops 10 times and emits a random number on that flow, delaying 200 milliseconds between random

numbers.

We set up two observers of that flow, each set to `collect()` what is emitted by the flow. If you run the sample, you will see that they each get the same random numbers, since this is one flow (and one set of random numbers) with two subscribers.

Limiting Exposure

To publish objects, you use a `MutableSharedFlow`. To consume, though, you only need either the `Flow` or `SharedFlow` interfaces. That way, you can better control who is able to publish by choosing which data type to expose.

In the sample shown above, `randomSharedFlow()` returns a `Flow`, as that is all that is needed. And, this way, we do not "leak" the ability to emit things on the underlying `MutableSharedFlow`.

Replay and Buffering

By default, `MutableSharedFlow` does not cache emitted objects. Everything "flows" through as it is received. However, it does have some options for caching, as part of the `MutableSharedFlow()` factory function.

The `replay` parameter represents how many objects should be cached and delivered to new subscribers. The default is `0`, so a late subscriber only receives objects emitted after that subscription point. But, you can set this to a higher value if desired.

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

fun randomSharedFlow(): Flow<Int> {
  val sharedFlow = MutableSharedFlow<Int>(replay = 2)

  GlobalScope.launch(Dispatchers.Default) {
    for (i in 0 until 10) {
      sharedFlow.emit(Random.nextInt(1,100))
      delay(200)
    }
  }

  return sharedFlow
```

```
}

fun main() {
  val sharedFlow = randomSharedFlow()

  GlobalScope.launch(Dispatchers.Main) {
    sharedFlow.collect { println("Collector A: $it") }
    println("That's all folks!")
  }

  GlobalScope.launch(Dispatchers.Main) {
    delay(1000)
    sharedFlow.collect { println("Collector B: $it") }
    println("That's all folks!")
  }

  println("...and we're off!")
}
```

(from ["MutableSharedFlow Replay Option" in the Klassbook](#))

In this example, randomSharedFlow() is almost identical to the original. The difference is that replay is set to 2. Similarly, our subscribers are almost identical — the sole difference is that "B" delays its collect() call by 1000 milliseconds.

During those 1000 milliseconds, the MutableSharedFlow should emit 5 objects. Ordinarily, "B" would get none of those, as "B" does not subscribe until after that point in time. But, courtesy of replay = 2, "B" gets the last 2 of those objects, plus all new ones:

```
...and we're off!
Collector A: 41
Collector A: 71
Collector A: 67
Collector A: 82
Collector A: 29
Collector B: 82
Collector B: 29
Collector A: 64
Collector B: 64
Collector A: 6
Collector B: 6
Collector A: 53
Collector B: 53
Collector A: 36
```

```
Collector B: 36
Collector A: 31
Collector B: 31
```

`MutableSharedFlow()` also has `extraBufferCapacity` and `onBufferOverflow` parameters. These add additional buffer capacity beyond what `replay` calls for, but without the `replay` semantics. This additional buffer does not result in additional objects being delivered to late subscribers. Instead, this buffer is solely for dealing with slow subscribers (or, if you prefer, fast emitters).

The default behavior of `MutableSharedFlow()` is to have `extraBufferCapacity` set to `0` and to have `onBufferOverflow` to be set to `BufferOverflow.SUSPEND`. This means that a call to `emit()` will suspend until all subscribers collect the object that is being emitted. This is fine if those subscribers are fairly quick or if the process of emission can wait a while. If that is a problem, `extraBufferCapacity` adds a buffer that will be used to allow `emit()` to return right away, if there is buffer space for the object that we are trying to emit.

You can also pass a different value to `onBufferOverflow` to control the overflow strategy. The default, `SUSPEND`, suspends the `emit()` call until it can emit the object (or a buffer slot frees up). Alternatives include:

- `DROP_OLDEST`, which evicts the oldest entry from the buffer and adds the to-be-emitted object to the buffer
- `DROP_LATEST`, which evicts the newest entry from the buffer and adds the to-be-emitted object to the buffer

In those latter two cases, even if there is no buffer space, `emit()` will not suspend. However, in those latter two cases, we lose data.

Introducing `StateFlow`

A `StateFlow`, like a `SharedFlow`, can have multiple consumers, and it delivers objects to each consumer. And, a `StateFlow` is hot, existing regardless of whether there are any consumers at all. The difference comes with backpressure: while `SharedFlow` has a few configuration options for handling this, `StateFlow` offers just pure "conflation": replacing old values with new ones.

A `StateFlow` has a single-element buffer. In fact, if you are using stock implementations, a `StateFlow` *always* has a value — the act of creating a `StateFlow` requires an initial value. If another object is sent to the flow, it replaces the previous

value (including replacing the initial value for the very first sent object).

Consumers of a StateFlow will get the current value delivered to them immediately, plus any new objects sent into the flow while the consumer is active.

Also, any code with access to the StateFlow can reference a value property to get the now-current value, whether that is still the initial value or some other object that was sent into the flow.

This is ideal for state management, particularly when coupled with a sealed class:

- You initialize the StateFlow with a sealed class implementation that represents the initial state (e.g., "loading")
- You send items into the StateFlow that represent changes in that state (e.g., "content" or "error")
- Consumers can work with the now-current state immediately, plus update themselves with fresh states as they become available

Creating and Consuming a StateFlow

StateFlow is a read-only API. It has a value property that you can use to get the most-recent object placed onto the Flow, along with the standard slate of Flow operators.

But, if you want to actually put states into the StateFlow, you need a MutableStateFlow. This gets states in two ways:

- The MutableStateFlow() constructor-style factory function takes an initial state. This is not optional — you must provide an initial state.
- The value property on MutableStateFlow is mutable. To emit a new state, just assign it to value.

So, here is an example of a MutableStateFlow emitting some states:

```
import kotlinx.coroutines.*
import kotlinx.coroutines.flow.*

fun sourceOfStates(): Flow<String> {
  val states = MutableStateFlow<String>("Alabama")

  GlobalScope.launch(Dispatchers.Default) {
    listOf("Alaska", "Arizona", "Arkansas", "California",
           "Colorado", "Connecticut", "Delaware", "Florida", "Georgia",
           "Hawaii", "Idaho", "Illinois", "Indiana", "Iowa",
```

```
        "Kansas", "Kentucky", "Louisiana", "Maine", "Maryland",
        "Massachusetts", "Michigan", "Minnesota", "Mississippi", "Missouri",
        "Montana", "Nebraska", "Nevada", "New Hampshire", "New Jersey",
        "New Mexico", "New York", "North Carolina", "North Dakota", "Ohio",
        "Oklahoma", "Oregon", "Pennsylvania", "Rhode Island", "South Carolina",
        "South Dakota", "Tennessee", "Texas", "Utah", "Vermont",
        "Virginia", "Washington", "West Virginia", "Wisconsin", "Wyoming"
      ).forEach { state ->
    states.value = state
    delay(200)
  }
 }

 return states
}

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    sourceOfStates().collect { println(it) }
    println("That's all folks!")
  }

  println("...and we're off!")
}
```

(from ["A Simple StateFlow" in the Klassbook](#))

(note: your states do not have to be US states)

sourceOfStates() creates a MutableStateFlow that starts with an initial state ("Alabama"). Then, in a background thread, it updates the value to a new state every 200 milliseconds. In main(), we collect() that flow and print its output.

With the 200-millisecond delay, usually our collector will keep up, and so the output will have the list of the 50 US states. But, if you add a delay in the collector, so it cannot process states as quickly as they are emitted, you will find that it only collects a subset. The rest were lost as a result of conflation, when the new values replaced the old ones before those old ones wound up being observed.

Managing Exposure

As noted in the previous section, StateFlow is a read-only API, while MutableStateFlow adds the ability to emit states into the flow. As a result, you can design your API to control what code has write access:

- Expose the MutableStateFlow only to those objects that need the ability to emit states
- Expose StateFlow to objects that need to consume states and also get the last-emitted state on demand

- Expose Flow to objects that just need to consume states

In the sample shown above, sourceOfStates() returns a Flow, not a StateFlow, since main() is only using normal Flow functions.

Limitations

While StateFlow and MutableStateFlow are thread-safe, the API is designed around a single "owner" emitting states. While it is technically possible to have more than one parallel operation emit states onto a MutableStateFlow, there are no operators on MutableStateFlow to coordinate that sort of work.

Content Equality

With StateFlow, content equality determines how conflation works.

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

data class State(
  val number: Int
)

fun sourceOfStates(): Flow<State> {
  val states = MutableStateFlow<State>(State(123))

  GlobalScope.launch(Dispatchers.Default) {
    repeat(25) {
      delay(200)
      states.value = State(456)
    }
  }

  return states
}

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    sourceOfStates().collect { println(it) }
  }

  println("...and we're off!")
}
```

(from ["StateFlow and Content Equality" in the Klassbook](#))

Here, we put a total of 26 `State` objects into a `MutableStateFlow`. However, only the first and second differ in content. The remaining 24 are the same as the second (`State(123)`). As a result, our `collect()` lambda is only invoked twice: once for the original `State(456)` value, then once for the first `State(123)` value. The remaining 24 are ignored, since they are equal to the current value.

This comes back to the name: `StateFlow` is a flow of states. States are represented by their content, often in the form of a `data class` so we get the code-generated `equals()` to use. We should not need to re-apply logic in the `StateFlow` consumer if the state is not really changing (content of the old and new state are equal). So, `StateFlow` simply does not emit the replacement state object.

Comparing to `LiveData`

Android developers will look at `StateFlow` and `MutableStateFlow` and wonder if Kotlin developers just did a search-and-replace on the string "LiveData". There seems little doubt that some aspects of `StateFlow` were inspired by `LiveData`. However, there are some noteworthy differences.

Dispatcher Control

`LiveData` can be observed only on the main application thread. There is no form of `observe()` that takes a parameter (e.g., a `Looper` or `Executor`) that `LiveData` might use as a way to deliver results on another thread. This thread limitation is fine for "last mile" sorts of delivery to results to the UI layer, but it can be annoying in other cases. If, say, you have a Room database DAO that has a function that returns a `LiveData`, you need to try to avoid observing it until you get to the UI. Anything between the DAO and the UI needs to try to work with `Transformations` and `MediatorLiveData` to manipulate the `LiveData` stream while not *consuming* that stream and causing work to be done on the main application thread.

Contrast that with `StateFlow`, where you can use operators like `flowOn()` to control the dispatcher that gets used by default for downstream collectors of the emitted objects. By using `Dispatchers.Main`, you can get equivalent results as what you get from `LiveData`, with the flexibility of using other dispatchers where they may be needed.

More Operators

Compounding the problems of the previous segment is the fact that `LiveData` has

very few operators. We have `map()` and `switchMap()` on `Transformations`... and that is about it. Third-party libraries could, in theory, help fill the gap, but there has not been a lot of movement in this area.

`StateFlow`, on the other hand, is a `Flow`, so `StateFlow` gets access to a lot of operators "for free", such as `combine()`, `debounce()`, `filter()`, and `zip()`, along with `map()` and `switchMap()`.

Scope Flexibility

`LiveData` is aware of Android lifecycles. Anything that naturally has a lifecycle, such as activities and fragments, can work well with `LiveData`. In places where you lack a decent lifecycle, though, `LiveData` can be aggravating. You either need to concoct a `Lifecycle` implementation or use things like `observeForever()` and ensure that you do not leak observers.

Creating a `CoroutineScope` is simpler than is implementing a `Lifecycle`, and adding `CoroutineScope` semantics to existing APIs is easier than is grafting a `Lifecycle` on them.

The Missing Scenario

`SharedFlow` and `StateFlow` cover a lot of scenarios. One that they do not is the exactly-once scenario.

With `SharedFlow` and `StateFlow`, we can have 0 to N subscribers. Each object posted to those types of flows gets delivered to each of those subscribers. And, with `StateFlow`, the last-sent value is cached and delivered to future subscribers.

But, what happens if we have an event that we want to process exactly once? These types of flows are not well-suited for that case:

- If there are 2+ subscribers at the time of the event, we might process the event more than once
- If there are 0 subscribers to a `SharedFlow`, nothing will process the event — it will get "dropped on the floor"
- If there are 0 subscribers to a `StateFlow`, the event gets cached, so it might be processed by some later subscriber... but it might be replaced before that happens, or it might be processed by multiple later subscribers

Right now, JetBrains is recommending the lower-level Channel API for that. We explored channels a bit [earlier in the book](), and we will expand on that more [in the next chapter]().

Tuning Into Channels

We looked at the basic use of channels [earlier in the book](). And while channels are a bit "low level", particularly for the suppliers of data, channels still have their uses. So, in this chapter, we will explore some more details about the operation of channels.

A Tale of Three Interfaces

`Channel` is an interface. In itself, this is not very remarkable.

However, `Channel` extends two other interfaces:

- `SendChannel`, which contains the basic functions for putting data into a channel, such as `offer()` and `send()`
- `ReceiveChannel`, which contains the basic functions for getting data from a channel, such as `poll()` and `receive()`

You can use these separate interfaces to better compartmentalize your code. Even though you will create something that implements all of `Channel`, you can pass `SendChannel` to code that only needs send (not receive) and `ReceiveChannel` to code that only needs to receive (not send).

Creating a Channel

Back in the chapter on [basic flows and channels](), we saw creating a channel using `produce()`:

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    randomPercentages(10, 200).consumeEach { println(it) }
    println("That's all folks!")
  }

  println("...and we're off!")
}

fun CoroutineScope.randomPercentages(count: Int, delayMs: Long) = produce {
  for (i in 0 until count) {
    delay(delayMs)
    send(Random.nextInt(1,100))
  }
}
```

(from ["A Simple Channel" in the Klassbook](#))

This works if you are in position to provide a lambda expression that is the source of the objects to put into the channel. However, that is not always the case, so we have other options for creating channels.

Channel()

Another approach is `Channel()`:

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

fun main() {
  val channel = Channel<Int>(Channel.BUFFERED)

  GlobalScope.launch(Dispatchers.Default) {
    for (i in 0 until 10) {
      delay(200)
      channel.send(Random.nextInt(1,100))
    }

    channel.close()
  }

  GlobalScope.launch(Dispatchers.Main) {
```

```
    channel.consumeEach { println(it) }
    println("That's all folks!")
  }

  println("...and we're off!")
}
```

`Channel()` looks like a constructor... but `Channel` is an interface, not a class. So, `Channel()` is actually a top-level factory function that creates `Channel` objects of a particular type, based on its parameter.

That parameter indicates the nature of buffering of the channel. Here, we use `Channel.BUFFERED`, which means that the channel has a buffer of a default size (64 elements). If we call `send()` and the buffer has room, `send()` returns immediately after adding the object being sent to that buffer. If we call `send()` and the buffer is full, `send()` suspends until the consumer of the channel can receive objects and free up some buffer space. We will examine the full range of buffering options <u>later in the chapter</u>.

While here we have a trivial bit of code that just uses the resulting `Channel` in a single function, we could easily have cases where we:

- Create the `Channel`
- Have the creator hold onto the `Channel` and `send()` or `offer()` elements to it
- Have the creator make the channel available via the `ReceiveChannel` interface to other objects that will consume the channel contents

actor()

If you are developing in Kotlin/JVM — whether for Android or other JVM targets — you have an `actor()` top-level function that you can use. This inverts the `produce()` pattern:

- `produce()` takes a lambda expression that offers or sends objects and returns a `ReceiveChannel` for consuming those objects
- `actor()` takes a lambda expression that consumes objects from the channel and returns a `SendChannel` for offering or sending those objects

```
class ActorSample {
  val receivedItems: MutableList<Int> =
    Collections.synchronizedList(mutableListOf<Int>())

  fun actOut(scope: CoroutineScope) = scope.actor<Int>(Dispatchers.Default) {
    consumeEach { receivedItems.add(it) }
  }
}
```

Here, we have an ActorSample class with an actOut() function. actor() is an
extension function on a CoroutineScope, so actOut() takes a CoroutineScope as a
parameter. We pass in Dispatchers.Default to actor() to indicate that we want our
work to be done on that dispatcher's thread pool. And, we supply a lambda
expression to consume the objects. this in the scope of that lambda expression is an
ActorScope, which implements both CoroutineScope and ReceiveChannel. As a
result, we can just call consumeEach() to pull all the objects off of the supplied
channel until that channel is closed. In our case, since this code is for Kotlin/JVM,
we can use Collections.synchronizedList() to set up a thread-safe MutableList,
into which we collect the received values.

Our sending code can then use the SendChannel returned by actor(), such as
calling send() to add an Int to the channel:

```
val sample = ActorSample()
val sendChannel = sample.actOut(GlobalScope)

GlobalScope.launch(Dispatchers.Default) {
  for (i in 0..100)  {
    sendChannel.send(i)
  }
}

sendChannel.close()

val results = sample.receivedItems
```

In the end, we wind up with a list of 101 integers.

The documentation for actor() hints at a plan to replace this function with
"complex actors", though the associated issue indicates that this work is on hold for a
while.

Using a `SendChannel`

The `SendChannel` interface is where we find `offer()` and `send()`:

- `offer()` adds the object to the channel if there is room in its buffer and returns `true`, otherwise it returns `false`
- `send()` adds the object to the channel if there is room in its buffer and suspends otherwise until there is room

Since `send()` suspends, it has the `suspend` keyword and must be called inside of a coroutine. `offer()` does not suspend and can be called from normal functions, but you need to deal with the case where it returns `false`. Here, "deal with" could simply mean "ignore the return value and do not worry about the case where the offer was not accepted".

In Kotlin/JVM, there is a third option for publishing objects via a `SendChannel`: `sendBlocking()`. As the name suggests, where `send()` would suspend, `sendBlocking()` blocks. This gives you `send()`-style "wait until we can send it" semantics without the need for a coroutine. However, you need to be in position to block the calling thread, which may or may not be practical. Mostly, this appears to be here for Java interoperability, so Java code has a convenient, if inelegant, way to publish objects to a `SendChannel`.

You can proactively close a `SendChannel` by calling `close()`. A holder of a `SendChannel` can see if it is closed via `isClosedForSend()`, while the holder of a `ReceiveChannel` can see if it is closed via `isClosedForReceive()`. Iterating consumers, such as `consumeEach()`, automatically check for closure and will handle that accordingly — in the case of `consumeEach()`, it returns.

Consuming a `ReceiveChannel`

So far, our examples have focused mostly on using `consumeEach()` to get the objects from a `ReceiveChannel`. That is a `suspend` function and so needs to be run inside a coroutine, and it loops until the channel is closed.

However, that is not our only way to get objects out of a `ReceiveChannel`.

Direct Consumption

There are other "terminal operators" besides `consumeEach()` for getting data off of a

ReceiveChannel.

poll()

poll() is the opposite of offer():

- offer() is a regular (non-suspend) function that puts an object into the channel if there is room
- poll() is a regular (non-suspend) function that takes an object out of the channel if there is one, returning null otherwise

As a result, poll() has a very good chance of returning null:

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

fun main() {
  val channel = Channel<Int>(Channel.BUFFERED)

  GlobalScope.launch(Dispatchers.Default) {
    for (i in 0 until 10) {
      delay(200)
      channel.send(Random.nextInt(1,100))
    }

    channel.close()
  }

  println(channel.poll())
}
```

This is all but guaranteed to print null, as our coroutine has a 200-millisecond delay before the send() call, so the channel is likely to be empty when our poll() call is made.

Note that if the channel wound up failing due to some exception, poll() will throw that exception. It would be your job to wrap the poll() call in a try/catch construct to handle such exceptions.

receive() and receiveOrNull()

Like poll(), receive() and receiveOrNull() pull an object off of the channel and

return it, if an object is available.

Unlike `poll()`, `receive()` and `receiveOrNull()` will suspend if the channel is empty at the time of the call. As suspend functions, you will have to call them inside of a coroutine. But, it means that they appear to "block" waiting for an object.

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

fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    println(randomPercentages(10, 200).receive())
    println("That's all folks!")
  }

  println("...and we're off!")
}

fun CoroutineScope.randomPercentages(count: Int, delayMs: Long) = produce {
  for (i in 0 until count) {
    delay(delayMs)
    send(Random.nextInt(1,100))
  }
}
```

(from ["Channel and receive()" in the Klassbook](#))

Here, we will get just one random number, despite setting up a channel to generate ten of them, since we are only calling `receive()` once.

The difference between `receive()` and `receiveOrNull()` comes with a closed channel. `receiveOrNull()` returns `null` in that case. `receive()` throws a dedicated `ClosedReceiveChannelException` instead.

toList()

If you cannot do anything with the `ReceiveChannel` output until the channel is closed, you could elect to use `toList()`. This collects all the objects emitted into the channel into a `List`, returning it when the channel gets closed:

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

```
fun main() {
  GlobalScope.launch(Dispatchers.Main) {
    println(randomPercentages(10, 200).toList())
    println("That's all folks!")
  }

  println("...and we're off!")
}

fun CoroutineScope.randomPercentages(count: Int, delayMs: Long) = produce {
  for (i in 0 until count) {
    delay(delayMs)
    send(Random.nextInt(1,100))
  }
}
```

(from ["toList() for Channel" in the Klassbook](#))

Here, we get all of our random numbers after the full 2000-millisecond amount of time has elapsed:

```
...and we're off!
[29, 14, 26, 30, 42, 17, 67, 76, 64, 16]
That's all folks!
```

toList() is a suspend function, like consumeEach(), and so needs to be called from inside of a coroutine.

Converting to Flow

There are two extension functions for ReceiveChannel that wrap it in a Flow: receiveAsFlow() and consumeAsFlow(). Both give you a Flow and you can use normal Flow operators to process the items sent on the Channel.

The difference is that the Flow returned by consumeAsFlow() can only be collected once (after which the channel is closed), whereas the Flow returned by receiveAsFlow() can be collected multiple times.

Capacity Planning

Functions like receive() and poll() behave differently depending on whether or not there is an object available in the channel at the time of the call. This implies the possibility of some amount of buffering in a Channel.

Exactly how that buffering works depends on what value you pass to the `Channel` constructor or as a parameter to functions like `produce()`.

Zero

The default capacity is zero. This means that the `Channel` implementation is an internal class called `RendezvousChannel`, and such a channel has no buffer. This imposes some synchronization on object passing: a `send()` call will suspend until something, such as a `receive()` call, is in position to consume the object being sent.

This is useful in cases where your receiver will usually be suspended on a `receive()` call at the time of the `send()` call. For example, if you are using `consumeEach()`, other than the time you spend processing an individual object that you consume, the rest of the time you are suspended in `receive()`, so `send()` is unlikely to suspend.

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() {
  val channel = Channel<Int>(0)

  GlobalScope.launch(Dispatchers.Main) {
    println("before receiving")
    println(channel.receive())
    println("after receiving")
  }

  GlobalScope.launch(Dispatchers.Default) {
    println("before sending")
    delay(2000)
    channel.send(123)
    println("after sending")
  }

  println("...and we're off!")
}
```

(from ["A Zero-Buffer Channel" in the Klassbook](#))

Here, we have a zero-buffer `Channel`. So, when we `delay()` our `send()` for two seconds, our `receive()` suspends for those two seconds.

An Ordinary Positive Number

If you pass in some arbitrary Int value — say, 4 — you get a channel with that size of buffer. You will be able to successfully send() without suspending that many times (filling the buffer) before subsequent send() calls will suspend until the consumer is able to start pulling objects off of the channel.

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() {
  val channel = Channel<Int>(4)

  GlobalScope.launch(Dispatchers.Main) {
    delay(2000)
    println("before receiving")
    channel.consumeEach { println(it) }
    println("after receiving")
    channel.close()
  }

  GlobalScope.launch(Dispatchers.Default) {
    println("before first send")
    channel.send(123)
    println("before second send")
    channel.send(456)
    println("before third send")
    channel.send(789)
    println("before fourth send")
    channel.send(1234)
    println("before fifth send")
    channel.send(5678)
    println("after sending all")
  }

  println("...and we're off!")
}
```

(from ["A Fixed-Buffer Channel" in the Klassbook](#))

Here, we have a Channel with a four-object buffer. We use delay() to slow down our consumption of objects from the channel. Since we have a four-object buffer, our fifth send() suspends until the delay() completes and we can start pulling objects off the channel.

In many cases, having a small buffer is the safest approach, to reduce the frequency

with which sending code suspends while at the same time controlling memory consumption.

Unlimited

If you pass in the magic value of `Channel.UNLIMITED`, you get a channel that has an "unlimited" buffer. Like a `MutableList`, each new object added via `send()` or `offer()` just gets added to the channel, and the channel's buffer grows. The only practical limit is the amount of available heap space.

On the plus side, you are guaranteed that `send()` will never suspend and that `offer()` will always succeed.

On the minus side, your app crashes if you either:

- Stopped consuming the channel for some reason, or
- Your sending code is much faster than is your receiving code

In either case, your channel's buffer will just keep growing and growing, until eventually some memory allocation fails and you get an `OutOfMemoryError`.

As a result, this is a relatively risky choice.

Conflated

If you pass in the magic value of `Channel.CONFLATED`, you get a `ConflatedChannel` implementation. Similar to if you passed `1` as the capacity, you get a one-element buffer with a `ConflatedChannel`. The difference is what happens when there is an element in that buffer and something tries to `send()` or `offer()` a new object:

- With a regular `Channel` created with a capacity of `1`, `send()` will suspend and `offer()` will return `false`, as the buffer is full
- With a `ConflatedChannel`, `send()` and `offer()` will *overwrite* the buffer with the new value

As a result, if objects are placed into the channel faster than they are consumed, some objects will be lost.

```
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*
```

```kotlin
fun main() {
  val channel = Channel<Int>(Channel.CONFLATED)

  GlobalScope.launch(Dispatchers.Main) {
    delay(2000)
    println("before receiving")
    channel.consumeEach { println(it) }
    println("after receiving")
    channel.close()
  }

  GlobalScope.launch(Dispatchers.Default) {
    println("before first send")
    channel.send(123)
    println("before second send")
    channel.send(456)
    println("before third send")
    channel.send(789)
    println("before fourth send")
    channel.send(1234)
    println("before fifth send")
    channel.send(5678)
    println("after sending all")
  }

  println("...and we're off!")
}
```

(from ["A Conflated Channel" in the Klassbook](#))

This is the same example as for the fixed-size buffer, except that we are using
Channel.CONFLATED. Now, all five send() calls complete without interruption, but
our consumeEach() only winds up seeing the last of them, because it was delayed by
two seconds.

Broadcasting Data

By default, objects placed into a channel get delivered to one consumer. You can
have multiple consumers, but each object is delivered to just one of those
consumers. This allows for fan-out behavior for parallelizing work:

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() {
```

```kotlin
  val channel = Channel<Int>(4)

  GlobalScope.launch(Dispatchers.Main) {
    println("A: before receiving")
    channel.consumeEach { println("A: $it") }
    println("A: after receiving")
  }

  GlobalScope.launch(Dispatchers.Main) {
    println("B: before receiving")
    channel.consumeEach { println("B: $it") }
    println("B: after receiving")
  }

  GlobalScope.launch(Dispatchers.Default) {
    println("before first send")
    channel.send(123)
    println("before second send")
    channel.send(456)
    println("before third send")
    channel.send(789)
    println("before fourth send")
    channel.send(1234)
    println("before fifth send")
    channel.send(5678)
    println("after sending all")
    delay(100)
    channel.close()
  }

  println("...and we're off!")
}
```

(from ["Channels and Fan-out" in the Klassbook](#))

Here, we have two consuming coroutines, each pulling from the same channel. Each item put into the channel will be delivered to one of those coroutines, but not both. So, you might get:

```
...and we're off!
A: before receiving
B: before receiving
before first send
before second send
before third send
before fourth send
before fifth send
```

```
after sending all
A: 123
A: 789
A: 1234
A: 5678
B: 456
```

Sometimes, though, we may want to do a "broadcast", where objects get delivered to all active consumers, not just to one consumer out of the collection. For that, we have BroadcastChannel.

A Basic Broadcast

```kotlin
import kotlinx.coroutines.*
import kotlinx.coroutines.channels.*

fun main() {
  val channel = BroadcastChannel<Int>(4)

  GlobalScope.launch(Dispatchers.Main) {
    println("A: before receiving")
    channel.consumeEach { println("A: $it") }
    println("A: after receiving")
  }

  GlobalScope.launch(Dispatchers.Main) {
    println("B: before receiving")
    channel.consumeEach { println("B: $it") }
    println("B: after receiving")
  }

  GlobalScope.launch(Dispatchers.Default) {
    println("before first send")
    channel.send(123)
    println("before second send")
    channel.send(456)
    println("before third send")
    channel.send(789)
    println("before fourth send")
    channel.send(1234)
    println("before fifth send")
    channel.send(5678)
    println("after sending all")
    delay(100)
    channel.close()
  }
```

```
  println("...and we're off!")
}
```

This is identical to the previous example, except that we use the
BroadcastChannel() factory function, instead of Channel(). This gives us a
BroadcastChannel instance, and that delivers each sent item to each active
consumer at the time the item was sent.

So, in this case, both the A and B consuming coroutines get all five items:

```
...and we're off!
A: before receiving
B: before receiving
before first send
before second send
before third send
before fourth send
before fifth send
after sending all
A: 123
A: 456
A: 789
A: 1234
A: 5678
B: 123
B: 456
B: 789
B: 1234
B: 5678
A: after receiving
B: after receiving
```

Setting Up a Broadcast

The BroadcastChannel() factory function takes an Int that serves in the same role
as it does with Channel(), indicating the capacity of the channel.

Alternatively, you can get a BroadcastChannel by:

- Calling broadcast() on a ReceiveChannel, which gives you a
  BroadcastChannel that broadcasts each item delivered to that
  ReceiveChannel

- Calling `broadcastIn()` on a `Flow`, which gives you a `BroadcastChannel` that broadcasts each item delivered on that `Flow`

Consuming Broadcasts

In addition to `consumeEach()` to consume the "broadcast" items, you can get a `ReceiveChannel` from a `BroadcastChannel` by calling `openSubscription()`. This simply subscribes to the `BroadcastChannel` and returns the `ReceiveChannel` for you to use like any other `Channel` to consume the items.

Alternatively, `asFlow()` subscribes to the `BroadcastChannel` and returns a `Flow` that you can use like any other `Flow` to consume the items.

ConflatedBroadcastChannel

If you pass `Channel.CONFLATED` to the `BroadcastChannel()` factory function, you get a `ConflatedBroadcastChannel`. As with a regular conflated channel, anyone who starts consuming items gets the last-sent value immediately, in addition to future values. And, the channel's one-item buffer gets overwritten with new values, rather than suspending any `send()` call.

`ConflatedBroadcastChannel`, though, will be deprecated in an upcoming coroutines version, [replaced by `StateFlow`](#).

Operators

`Channel`, and its various subtypes like `BroadcastChannel`, has a lot of extension functions that serve as operators, much like the ones we have for `Flow`. However, most, if not all, are marked as deprecated. In general, the expectation now is that you will do any "heavy lifting" using a `Flow`, perhaps one adapted from a `Channel`.

Bridging to Callback APIs

Perhaps, one day in the future, all of our APIs will be ready for Kotlin coroutines. We will be given a rich set of `suspend` functions and `Flow` objects that we can launch on our desired dispatchers and organize the results as we see fit.

That day is not today. Nor is it likely to be tomorrow, or the day after that.

It is far more likely that we will be dealing with APIs that have some other sort of asynchronous model:

- Perhaps they use a callback or listener object per request, where you pass the callback to some API call and that callback gets results for that call's bit of work
- Perhaps they use more of a polling approach, where you make an API call, then periodically check for a status update later.
- Perhaps they use some other semi-reactive solution, like `Future` objects from `java.util.concurrent`

For those, if you want to use them in a coroutines-centric app, you will want to build a bridge between these other sorts of patterns and `suspend` functions, `Flow`, or similar coroutines options. In this chapter, we will explore a bit about how to do this.

Coroutine Builders for Callbacks

Sometimes, we have an API that we need to call that does a single thing, but does so asynchronously. We have to provide a callback implementation to that API, and that callback will be called upon completion of the work, for both success and failure cases.

In other words, we are in a level of callback hell:

```
doSomething(new Something.Callback() {
  public void whenDone() {
    doTheNextThing(new NextThing.Callback() {
      public void onSuccess(List<String> stuff) {
        doSomethingElse(stuff, new SomethingElse.Callback() {
          public void janeStopThisCrazyThing(String value) {
            // TODO
          }
        });
      }

      public void onError(Throwable t) {
        // TODO
      }
    });
  }
});
```

Here, doSomething(), doTheNextThing(), and doSomethingElse() each need a
callback. And, as we saw in the introductory chapter, we would love to be able to
rewrite this in Kotlin and coroutines as:

```
someCoroutineScope.launch {
  doSomethingCoroutine()

  try {
    val result = doSomethingElseCoroutine(doTheNextThingCoroutine())

    // TODO
  } catch (t: Throwable) {
    // TODO
  }
}
```

This means that we need to be able to adapt a callback-based API to one that uses a
suspend function to return results directly or throw exceptions as needed. For that,
we have suspendCoroutine() and suspendCancellableCoroutine(), which we can
use to build the ...Coroutine() wrappers around our callback-based APIs.

suspendCoroutine()

suspendCoroutine() is for cases where the API is "fire and forget". We ask it to do
something, and we get the result back asynchronously, but there is no means to

manipulate the in-flight piece of work. In particular, it is for cases where we have no option to cancel that ongoing work if it no longer needed (e.g., the CoroutinesScope has been canceled).

suspendCoroutine() takes a lambda expression or other function type. You get a Continuation object that you use from your callback to indicate when the work is done and what the result was.

In our fictitious code sample, doSomething() performs work but does not return anything (i.e., it returns Unit from a Kotlin standpoint). For that, doSomethingCoroutine() might look like:

```kotlin
suspend fun doSomethingCoroutine() = suspendCoroutine<Unit> { continuation ->
  doSomething(object : Something.Callback() {
    override fun whenDone() {
      continuation.resume(Unit)
    }
  })
}
```

You call resume() on the Continuation to indicate that the work is complete. Whatever object you supply to resume() is what the overall coroutine returns. In this case, we use Unit.

doTheNextThing() is more complex, in that its callback either gets a list of string values or a Throwable, depending on success or failure. As a result, doTheNextThingCoroutine() is also a bit more complex:

```kotlin
suspend fun doTheNextThingCoroutine(): List<String> =
  suspendCoroutine { continuation ->
    doTheNextThing(object : NextThing.Callback() {
      override fun onSuccess(result: List<String>) {
        continuation.resume(result)
      }

      override fun onError(t: Throwable) {
        continuation.resumeWithException(t)
      }
    })
  }
```

Here, we use both resume() and resumeWithException() on Continuation. Our successful result is handled via resume(), while our failure is handled via resumeWithException(). The resulting coroutine either returns the result or throws

the Throwable, so we can consume the results through normal imperative syntax inside of our CoroutineScope.

suspendCancellableCoroutine()

suspendCancellableCoroutine() is for cases where the API we are using has an option to be canceled. Syntactically, suspendCancellableCoroutine() is nearly identical to suspendCoroutine(). The big difference is that the lambda expression (or other function type) gets passed a CancellableContinuation, and that has an invokeOnCancellation() function. invokeOnCancellation() takes a lambda expression (or other function type) that will get called if the coroutine itself is being canceled, due to the CoroutineScope being canceled. This gives you an opportunity to cancel the ongoing work through the callback-style API's facility for doing so.

For example, Andrey Mischenko maintains [kotlin-coroutines-okhttp](#), a library that provides coroutine support for older versions of OkHttp. OkHttp is an HTTP client API. It was originally written in Java and lacked coroutine support. OkHttp is now written in Kotlin, but that came with some new restrictions (e.g., for Android, it requires Android 5.0 or higher), and, as of 4.9.0, it still lacks coroutine support. kotlin-coroutines-okhttp adapts OkHttp for coroutines.

In OkHttp, you can make an HTTP request via a Call object. A Call contains information about what request to make (e.g., the URL to use, the HTTP action to perform). For asynchronous work, you can enqueue() the Call, supplying a Callback to find out the results. While that Call is enqueued, you can call cancel() on it to cancel its ongoing work.

So, the library uses suspendCancellableCoroutine() as part of an await() extension function on Call. [The library does some extra stuff](#), but a simplified version of await() might look like:

```
public suspend fun Call.await(): Response {
    return suspendCancellableCoroutine { continuation ->
        enqueue(object : Callback {
            override fun onResponse(call: Call, response: Response) {
                continuation.resume(response)
            }

            override fun onFailure(call: Call, e: IOException) {
                continuation.resumeWithException(e)
            }
        })
```

```
    continuation.invokeOnCancellation {
        try {
            cancel()
        } catch (ex: Throwable) {
            //Ignore cancel exception
        }
    }
  }
}
```

Here, in addition to using `resume()` and `resumeWithException()` for handling results from the asynchronous work, we use `invokeOnCancellation()` to find out when the coroutine gets canceled. We then call `cancel()` on the `Call` to cancel the network I/O.

Flow Builders for Callbacks

Both `suspendCoroutine()` and `suspendCancellableCoroutine()` are fine for one-shot asynchronous work. If your callback-based API will return many items, then most likely you would want to adapt that into a `Flow`, not a simple `suspend` function.

For that, we have `callbackFlow()`.

Like `suspendCoroutine()` and `suspendCancellableCoroutine()`, `callbackFlow()` takes a lambda expression (or other function type). Inside the lambda, `this` is a `ProducerScope`, which is a combination of a `CoroutineScope` (so you can call suspend functions) and a `SendChannel` (so you can `offer()` objects to be emitted by the `Flow`).

You then:

- Invoke your callback-based API
- Inside that callback call `SendChannel` functions like `offer()` (when you have a value to emit) or `close()` (if the API is telling you that there will be no more data to receive)
- Call `awaitClose()`, which takes a lambda expression (or other function type) that will get called when the `Flow` itself is closed, so you can clean things up (e.g., unregister from the callback)

`callbackFlow()` then returns the configured `Flow`, and your lambda expression will be called as soon as a terminal operator is called on the `Flow` to collect the results.

For example, imagine we have a registerForStuff() function that takes a callback that will be called when stuff is ready for use. There is a corresponding unregisterForStuff() that unregisters our callback so we stop receiving the stuff. Exactly where the "stuff" comes from and how it is obtained (e.g., polling) is part of the implementation details of the registerForStuff() API that we are using.

We could convert those functions into a Flow using callbackFlow():

```
fun stuffFlow() = callbackFlow<Stuff> {
  val watcher = object : StuffWatcher {
    override fun onStuff(s: Stuff) {
      offer(s)
    }
  }

  registerForStuff(watcher)

  awaitClose { unregisterForStuff(watcher) }
}
```

Whenever our watcher is called with onStuff(), we offer() the object that we receive to the ProducerScope, so it is emitted by the Flow created by callbackFlow(). When the Flow is closed, we unregister so we stop trying to use this API.

If StuffWatcher has a function where we find out that the API itself is out of stuff, we can use that to close() the ProducerScope and signal that the Flow is closed:

```
fun stuffFlow() = callbackFlow<Stuff> {
  val watcher = object : StuffWatcher {
    override fun onStuff(s: Stuff) {
      offer(s)
    }

    override fun onComplete() {
      close()
    }
  }

  registerForStuff(watcher)

  awaitClose { unregisterForStuff(watcher) }
}
```

Seeing These in Action

The problem with these coroutines options is that to have code that runs, you need suitable APIs to call. That is why the code snippets in this chapter are just plain Kotlin code and are not runnable samples in the Klassbook.

In an upcoming chapter on Android UIs, we will see examples of these constructs used as part of combining coroutines with the classic Android view system and related UI concerns.

Selecting Results

This chapter will be added in an upcoming edition of the book!

Creating Custom Scopes

`CoroutineScope` is where we get `launch()` and `async()`, and as such it is a fundamental piece of our use of coroutines.

Usually, we get a `CoroutineScope` from somebody else. That could be:

- `GlobalScope`
- A scope from some framework that we are using, such as the Jetpack `ViewModel` for Android app development
- Derived from other scopes, such as via `withContext()`

However, from time to time, we will want our own `CoroutineScope` that we control outright. In this chapter, we will explore why we might need that and how to implement it.

Getting to the Root of the Problem

In the end, a `CoroutineScope` is supposed to model some sort of lifecycle. And, sometimes, we have a lifecycle for which there is no externally-supplied `CoroutineScope`.

Avoiding `GlobalScope`

Sometimes, the "lifecycle" is simply the lifetime of the process. You have some scope that gets created (perhaps lazily) and that scope can be used for the entire duration of the process.

`GlobalScope` is the coroutines library's built-in solution for this. For lightweight purposes — such as book examples — it is fine.

However, it is a global singleton. Its definition is controlled by the coroutines library, not by you. In regular code, that might be fine. However, in test code, you may want to be able to use some other scope, such as `TestCoroutineScope`, that allows you to better control the behavior of coroutines for better repeatable results. Referring to `GlobalScope` directly locks you into its implementation.

You could work around that using dependency inversion techniques. You could have code needing a process-wide `CoroutineScope` receive it via injection. Then, you could arrange to inject `GlobalScope` in production code and a `TestCoroutineScope` in your unit tests.

However, you might still want a custom scope instead of injecting `GlobalScope`:

- You might read [the documentation for `GlobalScope`](), see the warning about using `GlobalScope`:

Application code usually should use an application-defined `CoroutineScope`. Using `async` or `launch` on the instance of `GlobalScope` is highly discouraged.

- You might want to configure this process-wide scope, such as choosing what the `Job` is that it uses, and you have no ability to configure `GlobalScope`
- Code from libraries might be using `GlobalScope`, and while that should not introduce any harmful side effects... why risk it?

Creating Frameworks

As mentioned above, we often get a `CoroutineScope` from some coroutines-aware framework that we use.

One counter-example is when *you* are the one creating the framework. In that case, you may want to be creating custom scopes tied to your framework's own lifecycles.

Every development team that created custom scopes for us to consume had to go through that work:

- The Android Jetpack team created custom scopes tied to lifecycles of activities and fragments
- The Android Jetpack team also created custom scopes tied to lifecycles of viewmodels, which themselves could be tied to all sorts of lifecycles

(activities, fragments, navigation graphs, composables in Jetpack Compose, etc.)
- A Web server team might offer a custom scope tied to a specific Web request, so you can use `suspend` functions for fulfilling that request
- And so on

Setting Up a Custom Scope

Fortunately, setting up a custom `CoroutineScope` is fairly simple, though there are a few options to consider.

Creating the Scope

There are top-level factory functions for creating a `CoroutineScope`. The one you will most likely use is `CoroutineScope()`.

CoroutineScope()

`CoroutineScope()` creates a scope, given a `CoroutineContext` as a parameter. Dispatchers and jobs are `CoroutineContext` elements.

So, you can create a `CoroutineScope` with a `SupervisorJob`:

```kotlin
val scope = CoroutineScope(SupervisorJob())
```

You can create a `CoroutineScope` tied to a particular dispatcher:

```kotlin
val scope = CoroutineScope(Dispatchers.Default)
```

Since `CoroutineContext` instances can be combined using +, you can create a scope with a `SupervisorJob` and a particular dispatcher:

```kotlin
val scope = CoroutineScope(SupervisorJob() + Dispatchers.Default)
```

If your `CoroutineContext` does not specify a type of `Job`, a regular `Job` will be used by default.

Specifying a dispatcher is optional. It indicates the default dispatcher used for `launch()` and `async()` calls, if they do not specify a dispatcher themselves.

```
MainScope()
```

There is also the `MainScope()` factory function. This is equivalent to calling `CoroutineScope(SupervisorJob() + Dispatchers.Main)`. Some type of UI framework might elect to use `MainScope()` to create a custom scope. Otherwise, some other dispatcher is probably a better choice.

Adding a Name

For logging and debugging purposes, you might want to associate a name with your custom scope, to indicate what that scope is tied to. You can add `CoroutineName()` to the `CoroutineContext` via `+`, where you provide some string to `CoroutineName()` that serves as the name:

```
val scope = CoroutineScope(SupervisorJob() + Dispatchers.Default +
CoroutineName("request $request"))
```

Frequently, that name will be dynamic, with some identifier tying it back to the lifecycle for which you are creating a scope.

Note that this works for derived scopes as well:

```
withContext(Dispatchers.IO + CoroutineName("user authentication")) {
  TODO("something good, involving user authentication and I/O")
}
```

Using the Scope

You then use your custom scope much the same as how you would use any other `CoroutineScope` that you might get from some framework:

- Call `launch()` or `async()` on it to start a coroutine
- Use it with `withContext()` by obtaining the scope's `CoroutineContext`:

```
withContext(myCustomScope.coroutineContext) {
  // TODO something here that needs the custom scope
}
```

We will see an example of this in the context of Android later in the book.

Cleaning Up the Scope

For an application-level scope, one that lives as long as your process does, you will not have an opportunity to clean it up.

For anything narrower, you need to `cancel()` that scope when the associated lifecycle is complete. So, for example:

- A Web framework offering a custom scope per Web request would `cancel()` that scope when the request is completed and the response has been sent to the client
- A GUI framework offering a custom scope per screen would `cancel()` that scope when the screen is no longer being used (e.g., its window was closed)
- A `cron`-style program for executing periodic tasks might offer a custom per-task scope; that scope should be canceled when the task completes or times out

Anti-Pattern: Extending `CoroutineScope`

`CoroutineScope` is an `interface`. Technically, there is nothing stopping you from declaring it on whatever it is that you want. You simply need to have a `coroutineContext` property supplying your `CoroutineContext` — there are no functions that you need to override.

For example, in Android, one early pattern for coroutines was to implement `CoroutineScope` on your `ViewModel`:

```
class MyViewModel : ViewModel(), CoroutineScope {
  override val coroutineContext = Job() + Dispatchers.Default

  override fun onCleared() {
    cancel() // to cancel the coroutines that are still outstanding
  }

  fun doSomething() {
    launch {

    } // we can do this, because this is a CoroutineScope!
  }
}
```

This works. However, while `CoroutineScope` *requires* very little, it *exposes* a lot, by

means of extension functions. `launch()` and `async()` are extension functions on `CoroutineScope`, for example. [The API surface of a CoroutineScope](#) is bigger than you might think. Anything that holds a `ViewModel` — such as an activity or fragment — could access that API. That is not really the role of a `ViewModel`, though.

JetBrains recommends composition over inheritance here:

> Manual implementation of this interface is not recommended, implementation by delegation should be preferred instead.

There may be edge cases where implementing `CoroutineScope` on a custom class is the best solution. But, on the whole, your custom scopes should be properties of things that have lifecycles, not *be* things that have lifecycles.

Testing Coroutines

This chapter will be added in a future version of this book!

Integrating with RxJava

This chapter will be added in an upcoming version of this book!

Coroutines and Android

Applying Coroutines to Your UI

Architecture has become a major topic of discussion in Android app development. Usually, the focus is on UI architecture: how our activities and fragments render data and collect input from the user, while also being able to perform asynchronous disk and network I/O. Android's "configuration changes" — where activities and fragments get destroyed and recreated when the user rotates the screen, etc. — make a UI architecture particularly interesting to develop.

In this chapter, we will explore how a modern Jetpack app, leveraging `ViewModel`, might employ coroutines.

Most of the code shown in this chapter comes from an appendix, consisting of a hands-on tutorial for converting an RxJava-based Android app into one that uses coroutines.

A Quick Architecture Review

There are lots of different architecture patterns in use in Android apps today.

Google, however, is steering developers towards a particular set of constructs for conventional model-centric UI development.
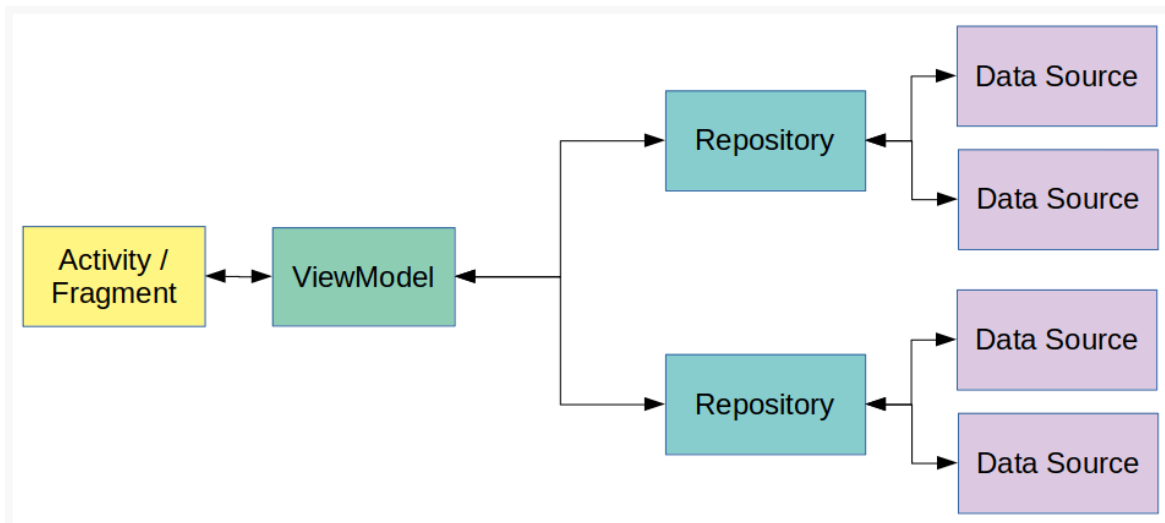

Figure 10: Android Common UI Architecture

Repositories and Data Sources

Repositories are singletons that are responsible for hiding all of the details of where model objects come from and go to. A repository provides a clean asynchronous API for retrieving models and making changes to them (insert, update, delete).

For some apps, the repository might do all of that work directly. For more complex apps, you might pull some of those details out into data sources. As the name suggests, a data source wraps some source of data, typically either a server (a "remote" data source) or a database (a "local" data source). Those data sources handle those details (e.g., communicating with a Web service via Retrofit). The repository, in this approach, orchestrates the work of those data sources and handles other cross-cutting concerns, such as in-memory caching and data conversion from data-source-specific types to normalized object models.

Because servers and database schemas may need to change, the repository does not expose objects that are tied to such things. Objects that model Web service responses or Room query results are considered to be data transfer objects (DTOs) are part of the repository/data source implementation. The model objects that the repository exposes via its API should be free from any data-store-specific encumbrances (e.g., Room annotations) and should provide a reasonable

representation of the data (e.g., replacing magic constant strings/numbers with type-safe enums).

`ViewModel` **and the UI Layer**

With the repository providing a clean API of clean models, the UI layer can focus mostly on rendering the UI and processing user input. Android, with its configuration changes, makes that a bit complicated, as our activities and fragments can get destroyed and recreated on the fly.

As a result, rather than having activities or fragments work directly with repositories, we use `ViewModel` as an intermediary. An Android `ViewModel` is set up to survive configuration changes yet be cleared when the associated activity or fragment instance is being destroyed permanently. So, the `ViewModel` is responsible for communications with the repository, as the `ViewModel` is (relatively) stable. The activities and fragments work with the `ViewModel` to indirectly interact with the repositories.

Use Cases

You will find some projects, and some discussions around architecture, introduce "use cases". These form a layer between the viewmodels and the repositories. As the name suggests, use cases model specific operations, particularly ones that might affect two or more repositories. Rather than giving viewmodels unfettered access to a bunch of repositories, the viewmodels might only get a few use cases. This improves data hiding at the cost of another layer of indirection.

This book will skip use cases, but a lot of what is discussed regarding repositories themselves should be relevant for use cases that, in effect, expose a focused subset of repository capabilities.

The Rest of the Stuff

Not everything neatly fits this approach. This architecture is fine for model-centric activities and fragments, but it will not necessarily hold up for:

- Responding to push messages
- Scheduling periodic background syncs with a server
- Implementing an app widget
- And so on

We will explore how coroutines can work with other sorts of Android components and Jetpack libraries later in the book. For now, we will keep our focus on basic UI integration.

The General Philosophy

Given the aforementioned generalized architecture, one possible role of couroutines is fairly straightforward:

- The repository exposes a coroutines-centric API to the `ViewModel`
- The `ViewModel` exposes a `LiveData-` or coroutines-centric API to the activity/fragment, where `LiveData` (if you use it) adapts the results of the coroutines

Coroutines and Repositories

A well-designed repository has a mostly-asynchronous API. Anything that might take time — disk I/O, network I/O, data conversion, etc. — should be able to be performed on a background thread. Whether that background thread is supplied by the repository or is supplied by a consumer of the repository depends a bit on the means by which you implement the asynchronous API.

In a coroutines-based app, this means that a repository should expose a lot of `suspend` functions, frequently using `withContext()` to control the dispatcher that is used for the background work. Callers of those functions then can control:

- The dispatcher that is used to receive the results
- How much concurrency is involved (e.g., do we use `async()`/`await()` or just `launch()`?)

For things where the repository will provide a stream of results, the repository can return a `Flow` instead of a simple `suspend` function. The consumer of that API can then `collect()` the results from the `Flow` for as long as the consumer is interested in those results (e.g., until we exit the activity).

ViewModel, LiveData, and Coroutines

The point behind the Android `ViewModel` is to provide a stable spot for data despite the fact that the activities and fragments in the UI layer might be destroyed and recreated due to configuration changes. The idea in Google's recommended architecture is that the `ViewModel` be responsible for working with the repositories,

with the UI layer talking only to the `ViewModel`.

However, if you have a reactive API published by the repositories, you will need a corresponding reactive API in the `ViewModel` to push those results to the UI layer.

Google's original solution for that was `LiveData`. `LiveData` is a value holder with a lifecycle-aware observer framework. An activity or fragment can observe the `LiveData`, get whatever value is available when the activity or fragment is created (or re-created), and get updates over time while it observes the `LiveData`. The lifecycle awareness of `LiveData` makes it very easy to integrate, and there are a few options for piping the results of coroutines into a `LiveData`.

Another approach, though, is to skip `LiveData` and have the `ViewModel` expose a coroutines-based API, using the same mechanisms that you used in your repositories. While coroutines itself knows nothing about Android's lifecycle, the Jetpack libraries provide some Android-aware coroutines hooks, such as a `CoroutineScope` tied to the activity and fragment lifecycle. This way, you can consume the coroutines responses and still remain safe from a lifecycle standpoint. However, most aspects of coroutines do not cache results, and so you need to beware of losing past results on a configuration change.

We will look at both approaches — `LiveData` adapted from coroutines, or simply using coroutines directly — in this chapter.

Exposing Coroutines from a Repository

In Google's architecture, your I/O — whether disk or network — is handled by data sources, orchestrated by a repository. Ideally, these offer a reactive API, and if you are reading this book, that reactive API probably involves coroutines.

Data Sources

Simple architectures might have repositories perform I/O directly. However, if you need to have significant business logic — such as converting between database/server data types and model objects — you may want to have the I/O be isolated in data source objects. Those tend to be fairly tightly tied to the OS (for disk I/O) or a server (for network I/O). Isolating that code in data source objects allows you to swap in mocks for testing the repositories and keeps the I/O-specific code in tightly-scoped objects.

Remote

The simplest solution for a remote data source is to use a library that can talk to your Web service with a coroutines-centric API. For Android, the preeminent example of that is [Retrofit](). [The `Retrofit` module]() of [the book's primary sample Android project]() has a remote data source in the form of `NWSInterface`, a Retrofit-declared Kotlin `interface` for getting values from the US National Weather Service:

```
package com.commonsware.jetpack.weather

import retrofit2.http.GET
import retrofit2.http.Headers
import retrofit2.http.Path

interface NWSInterface {
  @Headers("Accept: application/geo+json")
  @GET("/gridpoints/{office}/{gridX},{gridY}/forecast")
  suspend fun getForecast(
    @Path("office") office: String,
    @Path("gridX") gridX: Int,
    @Path("gridY") gridY: Int
  ): WeatherResponse
}
```

(from [Retrofit/src/main/java/com/commonsware/jetpack/weather/NWSInterface.kt]())

With Retrofit, you can just use the `suspend` keyword on a Retrofit API function, as we have here for `getForecast()`. Retrofit will know to code-generate an implementation of `NWSInterface` that uses `withContext()` to have the actual network I/O be conducted on a Retrofit-supplied dispatcher.

However, if you do not have a library that offers a coroutines-based API for accessing your server, you will need to consider:

- Adapting a library that offers some other reactive API, such as one based on [RxJava]()
- Setting up a [bridge]() to an API that uses callbacks
- Wrapping a synchronous API in a `withContext()` call, to arrange to have that work be performed on a suitable dispatcher (e.g., `Dispatchers.IO`)

Local

Similarly, for a local data source, ideally you use some library that knows how to

work with local data with a coroutines-centric API. Here, Room from Google's
Jetpack libraries fills a similar role for a local SQLite database as Retrofit does for
hitting a REST-style Web service. The Room module of the book's primary sample
Android project has a local data source in the form of the RoomEntity.Store
interface:

```kotlin
package com.commonsware.todo.repo

import androidx.room.*
import kotlinx.coroutines.flow.Flow
import org.threeten.bp.Instant
import java.util.*

@Entity(tableName = "todos", indices = [Index(value = ["id"])])
data class ToDoEntity(
  val description: String,
  @PrimaryKey
  val id: String = UUID.randomUUID().toString(),
  val notes: String = "",
  val createdOn: Instant = Instant.now(),
  val isCompleted: Boolean = false
) {
  constructor(model: ToDoModel) : this(
    id = model.id,
    description = model.description,
    isCompleted = model.isCompleted,
    notes = model.notes,
    createdOn = model.createdOn
  )

  fun toModel(): ToDoModel {
    return ToDoModel(
      id = id,
      description = description,
      isCompleted = isCompleted,
      notes = notes,
      createdOn = createdOn
    )
  }

  @Dao
  interface Store {
    @Query("SELECT * FROM todos")
    fun all(): Flow<List<ToDoEntity>>

    @Query("SELECT * FROM todos WHERE id = :modelId")
    fun find(modelId: String): Flow<ToDoEntity?>
```

```
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun save(vararg entities: ToDoEntity)

    @Delete
    suspend fun delete(vararg entities: ToDoEntity)
  }
}
```

(from Room/src/main/java/com/commonsware/todo/repo/ToDoEntity.kt)

As with Retrofit, if you add the suspend keyword to a @Dao function, Room will know to code-generate an implementation that performs the database I/O in a CoroutineContext that uses a Room-supplied dispatcher. Room also supports using Flow as a return type, to give you both the initial results of the query and updated results should the database change while you are still collecting the results from the Flow. Once again, Room will supply a suitable dispatcher to get this work off of the main application thread.

And, if you do not have a suitable coroutines-ready API, you have the same basic choices as you would with a remote data source:

- Adapting a library that offers some other reactive API, such as one based on RxJava
- Setting up a bridge to an API that uses callbacks
- Wrapping a synchronous API in a withContext() call, to arrange to have that work be performed on a suitable dispatcher (e.g., Dispatchers.IO)

Repositories

Some repositories are mostly for orchestration of loading data from local or remote data stores. In those cases, the repositories might largely be a "pass-through" implementation, perhaps with some data conversion logic. In particular, if you are trying to have an optimized app model implementation independent from any data transfer objects (DTOs), the repository would handle conversion between those representations. For example, you might want to have model objects that differ from Retrofit responses or Room entities, and the repository would convert between those as needed.

Generally speaking, that sort of work is "simply" a matter of chaining conversion logic onto your existing data source coroutines API:

```kotlin
suspend fun saveModel(model: SomethingModel) = somethingDao.save(model.toEntity())

fun loadModels(): Flow<List<SomethingModel>> =
  somethingDao.load().map { entities -> entities.map { it.toModel() } }
```

Here, a fictitious `somethingDao` (perhaps a Room DAO implementation) might have a suspending `save()` function that saves an entity, and a corresponding `load()` function that loads all the entities with the results wrapped in a `Flow`. The repository functions, shown above, would use `toEntity()` and `toModel()` functions to convert between the types.

Things start to get more complicated if you are:

- Implementing an in-memory cache
- Supporting an on-disk cache with a local data source but also need to synchronize with some server

All of that logic would go in the repository. The repository still can consume the coroutines-based API from the data sources and can expose its own coroutines-based API.

Consuming Coroutines in a `ViewModel`

Frequently, our coroutines get consumed in a `ViewModel`. The `ViewModel` might be tasked with converting the model objects or other data into forms appropriate for rendering (e.g., converting an `Instant` into a formatted `String`). The `ViewModel` also often needs to deal with things like tracking whether the coroutine is still running (so we can show a loading state) or whether it threw an exception (so we can show an error state). And, we want the `ViewModel` to be able to handle the Android activity/fragment lifecycle, so we can get the results of our coroutines even if the user rotated the screen (or underwent another type of configuration change) while that work was proceeding.

viewModelScope

To consume coroutines in a `ViewModel`, we often use `viewModelScope`. This is an extension property on `ViewModel`, supplied by the `androidx.lifecycle:lifecycle-viewmodel-ktx` dependency. It maps to a `CoroutineScope` that is tied to the lifecycle of the `ViewModel` itself. When the `ViewModel` is cleared, the `CoroutineScope` is canceled.

The `MainMotor` in [the Retrofit module](#) uses `viewModelScope` to load the weather forecast from the `WeatherRepository`:

```kotlin
package com.commonsware.jetpack.weather

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class MainMotor(application: Application) : AndroidViewModel(application) {
  private val _results = MutableLiveData<MainViewState>()
  val results: LiveData<MainViewState> = _results

  fun load(office: String, gridX: Int, gridY: Int) {
    val scenario = Scenario(office, gridX, gridY)
    val current = _results.value

    if (current !is MainViewState.Content || current.scenario != scenario) {
      _results.value = MainViewState.Loading

      viewModelScope.launch(Dispatchers.Main) {
        val result = WeatherRepository.load(office, gridX, gridY)

        _results.value = when (result) {
          is WeatherResult.Content -> {
            val rows = result.forecasts.map { forecast ->
              val temp = getApplication<Application>()
                .getString(
                  R.string.temp,
                  forecast.temperature,
                  forecast.temperatureUnit
                )

              RowState(forecast.name, temp, forecast.icon)
            }

            MainViewState.Content(scenario, rows)
          }
          is WeatherResult.Error -> MainViewState.Error(result.throwable)
        }
      }
    }
```

```
    }
  }
}
```

`load()` opens a `CoroutineScope` using `viewModelScope`, and in there calls the `load()` suspend function from `WeatherRepository`. It then maps the results or exceptions into `MainViewState` objects, which it emits using a `MutableLiveData`. `MainViewState` represents the data that the UI will wind up rendering:

```kotlin
package com.commonsware.jetpack.weather

data class Scenario(val office: String, val gridX: Int, val gridY: Int)

sealed class MainViewState {
  object Loading : MainViewState()
  data class Content(val scenario: Scenario, val forecasts: List<RowState>) :
    MainViewState()

  data class Error(val throwable: Throwable) : MainViewState()
}
```

Dealing with Write Operations

For read operations, `viewModelScope` usually works fine. If the user navigates away from our activity or fragment, most likely we no longer need the data that we are trying to load. Canceling the `CoroutineScope` may save us some CPU time, network bandwidth, etc.

However, typically a *write* operation is one that we want to have succeed, even if the user navigates away. Just because they were quick to tap the BACK button does not mean that they want to lose the data they just entered and asked to save to the database or server.

There are a few approaches for addressing this problem.

NonCancellable

One approach is to mark the critical work as `NonCancellble`. This will tell the coroutines system to not cancel this coroutine, even if its scope is canceled. So, for example, a repository could wrap its Room DAO write calls in a `NonCancellable`

coroutine:

```
override suspend fun refresh() = withContext(NonCancellable) {
  db.observationStore().save(convertToEntity(remote.getCurrentObservation()))
}

override suspend fun clear() = withContext(NonCancellable) {
  db.observationStore().clear()
}
```

Or, in cases where the ViewModel is the one deciding what should and should not be canceled, NonCancellable can be added to the setup of the CoroutineScope:

```
viewModelScope.launch(Dispatchers.Main + NonCancellable) {
  // do cool stuff that should not be canceled here
}
```

This approach is not bad for "cleanup" code, such as things that might run in a finally block of a try/catch/finally structure. The downside is that the coroutine cannot be canceled at all, and the code in that coroutine might assume that it can be cancelled normally.

Custom Scope and withContext()

Another approach is to switch to a different CoroutineScope, one that will not be canceled by the user navigating away from our UI. For example, you could use GlobalScope… but we do not have a lot of control over the behavior of GlobalScope.

But, we can create our own CoroutineScope easily enough, as we saw earlier in the book.

That is illustrated using the hands-on tutorial profiled in an appendix as a basis. Like the Retrofit sample, the tutorial app collects weather forecasts. This time, though, we save past forecasts in a Room database, after fetching them from the US National Weather Service via Retrofit

This app uses Koin for dependency inversion, and it declares a singleton instance of a custom CoroutineScope, named appScope:

```
    single(named("appScope")) { CoroutineScope(SupervisorJob()) }
```

The recommendation is to use a CoroutineScope with a SupervisorJob. The SupervisorJob will ensure that any exceptions or intentional cancellations from one

coroutine will not cause other coroutines in that scope to be canceled.

We can then use this CoroutineScope in other places. Since we are using Koin, we can "inject" the appScope into other Koin-declared objects, such as a repository:

```
single<IObservationRepository> {
  ObservationRepository(
    get(),
    get(),
    get(named("appScope"))
  )
}
```

And the repository can, in turn, use that CoroutineScope to wrap existing suspend logic in a coroutine under that scope:

```
override suspend fun refresh() = withContext(appScope.coroutineContext) {
  db.observationStore()
    .save(convertToEntity(remote.getCurrentObservation())))
}

override suspend fun clear() = withContext(appScope.coroutineContext) {
  db.observationStore().clear()
}
```

Here, we use withContext() to switch to our own CoroutineContext.

Now, even if the caller of these repository functions (say, a ViewModel) cancels the coroutine that is executing refresh() or clear(), the nested coroutine that is doing the work (DAO calls of a Room database in this case) will continue to completion.

liveData()

Eventually, all this lovely reactive data needs to get to our UI (activity or fragment), also in a reactive fashion. One way to do that is to use LiveData, as that is both reactive and lifecycle-aware. Our viewmodel can adapt coroutines-based APIs (e.g., from a repository) into LiveData for the UI layer to consume.

One way to do that is via the liveData() builder. This extension function on ViewModel, found in the androidx.lifecycle:livecycle-livedata-ktx library, executes some lambda expression or other function type that you supply. It returns a LiveData that you can hold onto in a viewmodel property. For your lambda expression, this is set up to be a CoroutineScope tied to the ViewModel lifecycle. But

that `LiveDataScope` also has an `emit()` function — calling it sets the value on the `LiveData` that `liveData()` returns:

```kotlin
class LiveDataExamples {
  private val random = Random(SEED)

  fun liveSuspend() = liveData { emit(randomInt()) }

  private suspend fun randomInt(): Int = withContext(Dispatchers.IO) {
    delay(2000)
```

(from MiscSamples/src/main/java/com/commonsware/coroutines/misc/LiveDataExamples.kt)

This code appears in the `LiveDataExamples` Kotlin class in the MiscSamples module Here, we have a `randomInt()` suspend function that pretends that we have to do two seconds of I/O in order to generate a random number. The `liveSuspend()` function converts that into a `LiveData`. An activity or fragment could `observe()` that `LiveData` to both:

- Get the random number when it finally arrives
- Get that same random number back on a configuration change

This is obviously a fairly contrived example, as it does not take two seconds and a background thread to generate a random number.

But, you can use `liveData()` to adapt any sort of suspend function. In *Elements of Android Room*, there is an `FTS` sample app that demonstrates Room's support for SQLite's full-text search capability. It has a `BookRepository` that has `all()` and `filtered()` suspend functions that return either all of the paragraphs in a book or those matching some search expression. The viewmodels for the UI convert those into `LiveData` using `liveData()`:

```kotlin
class BookViewModel(private val repo: BookRepository) : ViewModel() {
  val paragraphs = liveData { emit(repo.all()) }
}
```

```kotlin
class SearchViewModel(
  private val search: String,
  private val repo: BookRepository
) : ViewModel() {
  val paragraphs: LiveData<List<String>> = liveData { emit(repo.filtered(search)) }
}
```
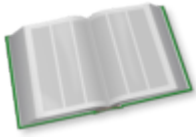
> You can learn more about Room and full-text search in the "Room and Full-Text Search" chapter of *Elements of Android Room*!

Note that `liveData()` accepts a `CoroutineContext` as a parameter. This can include the dispatcher used for the `CoroutineScope`:

```
val results: LiveData<Something> = liveData(Dispatchers.Default) {
emit(somethingSupplier()) }
```

Also note that the `LiveDataScope` in which our lambda expression is executed has not only `emit()` but `emitSource()`. `emit()` takes an object to be emitted; `emitSource()` takes another `LiveData` of objects to be emitted. This works akin to `addSource()` on a `MediatorLiveData`, where any objects emitted by the `LiveData` passed into `emitSource()` are themselves emitted by the `LiveData` that `liveData()` returns:

```
val results: LiveData<Something> = liveData { emitSource(liveStreamOfSomethings()) }
```

The combination of `emit()` and `emitSource()` might be useful when you want a single `LiveData` to stem from two bits of I/O:

- Load the initial data from a disk cache via `emit()`
- Load ongoing changes from some network-based supplier via `emitSource()`

asLiveData()

If you have a `Flow` that you want to expose as `LiveData`, there is an `asLiveData()` extension function defined in the `androidx.lifecycle:livecycle-livedata-ktx` library. It takes a `CoroutineContext` as a parameter, allowing you to define the dispatcher to use for collecting the `Flow`:

```
fun liveFlow() = randomPercentages(3, 500).asLiveData(Dispatchers.IO)

private fun randomPercentages(count: Int, delayMs: Long) = flow {
  for (i in 0 until count) {
    delay(delayMs)
    emit(random.nextInt(1, 100))
  }
}
```

(from MiscSamples/src/main/java/com/commonsware/coroutines/misc/LiveDataExamples.kt)

Here, in randomPercentages(), we create a Flow using the flow() function to return a stream of random numbers, with an artificial delay between them. The liveFlow() function simply calls randomPercentages() with a specific count and delay period, then converts the Flow into a LiveData.

Similarly, in the Room sample module, our ToDoRepository exposes an items() function as a Flow. In RosterMotor, we map() the model objects into a RosterViewState and expose that viewstate via LiveData using asLiveData():

```
package com.commonsware.todo.ui.roster

import androidx.lifecycle.LiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.asLiveData
import com.commonsware.todo.repo.ToDoModel
import com.commonsware.todo.repo.ToDoRepository
import kotlinx.coroutines.GlobalScope
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.launch

class RosterViewState(
  val items: List<ToDoModel> = listOf()
)

class RosterMotor(private val repo: ToDoRepository) : ViewModel() {
  val states: LiveData<RosterViewState> =
    repo.items().map { RosterViewState(it) }.asLiveData()

  fun save(model: ToDoModel) {
    GlobalScope.launch {
      repo.save(model)
    }
  }
}
```

(from Room/src/main/java/com/commonsware/todo/ui/roster/RosterMotor.kt)

The RosterListFragment can then observe() the LiveData to work with the list of to-do items.

Other LiveData Updates

Those approaches — the liveData() builder and asLiveData() — work well when the value of your LiveData is uniquely determined from something that you can readily capture in a lambda expression (for liveData()) or in chained operators on a

Flow (for `asLiveData()`).

Many times, though, what you want to emit is more complicated than that. In those cases, simply consuming the coroutines "normally" and using them to emit values on a `MutableLiveData` is a reasonable approach.

We saw this earlier in [the section on viewModelScope](). `MainMotor` in the `Retrofit` example launches a coroutine to get the weather forecast (where needed) and updates a `MutableLiveData` as a result:

```kotlin
package com.commonsware.jetpack.weather

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.launch

class MainMotor(application: Application) : AndroidViewModel(application) {
  private val _results = MutableLiveData<MainViewState>()
  val results: LiveData<MainViewState> = _results

  fun load(office: String, gridX: Int, gridY: Int) {
    val scenario = Scenario(office, gridX, gridY)
    val current = _results.value

    if (current !is MainViewState.Content || current.scenario != scenario) {
      _results.value = MainViewState.Loading

      viewModelScope.launch(Dispatchers.Main) {
        val result = WeatherRepository.load(office, gridX, gridY)

        _results.value = when (result) {
          is WeatherResult.Content -> {
            val rows = result.forecasts.map { forecast ->
              val temp = getApplication<Application>()
                .getString(
                  R.string.temp,
                  forecast.temperature,
                  forecast.temperatureUnit
                )

              RowState(forecast.name, temp, forecast.icon)
            }
```

```
            MainViewState.Content(scenario, rows)
          }
          is WeatherResult.Error -> MainViewState.Error(result.throwable)
        }
      }
    }
  }
}
```

(from Retrofit/src/main/java/com/commonsware/jetpack/weather/MainMotor.kt)

The `LiveData` Alternatives

`LiveData` was created before Google endorsed Kotlin, let alone went "Kotlin-first" for their efforts. `LiveData` also was created before coroutines had a stable release.

So, many Android developers working in Kotlin are moving away from `LiveData` and to other options. Those options tie in coroutines types like `StateFlow` with lifecycle-aware `CoroutineScope` options. We will look at this approach later in the chapter.

Lifecycles and Coroutines

If you want to consume coroutines from an Android activity or fragment, it would be nice to have a `CoroutineScope` that we could use that was lifecycle-aware. That way, just as `LiveData` can clean up its observers when the lifecycle advances to a destroyed state, so too could a lifecyle-aware `CoroutineScope` cancel itself when needed.

And, as it turns out, the `lifecycle-runtime-ktx` library contains some extension functions for `LifecycleOwner` and `Lifecycle` that do just that.

lifecycleScope

`AppCompatActivity`, the AndroidX implementation of `Fragment`, and other implementations of `LifecycleOwner` offer a `lifecycleScope` extension property. This works similarly to `viewModelScope` in a `ViewModel`, and it returns a `CoroutineScope` that is tied to the lifecycle of the `LifecycleOwner`. When the owner advances to a destroyed state, the `CoroutineScope` is canceled. So, just as you use `viewModelScope` to consume coroutines in a `ViewModel`, you can use `lifecycleScope` to consume coroutines in a `LifecycleOwner`, knowing that the scope will get cleaned

up at a reasonable time.

We will see examples of this [later in this chapter](#).

when...() Functions

A more esoteric set of options are the when...() functions on Lifecycle:

- whenCreated()
- whenStarted()
- whenResumed()
- whenStateAtLeast()

These each take a lambda expression or other function type. The current context (this) is set to be a CoroutineScope, so you can use this when consuming coroutines. And, as the names suggest, that lambda expression will be executed when the Lifecycle reaches a particular state.

The idea is that you can arrange to do work when particular lifecycle events occur, and you can launch coroutines or consume flows and channels as part of that processing. The CoroutineScope used for the lambda expression is the same basic scope as lifecycleScope gives you, so it will be canceled once the associated Lifecycle is destroyed. It will also be canceled once the lifecycle leaves the requested state — whenResumed(), for example, will be canceled once the lifecycle leaves the resumed state (e.g., after onPause() is called on the activity/fragment).

[The documentation for whenStateAtLeast()](#) lists a bunch of limitations. A key one is that exception handlers (catch and finally) might be executed after the Lifecycle is destroyed, so it may not be safe to update the UI from there.

There are equivalent coroutine builders on the lifecycleScope, filling the same role with slightly different syntax:

- launchWhenCreated()
- launchWhenStarted()
- launchWhenResumed()

Immediate Dispatch

The scopes supplied by the Jetpack libraries — viewModelScope and lifecycleScope

among them — do not use `Dispatchers.Default` as the default dispatcher. Instead, they use `Dispatchers.Main.immediate` as the default.

`Dispatchers.Main` is tied to the "main application thread" of some UI framework. In the case of Android development, that is Android's main application thread.

As we saw [earlier in the book](), `Dispatchers.Main.immediate` is a special type of dispatcher. Normally, coroutines placed onto a dispatcher are always suspended, then get resumed when the dispatcher has a thread ready to run that coroutine. With `Dispatchers.Main.immediate`, if we are already on the main application thread, the coroutine code is run immediately, and it only suspends if it calls some `suspend` function, particularly one tied to a different dispatcher.

This is roughly analogous to the difference between `post()` on `Handler` or `View` and `runOnUiThread()` on `Activity`. Both take a `Runnable` and both run that `Runnable` on the main application thread. However:

- `post()` always puts the `Runnable` on the main application thread's work queue, so it will not get run until it reaches the head of that queue
- `runOnUiThread()` immediately runs the `Runnable` if we happen to be on the main application thread right now, only putting the `Runnable` on the queue if we are on a background thread

Generally speaking, the difference between `Dispatchers.Main` and `Dispatchers.Main.immediate` is relatively minor. In either case, our coroutine is run on the main application thread. `Dispatchers.Main.immediate` is incrementally more efficient if we are already on the main application thread at the time, and they are equivalent in other cases.

Bypassing `LiveData`

`LiveData` is a value holder that has a lifecycle-aware observer framework associated with it. But that is all that it is. If we wanted to replace `LiveData`, we need something that:

- Can accept data
- Can deliver that data to N observers
- Can hold onto the last piece of data and deliver it to new observers automatically
- Can clean up properly when its last observer is destroyed

As we saw [in a previous chapter](#), StateFlow was added in version 1.3.6 of the coroutines library. It is the long-term option for handling LiveData-style scenarios.

[The RetrofitStateFlow module](#) of [the book's primary sample Android project](#) shows using StateFlow and lifecycleScope as an alternative to LiveData. MainMotor uses a MutableStateFlow:

```kotlin
package com.commonsware.jetpack.weather

import android.app.Application
import androidx.lifecycle.AndroidViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.launch

class MainMotor(application: Application) : AndroidViewModel(application) {
  private val _results = MutableStateFlow<MainViewState>(MainViewState.Loading)
  val results: Flow<MainViewState> = _results

  fun load(office: String, gridX: Int, gridY: Int) {
    val scenario = Scenario(office, gridX, gridY)
    val current = _results.value

    if (current !is MainViewState.Content || current.scenario != scenario) {
      _results.value = MainViewState.Loading

      viewModelScope.launch(Dispatchers.Main) {
        val result = WeatherRepository.load(office, gridX, gridY)

        _results.value = when (result) {
          is WeatherResult.Content -> {
            val rows = result.forecasts.map { forecast ->
              val temp = getApplication<Application>()
                .getString(
                  R.string.temp,
                  forecast.temperature,
                  forecast.temperatureUnit
                )

              RowState(forecast.name, temp, forecast.icon)
            }

            MainViewState.Content(scenario, rows)
          }
          is WeatherResult.Error -> MainViewState.Error(result.throwable)
```

```
        }
      }
    }
  }
}
```

MainMotor exposes that `MutableStateFlow` to outside parties as a `Flow`.
MainActivity then consumes that `Flow`, using `onEach()` and `launchIn()`:

```
motor.results.onEach { state ->
  when (state) {
    MainViewState.Loading -> binding.progress.visibility = View.VISIBLE
    is MainViewState.Content -> {
      binding.progress.visibility = View.GONE
      adapter.submitList(state.forecasts)
    }
    is MainViewState.Error -> {
      binding.progress.visibility = View.GONE
      Toast.makeText(
        this@MainActivity, state.throwable.localizedMessage,
        Toast.LENGTH_LONG
      ).show()
      Log.e("Weather", "Exception loading data", state.throwable)
    }
  }
}.launchIn(lifecycleScope)
```

Events to Your UI

However, sometimes, we want to do something exactly once, as the "something" has
its own lifecycle and handles configuration changes on its own. Examples include:

- The viewmodel determines that we need to request a permission, so it needs
  to signal to the activity or fragment that we need to call
  `requestPermissions()` to display some system-supplied UI to ask the user to
  grant us the permission. That is an independent activity (albeit one styled
  like a dialog), so it handles its own configuration changes.
- The viewmodel catches an exception, and the UI calls for showing a dialog.
  The viewmodel needs to tell the activity or fragment to show that dialog,
  and our code to show the dialog uses a `DialogFragment`, which will
  automatically re-display the dialog on a configuration change.

- The viewmodel completes some work, and the next step is to navigate to the next screen. So, the viewmodel needs to signal to the activity or fragment to perform that navigation, whether using the Navigation component from the Jetpack, a manual `FragmentTransaction`, a `startActivity()` call, or something else.

`LiveData` Is Too Live

This is a problem with `LiveData`, as that is designed specifically to retain and redeliver the last-received piece of data after a configuration change. Suppose that our viewmodel considers the exception to be part of some viewstate and emits a fresh viewstate with that exception when the exception is caught. Our activity or fragment observes the `LiveData`, gets the new viewstate, sees that we have the exception, and shows the `DialogFragment`.

While that `DialogFragment` is visible, the user rotates the screen, toggles dark mode, or otherwise causes a configuration change.

Our activity and its fragments will be destroyed and recreated as part of standard configuration change processing. When the `DialogFragment` is recreated, it will display the dialog again. However, our `LiveData` still has the exception-laden viewstate, so when we observe that in the new activity or fragment, we see the exception, and we show *another* instance of the `DialogFragment`. This gives us two overlapping dialogs, or more if the user triggers more configuration changes.

In the case of this error dialog, we could avoid this problem by not using `DialogFragment`. Instead, we could just display the `AlertDialog` directly. That will get destroyed along with the activity and its fragments when the configuration change occurs, and it will not be recreated automatically by a `DialogFragment` (since we are not using one). When we observe the viewstate and see the exception, though, we would just show the `AlertDialog` again.

And, in that scenario, everything is fine.

However, that doesn't work for all of the scenarios. In the `requestPermissions()` scenario, while *visually* it looks the same as if we were showing some `AlertDialog`, *operationally* we are starting some system-supplied activity. That activity will handle configuration changes… but since that system-supplied activity is styled like a dialog, *our activity will also be destroyed and recreated*, since our activity is visible when the configuration change occurs. If our decision to call `requestPermissions()` is based on some viewstate that we are observing via `LiveData`, the new activity or fragment

will call requestPermissions() again, and we wind up with two overlapping permission dialogs.

We need a better solution.

What Came Before: "Single Live Event" Pattern

Google's cadre of Android developer advocates recognized this problem as they started applying LiveData to projects, and they offered up a solution: the "single live event" pattern.

This involves some sort of Event class that wraps some other object and tracks whether this Event has been handled or not:

```
class Event<out T>(private val content: T) {
  private var hasBeenHandled = false

  fun handle(handler: (T) -> Unit) {
    if (!hasBeenHandled) {
      hasBeenHandled = true
      handler(content)
    }
  }
}
```

You then wrap some object representing the event (e.g., a Throwable) with an Event and have a LiveData of those events:

```
private val _events = MutableLiveData<Event<Throwable>>()
val events: LiveData<Event<Throwable>> = _events
```

Your observer then gets the actual content (the Throwable) via the handle() function, which only executes the supplied lambda expression (or other function type) if the content has not already been handled:

```
motor.events.observe(this) { event ->
  event.handle { throwable ->
    showErrorDialog()
  }
}
```

The first time the Event is observed, we handle it. If that same Event gets observed again, after a configuration change, handle() sees that the Event was already

handled and skips running our showErrorDialog() code.

This works. However, it's a bit of a hack. We are having to use this Event wrapper to disable a LiveData feature: retaining and redelivering data after a configuration change. And, if we were to replace LiveData with a MutableStateFlow, we would wind up with the same problem and the same need for Event.

The New Hotness: Channel

Nowadays, Google engineers are steering us away from the single-live-event pattern and to... well, frankly, just about anything else for events. And, in the coroutines space, "just about anything else" is a Channel.

The WhereWeAt module retrieves our location from LocationManager and shows it on the screen. However, we have to take into account that we might not have ACCESS_FINE_LOCATION permission at the time, such as when we first run the app. And, there might be some other problem arising from our attempt to get the location fix.

To that end, MainMotor has:

- A MutableStateFlow to represent loading/content/error states
- A Channel to use for permission events

We can then post states to the MutableStateFlow or send() events on the Channel representing the work that was done, in this case in a loadLocations() function:

```kotlin
sealed class ViewState {
  object Loading : ViewState()
  data class Content(val location: Location) : ViewState()
  object Error : ViewState()
}

private val _results = MutableStateFlow<ViewState>(ViewState.Loading)
val results = _results.asStateFlow()
private val _permissions = Channel<List<String>>()
val permissions = _permissions.receiveAsFlow()
```

(from WhereWeAt/src/main/java/com/commonsware/coroutines/location/MainMotor.kt)

We can then

```
package com.commonsware.coroutines.location

import android.Manifest
import android.annotation.SuppressLint
import android.content.Context
import android.content.pm.PackageManager
import android.location.Location
import android.location.LocationListener
import android.location.LocationManager
import android.os.Bundle
import android.os.Handler
import android.os.HandlerThread
import android.util.Log
import androidx.core.content.ContextCompat
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.android.asCoroutineDispatcher
import kotlinx.coroutines.channels.Channel
import kotlinx.coroutines.flow.MutableStateFlow
import kotlinx.coroutines.flow.asStateFlow
import kotlinx.coroutines.flow.receiveAsFlow
import kotlinx.coroutines.launch

private const val PERM = Manifest.permission.ACCESS_FINE_LOCATION

class MainMotor(
  private val locationManager: LocationManager,
  private val context: Context
) : ViewModel() {
  sealed class ViewState {
    object Loading : ViewState()
    data class Content(val location: Location) : ViewState()
    object Error : ViewState()
  }

  private val _results = MutableStateFlow<ViewState>(ViewState.Loading)
  val results = _results.asStateFlow()
  private val _permissions = Channel<List<String>>()
  val permissions = _permissions.receiveAsFlow()
  private val handlerThread = HandlerThread("WhereWeAt").apply { start() }
  private val handler = Handler(handlerThread.looper)
  private val dispatcher = handler.asCoroutineDispatcher()

  @SuppressLint("MissingPermission")
  fun loadLocation() {
    viewModelScope.launch(dispatcher) {
      if (ContextCompat.checkSelfPermission(context, PERM) == PackageManager.PERMISSION_GRANTED) {
        val listener = object : LocationListener {
          override fun onLocationChanged(location: Location) {
            _results.value = ViewState.Content(location)
          }

          override fun onStatusChanged(
            provider: String,
            status: Int,
            extras: Bundle?
          ) {
            // unused
          }
```

```kotlin
        override fun onProviderEnabled(provider: String?) {
          // unused
        }

        override fun onProviderDisabled(provider: String?) {
          // unused
        }
      }

      try {
        locationManager.requestSingleUpdate(
          LocationManager.GPS_PROVIDER,
          listener,
          null
        )
      } catch (t: Throwable) {
        Log.e("WhereWeAt", "Error getting location", t)
        _results.value = ViewState.Error
      }
    } else {
      _permissions.send(listOf(PERM))
    }
  }
}

override fun onCleared() {
  super.onCleared()

  handlerThread.quitSafely()
}
}
```

(we will look at some strange aspects of this, like that HandlerThread, a bit later in this chapter)

The Channel is exposed as a Flow, using receiveAsFlow(). Our MainActivity can then use the onEach().launchIn() pattern that we used with Flow for viewstates for both the MutableStateFlow and the Channel flow:

```kotlin
    motor.results.onEach { state ->
      when (state) {
        MainMotor.ViewState.Loading -> {
          binding.loading.isVisible = true
          binding.content.isVisible = false
        }
        is MainMotor.ViewState.Content -> {
          binding.loading.isVisible = false
          binding.content.isVisible = true
          binding.content.text =
            getString(
              R.string.location,
```

```
            state.location.latitude,
            state.location.longitude
          )
        }
      MainMotor.ViewState.Error -> {
        binding.loading.isVisible = false
        binding.content.isVisible = true
        binding.content.text = getString(R.string.error_message)
      }
    }
  }
    .launchIn(lifecycleScope)

  motor.permissions
    .onEach { requestPermissions(it.toTypedArray(), REQUEST_PERMS) }
    .launchIn(lifecycleScope)
```

(from WhereWeAt/src/main/java/com/commonsware/coroutines/location/MainActivity.kt)

The activity is not concerned about the differences between the viewstates and the permission events — it just consumes what is offered and does what is needed:

- Displays the location in the UI
- Displays a DialogFragment for an error
- Requests the desired permissions from the user

Whether the data is retained across configuration changes and re-delivered, or whether it is a one-time-only event, is handled by the ViewModel and its choice of whether to use MutableStateFlow or Channel.

Coroutines and Handler/Looper

A bunch of Android APIs take a Handler or a Looper as a parameter. Occasionally, you can get away with using a Handler tied to the main application thread (e.g., Handler(Looper.getMainLooper()). Other times, though, you want the results to be delivered on some background thread.

One Android pattern for this is to use HandlerThread. HandlerThread is an ordinary Thread that you create and start(). That thread is set up to have a Looper, which you can either use directly or wrap in a Handler.

On its own, HandlerThread knows nothing about coroutines.

However, coroutines offers an extension function on `Handler`, `asCoroutineDispatcher()`. This works similarly to `asCoroutineDispatcher()` on an `Executor`, giving you a dispatcher that you can use with `launch()`, `withContext()`, etc.

The `WhereWeAt` sample from the preceding section uses this approach. The third parameter to `requestSingleUpdate()` on `LocationManager` takes a `Looper` to use to deliver results to the `LocationListener` callback. If you pass `null`, as we did, it will try to use the `Looper` associated with the current thread. That requires `requestSingleUpdate()` to be called on a suitable thread, one that is set up with a `Looper`... such as a `HandlerThread`.

`MainMotor` sets up a `HandlerThread` and `Handler`, from which it creates a dispatcher:

```kotlin
private val handlerThread = HandlerThread("WhereWeAt").apply { start() }
private val handler = Handler(handlerThread.looper)
private val dispatcher = handler.asCoroutineDispatcher()
```

(from WhereWeAt/src/main/java/com/commonsware/coroutines/location/MainMotor.kt)

That `dispatcher` is used in our `viewModelScope.launch()` call, so our coroutine will run on that `HandlerThread`. And, this means that when we pass `null` to `requestSingleUpdate()`, `LocationManager` can use the `Looper` associated with the `HandlerThread` that our coroutine is running on.

In practice, we do not need the coroutine here. `requestSingleUpdate()` is fairly inexpensive, with all the location-fix delays occurring on a background thread. However, in theory, we might be doing other work as part of this coroutine (Web service call, database I/O, etc.) and might need it.

"But I Just Want an `AsyncTask`!"

You may have existing code that uses `AsyncTask`. With that being officially deprecated, it is time to move off of `AsyncTask` and onto something else. And, since this book is about coroutines, it would seem like coroutines would be a possible solution here.

In the end, an `AsyncTask` is made up of 2-3 blocks of code:

- The code that you want to run on a background thread (`doInBackground()`)
- The code that you want to run on the main application thread when the

background work completes (`onPostExecute()`)
- Possibly some code that you want to run on the main application thread when the background thread "publishes progress" (`onProgressUpdate()`)

For simplicity, let's ignore the third block for now, and assume that you have not used `onProgressUpdate()`. For the other two, you can migrate to coroutines progressively, simplifying your code along the way.

Step #0: Write Instrumented Tests

After all, you want to know if this conversion works correctly, so having tests is important. Since we are going to be referring to `AsyncTask` temporarily, those tests need to be instrumented tests that run on Android, instead of unit tests that run on your development machine or directly on your CI server.

Step #1: Convert to Kotlin

Obviously, to use coroutines, you will need to be using Kotlin. So, if your code that implements and uses the `AsyncTask` is in Java, you will need to convert that to Kotlin, then confirm that your tests still work, your app still compiles, etc.

Step #2: Refactor to Lambda Expressions

`AsyncTask` is a very Java solution to a problem, in that it requires you to create a custom subclass. In Kotlin, while creating classes is clearly needed a lot of the time, for simpler scenarios we often like to just stick with functions and lambda expressions.

So, you could add a class like this to your project:

```kotlin
class SimpleTask<T>(
  private val background: () -> T,
  private val post: (T) -> Unit
) : AsyncTask<Unit, Unit, T>() {
  override fun doInBackground(vararg params: Unit): T {
    return background()
  }

  override fun onPostExecute(result: T) {
    post(result)
  }
}
```

This is a reusable `AsyncTask` implementation that pulls the bodies of `doInBackground()` and `onPostExecute()` out into `background` and `post` properties on the constructor. `background` returns some object and `post()` will be passed that object. Via generics, we can just treat that as some arbitrary type `T`.

You might even consider adding a top-level `asyncTask()` function to handle running the task:

```kotlin
fun <T> asyncTask(
  background: () -> T,
  executor: Executor = AsyncTask.THREAD_POOL_EXECUTOR,
  post: (T) -> Unit
) {
  SimpleTask(background, post).executeOnExecutor(executor)
}
```

This allows you to take something like:

```kotlin
class SomethingTask : AsyncTask<Unit, Unit, String>() {
  override fun doInBackground(vararg params: Unit?): String {
    return TODO("Do something useful")
  }

  override fun onPostExecute(result: String) {
    // TODO something with the value from doInBackground
  }
}

// somewhere later

SomethingTask().executeOnExecutor(AsyncTask.THREAD_POOL_EXECUTOR)
```

(where, in your project, those `TODO` elements would be your actual code)

...and replace it with:

```kotlin
asyncTask({ TODO("Do something useful")}) { result ->
  // TODO something with the value from the first lambda expression
}
```

We are still using `AsyncTask`, but the result is simpler.

Step #3: Swap In Coroutines

Now, we need to replace asyncTask() with something else that uses coroutines.

The biggest difference between coroutines and AsyncTask lies in structured concurrency. AsyncTask is largely "fire and forget" — while we could hold onto the AsyncTask and try to cancel() it later, we often do not do that. Coroutines instead require you to think through things like cancellation and exceptions, by requiring a CoroutineScope.

But we have lots of CoroutineScope objects to choose from, such as viewModelScope and lifecycleScope. So, having an appropriate CoroutineScope probably is not a problem.

So, we could have an extension function on CoroutineScope that mimics the API for asyncTask(), such as:

```kotlin
fun <T> CoroutineScope.asyncTaskAlt(
  background: () -> T,
  dispatcher: CoroutineDispatcher = AsyncTask.THREAD_POOL_EXECUTOR.asCoroutineDispatcher(),
  post: (T) -> Unit
) {
  launch(Dispatchers.Main) {
    post(withContext(dispatcher) {
      background()
    })
  }
}
```

(from MiscSamples/src/main/java/com/commonsware/coroutines/misc/AsyncTaskAlt.kt)

Now, you can switch:

```kotlin
asyncTask({ TODO("Do something useful")}) { result ->
  // TODO something with the value from the first lambda expression
}
```

to:

```kotlin
lifecycleScope.asyncTaskAlt({ TODO("Do something useful")}) { result ->
  // TODO something with the value from the first lambda expression
}
```

Immediately, we gain automatic cancellation of the work if our scope is cleared, something that we would have to handle manually with AsyncTask (and often fail to do).

By default, `asyncTaskAlt()` uses the same thread pool that `AsyncTask` does (`AsyncTask.THREAD_POOL_EXECUTOR`). However, later on, you could consider switching to a more conventional dispatcher (e.g., `Dispatchers.Default`).

Stuff That's Missing

For `publishProgress()` and `onProgressUpdate()`, you could use a `Channel`.

`AsyncTask` supports the notion of passing in parameters to `executeOnExecutor()`, which in turn get passed to `doInBackground()`. That is not needed here, as you can just capture those values in the lambda expression that you use for the background work.

Coroutines and Views

The Android SDK offers a callback-driven API. Sometimes, those callbacks come in the form of methods that you override in your main classes, such as `onCreate()` in an activity or `onCreateView()` in a fragment. Other times, you provide callback or listener objects to SDK methods, which then invoke those objects when events occur.

In Kotlin, our first thought is to use lambda expressions for these, for the simplest syntax. Sometimes, SAM conversion means that we can do this out of the box:

```
myButton.setOnClickListener {
  // TODO something nice
}
```

Other times, Android KTX offers extension functions, such as `doAfterTextChanged()` on `TextView`, to allow for lambda expressions when the underlying callback object is more complex (e.g., a `TextWatcher`):

```
myField.doAfterTextChanged { text ->
  // format the text, or update some backing store, or whatevs
}
```

However, these lambda expressions are not lifecycle-aware. It is possible that the underlying API winds up being lifecycle-aware, but that is far from certain. It is far too easy to wind up in a situation where your callback gets called but the hosting activity or fragment is destroyed and you cannot safely do what you intended to do.

While a lot of the focus on coroutines is using them for background work, the fact that they have built-in cancellation capability means that they are also useful for adding lifecycle awareness to our code.

Google's Chris Banes talked about this in [a Medium post](#), demonstrating a few applications of coroutines for UI. One is a coroutines-based implementation of Android KTX's doOnNextLayout() extension function:

```kotlin
// inspired by https://chris.banes.dev/2019/12/03/suspending-views/

suspend fun View.waitForNextLayout() =
  suspendCancellableCoroutine<Unit> { continuation ->
    val onChange = object : View.OnLayoutChangeListener {
      override fun onLayoutChange(
        v: View?,
        left: Int,
        top: Int,
        right: Int,
        bottom: Int,
        oldLeft: Int,
        oldTop: Int,
        oldRight: Int,
        oldBottom: Int
      ) {
        removeOnLayoutChangeListener(this)

        continuation.resume(Unit) { }
      }
    }

    continuation.invokeOnCancellation { removeOnLayoutChangeListener(onChange) }

    addOnLayoutChangeListener(onChange)
  }
```

(from [WidgetCoroutines/src/main/java/com/commonsware/android/coroutines/widget/MainActivity.kt](#))

Here, we use suspendCancellableCoroutine() to adapt the OnLayoutChangeListener callback-style API to a coroutine:

- Our listener, upon receiving the callback, unregisters itself and resumes the supplied continuation to unblock the calling coroutine
- If the coroutine is canceled while we are waiting for the callback, we unregister the listener
- When everything is set up, we register the listener

waitForNextLayout() can then be used in a suitable coroutine, such as one launched from an activity's lifecycleScope:

```kotlin
lifecycleScope.launch(Dispatchers.Main) {
  Log.d(TAG, "before waiting for layout")

  binding.button.waitForNextLayout()

  Log.d(TAG, "after waiting for layout")
}
```

(from WidgetCoroutines/src/main/java/com/commonsware/android/coroutines/widget/MainActivity.kt)

Inside that coroutine, we can do work before or after we block waiting for the next layout of the widget (in this case, a Button). The setup code is nice and imperative, yet we still will handle cases where the activity is destroyed while we are waiting.

suspendCancellableCoroutine() is fine for one-shot events like this. For ongoing events, callbackFlow() provides a similar API, but gives us a Flow instead. For example, you can adapt Android's TextWatcher for observing text entry into an EditText:

```kotlin
fun EditText.textEntryFlow() =
  callbackFlow<Editable> {
    val watcher = object : TextWatcher {
      override fun afterTextChanged(s: Editable) {
        offer(s)
      }

      override fun beforeTextChanged(
        s: CharSequence?,
        start: Int,
        count: Int,
        after: Int
      ) {
        // unused
      }

      override fun onTextChanged(
        s: CharSequence?,
        start: Int,
        before: Int,
        count: Int
      ) {
        // unused
      }
    }

    addTextChangedListener(watcher)
```

```
    awaitClose { removeTextChangedListener(watcher) }
  }
```

(from [WidgetCoroutines/src/main/java/com/commonsware/android/coroutines/widget/MainActivity.kt](#))

When the `TextWatcher` gets text in `afterTextChanged()`, we `offer()` it. We then add the `TextWatcher` via `addTextChangedListener()` and call `awaitClose()` to block until the `Flow` loses its consumer, where we then remove the `TextWatcher`.

This too can be consumed by a coroutine launched from `lifecycleScope`:

```
  lifecycleScope.launch(Dispatchers.Main) {
    binding.field.textEntryFlow()
      .debounce(500)
      .collect {
        Log.d(TAG, "text entered: $it")
      }
  }
```

(from [WidgetCoroutines/src/main/java/com/commonsware/android/coroutines/widget/MainActivity.kt](#))

Here, we use the `debounce()` operator on `Flow` to only pay attention to entries after a 500ms quiet period, so we are not flooded with events as the user is typing. Then, we `collect()` the resulting `Flow` and consume the text input.

Coroutines and Platforms

Java Interoperability

This chapter will be added in a future version of this book!

Appendices

Appendix A: Hands-On Converting RxJava to Coroutines

This appendix offers a hands-on tutorial, akin to those from *Exploring Android*. In this tutorial, we will convert an app that uses RxJava to have it use coroutines instead.

To be able to follow along in this tutorial, it will help to read the first few chapters of this book, plus have basic familiarity with how RxJava works. You do not need to be an RxJava expert, let alone a coroutines expert, though.

You can download the code for the initial project and follow the instructions in this tutorial to replace RxJava with coroutines. Or, you can download the code for the project after the changes to see the end result.

About the App

The WeatherWorkshop app will display the weather conditions from the US National Weather Service KDCA weather station. This weather station covers the Washington, DC area.

When you launch the app, it will immediately attempt to contact the National Weather Service's Web service API to get the current conditions at KDCA (an "observation"). Once it gets them, the app will save those conditions in a database, plus display them in a row of a list:
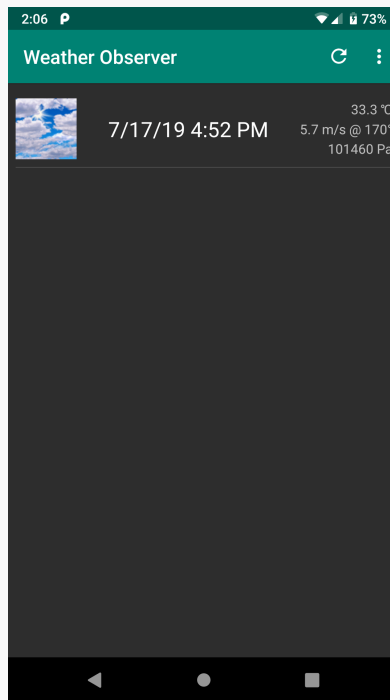
Figure 11: Weather Workshop Sample App, Showing Weather Info

The "refresh" toolbar button will attempt to get a fresh set of current conditions. However, the weather station only reports conditions every hour or so. If you wind up getting the same set of current conditions as before, the "new" set is not added to the database or the list. If, however, you request the current conditions substantially later than the most-recent entry, the new conditions will be added to the database and list.

The overflow menu contains a "clear" option that clears the database and the list. You can then use "refresh" to request the current conditions.

Step #1: Reviewing What We Have

First, let's spend a bit reviewing the current code, so you can see how it all integrates and how it leverages RxJava.

ObservationRemoteDataSource

This app uses Retrofit for accessing the National Weather Service's Web service. The Retrofit logic is encapsulated in an ObservationRemoteDataSource:

```kotlin
package com.commonsware.coroutines.weather

import io.reactivex.Single
import okhttp3.OkHttpClient
import retrofit2.Retrofit
import retrofit2.adapter.rxjava2.RxJava2CallAdapterFactory
import retrofit2.converter.moshi.MoshiConverterFactory
import retrofit2.http.GET
import retrofit2.http.Path

private const val STATION = "KDCA"
private const val API_ENDPOINT = "https://api.weather.gov"

class ObservationRemoteDataSource(ok: OkHttpClient) {
  private val retrofit = Retrofit.Builder()
    .client(ok)
    .baseUrl(API_ENDPOINT)
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .addConverterFactory(MoshiConverterFactory.create())
    .build()
  private val api = retrofit.create(ObservationApi::class.java)

  fun getCurrentObservation() = api.getCurrentObservation(STATION)
}

private interface ObservationApi {
  @GET("/stations/{stationId}/observations/current")
  fun getCurrentObservation(@Path("stationId") stationId: String): Single<ObservationResponse>
}

data class ObservationResponse(
  val id: String,
  val properties: ObservationProperties
)

data class ObservationProperties(
  val timestamp: String,
  val icon: String,
  val temperature: ObservationValue,
  val windDirection: ObservationValue,
  val windSpeed: ObservationValue,
  val barometricPressure: ObservationValue
)

data class ObservationValue(
  val value: Double?,
  val unitCode: String
)
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRemoteDataSource.kt)

Most of ObservationRemoteDataSource could work with any weather station; we

hard-code "KDCA" to keep the app simpler.

We have just one function in our ObservationApi: getCurrentObservation(). We are using Retrofit's RxJava extension, and getCurrentObservation() is set up to return an RxJava Single, to deliver us a single response from the Web service. That response is in the form of an ObservationResponse, which is a simple Kotlin class that models the JSON response that we will get back from the Web service.

Of note in that response:

- id in ObservationResponse is a server-supplied unique identifier for this set of conditions
- The actual weather conditions are in a properties property
- icon holds the URL to an icon that indicates the sort of weather that DC is having (sunny, cloudy, rain, snow, sleet, thunderstorm, kaiju attack, etc.)
- The core conditions are represented as ObservationValue tuples of a numeric reading and a string indicating the units that the reading is in (e.g., unit:degC for a temperature in degrees Celsius)

ObservationDatabase

Our local storage of conditions is handled via Room and an ObservationDatabase:

```kotlin
package com.commonsware.coroutines.weather

import android.content.Context
import androidx.annotation.NonNull
import androidx.room.*
import io.reactivex.Completable
import io.reactivex.Observable

private const val DB_NAME = "weather.db"

@Entity(tableName = "observations")
data class ObservationEntity(
  @PrimaryKey @NonNull val id: String,
  val timestamp: String,
  val icon: String,
  val temperatureCelsius: Double?,
  val windDirectionDegrees: Double?,
  val windSpeedMetersSecond: Double?,
  val barometricPressurePascals: Double?
) {
  fun toModel() = ObservationModel(
```

```kotlin
    id = id,
    timestamp = timestamp,
    icon = icon,
    temperatureCelsius = temperatureCelsius,
    windDirectionDegrees = windDirectionDegrees,
    windSpeedMetersSecond = windSpeedMetersSecond,
    barometricPressurePascals = barometricPressurePascals
  )
}

@Dao
interface ObservationStore {
  @Query("SELECT * FROM observations ORDER BY timestamp DESC")
  fun load(): Observable<List<ObservationEntity>>

  @Insert(onConflict = OnConflictStrategy.IGNORE)
  fun save(entity: ObservationEntity): Completable

  @Query("DELETE FROM observations")
  fun clear(): Completable
}

@Database(entities = [ObservationEntity::class], version = 1)
abstract class ObservationDatabase : RoomDatabase() {
  abstract fun observationStore(): ObservationStore

  companion object {
    fun create(context: Context) =
      Room.databaseBuilder(context, ObservationDatabase::class.java, DB_NAME)
        .build()
  }
}
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationDatabase.kt)

The JSON that we get back from the Web service is very hierarchical. The Room representation, by contrast, is a single observations table, modeled by an ObservationEntity. ObservationEntity holds the key pieces of data from the JSON response in individual fields, with the core conditions readings normalized to a single set of units (e.g., temperatureCelsius).

From an RxJava standpoint, we are using Room's RxJava extensions, and so our ObservationStore DAO has a load() function that returns an Observable for our select-all query, giving us a List of ObservationEntityobjects. We also have a save() function to insert a new reading (if we do not already have one for its ID) and a clear() function to delete all current rows from the table. Those are set to

return `Completable`, so we can find out when they are done with their work but are not expecting any particular data back.

In terms of threading, Room sets up the scheduler used for the code-generated functions implementing `ObservationStore`. While we might call `observeOn()` to control the scheduler used for delivering the results, we do not need to call `subscribeOn()`, as Room sets up that scheduler automatically. Specifically, it uses an RxJava `Scheduler` wrapped around a standard thread pool used by Room and many other pieces of the Jetpack family of libraries.

ObservationRepository

The gateway to `ObservationRemoteDataSource` and `ObservationDatabase` is the `ObservationRepository`. This provides the public API that our UI uses to request weather observations:

```kotlin
package com.commonsware.coroutines.weather

import io.reactivex.Completable
import io.reactivex.Observable
import io.reactivex.schedulers.Schedulers

interface IObservationRepository {
  fun load(): Observable<List<ObservationModel>>
  fun refresh(): Completable
  fun clear(): Completable
}

class ObservationRepository(
  private val db: ObservationDatabase,
  private val remote: ObservationRemoteDataSource
) : IObservationRepository {
  override fun load(): Observable<List<ObservationModel>> = db.observationStore()
    .load()
    .map { entities -> entities.map { it.toModel() } }
  }

  override fun refresh() = remote.getCurrentObservation()
      .subscribeOn(Schedulers.io())
      .map { convertToEntity(it) }
      .flatMapCompletable { db.observationStore().save(it) }

  override fun clear() = db.observationStore().clear()

  private fun convertToEntity(response: ObservationResponse): ObservationEntity {
    when {
      response.properties.temperature.unitCode != "unit:degC" ->
        throw IllegalStateException(
          "Unexpected temperature unit: ${response.properties.temperature.unitCode}"
        )
      response.properties.windDirection.unitCode != "unit:degree_(angle)" ->
        throw IllegalStateException(
```

```kotlin
          "Unexpected windDirection unit: ${response.properties.windDirection.unitCode}"
        )
      response.properties.windSpeed.unitCode != "unit:km_h-1" ->
        throw IllegalStateException(
          "Unexpected windSpeed unit: ${response.properties.windSpeed.unitCode}"
        )
      response.properties.barometricPressure.unitCode != "unit:Pa" ->
        throw IllegalStateException(
          "Unexpected barometricPressure unit: ${response.properties.barometricPressure.unitCode}"
        )
    }

    return ObservationEntity(
      id = response.id,
      icon = response.properties.icon,
      timestamp = response.properties.timestamp,
      temperatureCelsius = response.properties.temperature.value,
      windDirectionDegrees = response.properties.windDirection.value,
      windSpeedMetersSecond = response.properties.windSpeed.value,
      barometricPressurePascals = response.properties.barometricPressure.value
    )
  }
}

data class ObservationModel(
  val id: String,
  val timestamp: String,
  val icon: String,
  val temperatureCelsius: Double?,
  val windDirectionDegrees: Double?,
  val windSpeedMetersSecond: Double?,
  val barometricPressurePascals: Double?
)
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRepository.kt)

There are three public functions in that API, which is defined on an
IObservationRepository interface:

- load() calls load() on our DAO and use RxJava's map() operator to convert
  each of the ObservationEntity objects into a corresponding
  ObservationModel, so our UI layer is isolated from any changes in our
  database schema that future Room versions might require. load() then
  returns an Observable of our list of models.
- refresh() calls getCurrentObservation() on our
  ObservationRemoteDataSource, uses RxJava's map() to convert that to a
  model, then uses RxJava's flatMapCompletable() to wrap a call to save() on
  our DAO. The net result is that refresh() returns a Completable to indicate
  when the refresh operation is done. refresh() itself does not return fresh
  data — instead, we are relying upon Room to update the Observable passed
  through load() when a refresh results in a new row in our table.
- clear() is just a pass-through to the corresponding clear() function on our

DAO

KoinApp

All of this is knitted together by Koin as a service loader/dependency injector framework. Our Koin module — defined in a KoinApp subclass of Application — sets up the OkHttpClient instance, our ObservationDatabase, our ObservationRemoteDataSource, and our ObservationRepository, among other things:

```kotlin
package com.commonsware.coroutines.weather

import android.app.Application
import com.jakewharton.threetenabp.AndroidThreeTen
import okhttp3.OkHttpClient
import org.koin.android.ext.koin.androidContext
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.context.startKoin
import org.koin.dsl.module
import org.threeten.bp.format.DateTimeFormatter
import org.threeten.bp.format.FormatStyle

class KoinApp : Application() {
  private val koinModule = module {
    single { OkHttpClient.Builder().build() }
    single { ObservationDatabase.create(androidContext()) }
    single { ObservationRemoteDataSource(get()) }
    single<IObservationRepository> { ObservationRepository(get(), get()) }
    single { DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT) }
    viewModel { MainMotor(get(), get(), androidContext()) }
  }

  override fun onCreate() {
    super.onCreate()

    AndroidThreeTen.init(this);

    startKoin {
      androidLogger()
      androidContext(this@KoinApp)
      modules(koinModule)
    }
  }
}
```

MainMotor

The app uses a unidirectional data flow GUI architecture pattern. The UI (`MainActivity`) observes a stream of view-state objects (`MainViewState`) and calls functions that result in a fresh view-state being delivered to reflect any subsequent changes in the UI content.

`MainMotor` is what implements those functions, such as `refresh()` and `clear()`, and is what manages the stream of view-state objects:

```kotlin
package com.commonsware.coroutines.weather

import android.content.Context
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.disposables.CompositeDisposable
import io.reactivex.rxkotlin.subscribeBy
import org.threeten.bp.OffsetDateTime
import org.threeten.bp.ZoneId
import org.threeten.bp.format.DateTimeFormatter

data class RowState(
  val timestamp: String,
  val icon: String,
  val temp: String?,
  val wind: String?,
  val pressure: String?
) {
  companion object {
    fun fromModel(
      model: ObservationModel,
      formatter: DateTimeFormatter,
      context: Context
    ): RowState {
      val timestampDateTime = OffsetDateTime.parse(
        model.timestamp,
        DateTimeFormatter.ISO_OFFSET_DATE_TIME
      )
      val easternTimeId = ZoneId.of("America/New_York")
      val formattedTimestamp =
        formatter.format(timestampDateTime.atZoneSameInstant(easternTimeId))
```

```kotlin
      return RowState(
        timestamp = formattedTimestamp,
        icon = model.icon,
        temp = model.temperatureCelsius?.let {
          context.getString(
            R.string.temp,
            it
          )
        },
        wind = model.windSpeedMetersSecond?.let { speed ->
          model.windDirectionDegrees?.let {
            context.getString(
              R.string.wind,
              speed,
              it
            )
          }
        },
        pressure = model.barometricPressurePascals?.let {
          context.getString(
            R.string.pressure,
            it
          )
        }
      )
    }
  }
}

sealed class MainViewState {
  object Loading : MainViewState()
  data class Content(val observations: List<RowState>) : MainViewState()
  data class Error(val throwable: Throwable) : MainViewState()
}

class MainMotor(
  private val repo: IObservationRepository,
  private val formatter: DateTimeFormatter,
  private val context: Context
) : ViewModel() {
  private val _states =
    MutableLiveData<MainViewState>().apply { value = MainViewState.Loading }
  val states: LiveData<MainViewState> = _states
  private val sub = CompositeDisposable()

  init {
    sub.add(
      repo.load()
```

```kotlin
        .observeOn(AndroidSchedulers.mainThread())
        .subscribeBy(onNext = { models ->
          _states.value = MainViewState.Content(models.map {
            RowState.fromModel(it, formatter, context)
          })
        }, onError = { _states.value = MainViewState.Error(it) })
    )
  }

  override fun onCleared() {
    sub.dispose()
  }

  fun refresh() {
    sub.add(repo.refresh()
      .observeOn(AndroidSchedulers.mainThread())
      .subscribeBy(onError = { _states.value = MainViewState.Error(it) })
    )
  }

  fun clear() {
    sub.add(repo.clear()
      .observeOn(AndroidSchedulers.mainThread())
      .subscribeBy(onError = { _states.value = MainViewState.Error(it) })
    )
  }
}
```

(from app/src/main/java/com/commonsware/coroutines/weather/MainMotor.kt)
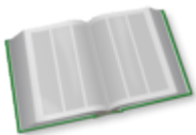
> You can learn more about the unidirectional data flow GUI architecture pattern in the "Adding Some Architecture" chapter of *Elements of Android Jetpack*!

MainMotor itself is a `ViewModel` from the Jetpack viewmodel system. Our Koin module supports injecting viewmodels, so it is set up to provide an instance of `MainMotor` to `MainActivity` when needed.

When our motor is instantiated, we immediately call `load()` on our repository. We then go through a typical RxJava construction to:

- Arrange to consume the results of that I/O on the main application thread (`observeOn(AndroidSchedulers.mainThread())`)
- Supply lambda expressions for consuming both the success and failure cases

- Add the resulting `Disposable` to a `CompositeDisposable`, which we clear when the viewmodel itself is cleared in `onCleared()`

The net effect is that whenever we get fresh models from the repository-supplied `Observable`, we update a `MutableLiveData` with a `MainViewState` wrapping either the list of models (`MainViewState.Content`) or the exception that we got (`MainViewState.Error`).

The motor's `refresh()` and `clear()` functions call `refresh()` and `clear()` on the repository, with the same sort of RxJava setup to handle threading and any errors. In the case of `refresh()` and `clear()`, though, since we get RxJava `Completable` objects from the repository, there is nothing to do for the success cases — we just wait for Room to deliver fresh models through the `load()` `Observable`.

Note that `MainViewState` actually has three states: `Loading`, `Content`, and `Error`. `Loading` is our initial state, set up when we initialize the `MutableLiveData`. The other states are used by the main code of the motor. The "content" for the `Content` state consists of a list of `RowState` objects, where each `RowState` is a formatted rendition of the key bits of data from an observation (e.g., formatting the date after converting it to Washington DC's time zone). Note that the `DateTimeFormatter` comes from Koin via the constructor, so we can use a formatter for tests that is locale-independent.

MainActivity

Our UI consists of a `RecyclerView` and `ProgressBar` inside of a `ConstraintLayout`:

```xml
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".MainActivity">

  <androidx.recyclerview.widget.RecyclerView
    android:id="@+id/observations"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_margin="8dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
```

```xml
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent">

  </androidx.recyclerview.widget.RecyclerView>

  <ProgressBar
    android:id="@+id/progress"
    style="?android:attr/progressBarStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

(from app/src/main/res/layout/activity_main.xml)

Each row of the `RecyclerView` will show the relevant bits of data from our observations, including an `ImageView` for the icon:

```xml
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:app="http://schemas.android.com/apk/res-auto"
  xmlns:tools="http://schemas.android.com/tools">

  <data>

    <variable
      name="state"
      type="com.commonsware.coroutines.weather.RowState" />
  </data>

  <androidx.constraintlayout.widget.ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:paddingBottom="8dp"
    android:paddingTop="8dp">

    <ImageView
      android:id="@+id/icon"
      android:layout_width="0dp"
      android:layout_height="64dp"
```

```xml
    android:contentDescription="@string/icon"
    app:imageUrl="@{state.icon}"
    app:layout_constraintDimensionRatio="1:1"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:srcCompat="@tools:sample/backgrounds/scenic" />

<TextView
    android:id="@+id/timestamp"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:gravity="center"
    android:text="@{state.timestamp}"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    app:autoSizeMaxTextSize="16sp"
    app:autoSizeTextType="uniform"
    app:layout_constraintBottom_toBottomOf="@id/icon"
    app:layout_constraintEnd_toStartOf="@id/barrier"
    app:layout_constraintStart_toEndOf="@id/icon"
    app:layout_constraintTop_toTopOf="@id/icon"
    tools:text="7/5/19 13:52" />

<TextView
    android:id="@+id/temp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{state.temp}"
    android:textAppearance="@style/TextAppearance.AppCompat.Small"
    app:layout_constraintBottom_toTopOf="@id/wind"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="@id/icon"
    tools:text="20 C" />

<TextView
    android:id="@+id/wind"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{state.wind}"
    android:textAppearance="@style/TextAppearance.AppCompat.Small"
    app:layout_constraintBottom_toTopOf="@id/pressure"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toBottomOf="@id/temp"
    tools:text="5.1 m/s @ 170°" />

<TextView
    android:id="@+id/pressure"
```

```xml
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@{state.pressure}"
        android:textAppearance="@style/TextAppearance.AppCompat.Small"
        app:layout_constraintBottom_toBottomOf="@id/icon"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toBottomOf="@id/wind"
        tools:text="10304 Pa" />

    <androidx.constraintlayout.widget.Barrier
        android:id="@+id/barrier"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:barrierDirection="start"
        app:constraint_referenced_ids="temp,wind,pressure" />

  </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

<div align="right">(from <u>app/src/main/res/layout/row.xml</u>)</div>

The row layout uses data binding, with binding expressions to update views based on `RowState` properties. For the `ImageView`, this requires a binding adapter, in this case using Glide:

```kotlin
package com.commonsware.coroutines.weather

import android.widget.ImageView
import androidx.databinding.BindingAdapter
import com.bumptech.glide.Glide

@BindingAdapter("imageUrl")
fun ImageView.loadImage(url: String?) {
  url?.let {
    Glide.with(context)
      .load(it)
      .into(this)
  }
}
```

<div align="right">(from <u>app/src/main/java/com/commonsware/coroutines/weather/BindingAdapters.kt</u>)</div>

The `MainActivity` Kotlin file not only contains the activity code, but also the adapter and view-holder for our `RecyclerView`:

```kotlin
package com.commonsware.coroutines.weather

import android.os.Bundle
import android.util.Log
import android.view.Menu
import android.view.MenuItem
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.lifecycle.observe
import androidx.recyclerview.widget.*
import com.commonsware.coroutines.weather.databinding.ActivityMainBinding
import com.commonsware.coroutines.weather.databinding.RowBinding
import org.koin.androidx.viewmodel.ext.android.viewModel

class MainActivity : AppCompatActivity() {
  private val motor: MainMotor by viewModel()

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val binding = ActivityMainBinding.inflate(layoutInflater)

    setContentView(binding.root)

    val adapter = ObservationAdapter()

    binding.observations.layoutManager = LinearLayoutManager(this)
    binding.observations.adapter = adapter
    binding.observations.addItemDecoration(
      DividerItemDecoration(
        this,
        DividerItemDecoration.VERTICAL
      )
    )

    motor.states.observe(this) { state ->
      when (state) {
        MainViewState.Loading -> binding.progress.visibility = View.VISIBLE
        is MainViewState.Content -> {
          binding.progress.visibility = View.GONE
          adapter.submitList(state.observations)
        }
        is MainViewState.Error -> {
          binding.progress.visibility = View.GONE
          Toast.makeText(
            this@MainActivity, state.throwable.localizedMessage,
```

202

```kotlin
                    Toast.LENGTH_LONG
          ).show()
          Log.e("Weather", "Exception loading data", state.throwable)
        }
      }
  }

  motor.refresh()
}

override fun onCreateOptionsMenu(menu: Menu): Boolean {
  menuInflater.inflate(R.menu.actions, menu)

  return super.onCreateOptionsMenu(menu)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
  when (item.itemId) {
    R.id.refresh -> { motor.refresh(); return true }
    R.id.clear -> { motor.clear(); return true }
  }

  return super.onOptionsItemSelected(item)
}

inner class ObservationAdapter :
  ListAdapter<RowState, RowHolder>(RowStateDiffer) {
  override fun onCreateViewHolder(
    parent: ViewGroup,
    viewType: Int
  ) = RowHolder(RowBinding.inflate(layoutInflater, parent, false))

  override fun onBindViewHolder(holder: RowHolder, position: Int) {
    holder.bind(getItem(position))
  }
}

class RowHolder(private val binding: RowBinding) :
  RecyclerView.ViewHolder(binding.root) {

  fun bind(state: RowState) {
    binding.state = state
    binding.executePendingBindings()
  }
}

object RowStateDiffer : DiffUtil.ItemCallback<RowState>() {
  override fun areItemsTheSame(
```

```
    oldItem: RowState,
    newItem: RowState
  ): Boolean {
    return oldItem === newItem
  }

  override fun areContentsTheSame(
    oldItem: RowState,
    newItem: RowState
  ): Boolean {
    return oldItem == newItem
  }
  }
}
```

(from app/src/main/java/com/commonsware/coroutines/weather/MainActivity.kt)

The activity gets its motor via Koin (`private val motor: MainMotor by inject()`). The activity observes the `LiveData` and uses that to update the states of the `RecyclerView` and the `ProgressBar`. The `RecyclerView.Adapter` (`ObservationAdapter`) uses `ListAdapter`, so we can just call `submitList()` to supply the new list of `RowState` objects. If we get an `Error` view-state, we log the exception and show a `Toast` with the message.

The activity also sets up the action bar, with refresh and clear options to call `refresh()` and `clear()` on the motor, respectively. We also call `refresh()` at the end of `onCreate()`, to get the initial observation for a fresh install and to make sure that we have the latest observation for any future launches of the activity.

Step #2: Deciding What to Change (and How)

The `Single` response from `ObservationRemoteDataSource`, and the `Completable` responses from `ObservationStore`, are easy to convert into `suspend` functions returning a value (for `Single`) or `Unit` (for `Completable`). We can use `Flow` to replace our `Observable`.

Once we have made these changes — both to the main app code and to the tests — we can remove RxJava from the project.

Step #3: Adding a Coroutines Dependency

Coroutines are not automatically added to a Kotlin project — you need the

org.jetbrains.kotlinx:kotlinx-coroutines-android dependency for Android, which in turn will pull in the rest of the coroutines implementation.

In our case, though, we also need to add a dependency to teach Room about coroutines, so it can generate suspend functions for relevant DAO functions. That, in turn, will add general coroutine support to our project via transitive dependencies.

To that end, add this line to the app module's build.gradle file's dependencies closure:

```
implementation "androidx.room:room-runtime:$room_version"
```

This uses a room_version constant that we have set up to synchronize the versions of our various Room dependencies:

```
def retrofit_version = "2.9.0"
```

<div align="right">(from <u>app/build.gradle</u>)</div>

Step #4: Converting ObservationRemoteDataSource

To have Retrofit use a suspend function, simply change the function signature of getCurrentObservation() in ObservationApi from:

```
fun getCurrentObservation(@Path("stationId") stationId: String): Single<ObservationResponse>
```

<div align="right">(from <u>app/src/main/java/com/commonsware/coroutines/weather/ObservationRemoteDataSource.kt</u>)</div>

to:

```
suspend fun getCurrentObservation(@Path("stationId") stationId: String): ObservationResponse
```

So now getCurrentObservation() directly returns our ObservationResponse, with the suspend allowing Retrofit to have the network I/O occur on another dispatcher.

This, in turn, will require us to add suspend to getCurrentObservation() on ObservationRemoteDataSource:

```
suspend fun getCurrentObservation() = api.getCurrentObservation(STATION)
```

Since we are no longer using RxJava with Retrofit, we can remove its call adapter factory from our Retrofit.Builder configuration. Delete the addCallAdapterFactory() line seen in the original code:

```
 private val retrofit = Retrofit.Builder()
    .client(ok)
    .baseUrl(API_ENDPOINT)
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .addConverterFactory(MoshiConverterFactory.create())
    .build()
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRemoteDataSource.kt)

...leaving you with:

```
 private val retrofit = Retrofit.Builder()
    .client(ok)
    .baseUrl(API_ENDPOINT)
    .addConverterFactory(MoshiConverterFactory.create())
    .build()
```

Also, you can remove any RxJava imports (e.g., for Single).

At this point, ObservationRemoteDataSource should look like:

```
package com.commonsware.coroutines.weather

import okhttp3.OkHttpClient
import retrofit2.Retrofit
import retrofit2.converter.moshi.MoshiConverterFactory
import retrofit2.http.GET
import retrofit2.http.Path

private const val STATION = "KDCA"
private const val API_ENDPOINT = "https://api.weather.gov"

class ObservationRemoteDataSource(ok: OkHttpClient) {
  private val retrofit = Retrofit.Builder()
    .client(ok)
    .baseUrl(API_ENDPOINT)
    .addConverterFactory(MoshiConverterFactory.create())
    .build()
  private val api = retrofit.create(ObservationApi::class.java)

  suspend fun getCurrentObservation() = api.getCurrentObservation(STATION)
}

private interface ObservationApi {
  @GET("/stations/{stationId}/observations/current")
  suspend fun getCurrentObservation(@Path("stationId") stationId: String): ObservationResponse
}

data class ObservationResponse(
  val id: String,
  val properties: ObservationProperties
)

data class ObservationProperties(
```

```kotlin
  val timestamp: String,
  val icon: String,
  val temperature: ObservationValue,
  val windDirection: ObservationValue,
  val windSpeed: ObservationValue,
  val barometricPressure: ObservationValue
)

data class ObservationValue(
  val value: Double?,
  val unitCode: String
)
```

Step #5: Altering `ObservationDatabase`

Changing our database writes is a matter of updating the function declarations on `ObservationStore`, replacing a `Completable` return type with suspend and switching the `load()` function to return a `Flow`.

So, change:

```kotlin
@Dao
interface ObservationStore {
  @Query("SELECT * FROM observations ORDER BY timestamp DESC")
  fun load(): Observable<List<ObservationEntity>>

  @Insert(onConflict = OnConflictStrategy.IGNORE)
  fun save(entity: ObservationEntity): Completable

  @Query("DELETE FROM observations")
  fun clear(): Completable
}
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationDatabase.kt)

to:

```kotlin
@Dao
interface ObservationStore {
  @Query("SELECT * FROM observations ORDER BY timestamp DESC")
  fun load(): Flow<List<ObservationEntity>>

  @Insert(onConflict = OnConflictStrategy.IGNORE)
  suspend fun save(entity: ObservationEntity)

  @Query("DELETE FROM observations")
  suspend fun clear()
}
```

You can also get rid of the RxJava-related `import` statements (e.g., `Observable`, `Completable`).

At this point, the overall `ObservationDatabase.kt` file should resemble:

```kotlin
package com.commonsware.coroutines.weather

import android.content.Context
import androidx.annotation.NonNull
import androidx.room.*
import kotlinx.coroutines.flow.Flow

private const val DB_NAME = "weather.db"

@Entity(tableName = "observations")
data class ObservationEntity(
  @PrimaryKey @NonNull val id: String,
  val timestamp: String,
  val icon: String,
  val temperatureCelsius: Double?,
  val windDirectionDegrees: Double?,
  val windSpeedMetersSecond: Double?,
  val barometricPressurePascals: Double?
) {
  fun toModel() = ObservationModel(
    id = id,
    timestamp = timestamp,
    icon = icon,
    temperatureCelsius = temperatureCelsius,
    windDirectionDegrees = windDirectionDegrees,
    windSpeedMetersSecond = windSpeedMetersSecond,
    barometricPressurePascals = barometricPressurePascals
  )
}

@Dao
interface ObservationStore {
  @Query("SELECT * FROM observations ORDER BY timestamp DESC")
  fun load(): Flow<List<ObservationEntity>>

  @Insert(onConflict = OnConflictStrategy.IGNORE)
  suspend fun save(entity: ObservationEntity)

  @Query("DELETE FROM observations")
  suspend fun clear()
}
```

```
@Database(entities = [ObservationEntity::class], version = 1)
abstract class ObservationDatabase : RoomDatabase() {
  abstract fun observationStore(): ObservationStore

  companion object {
    fun create(context: Context) =
      Room.databaseBuilder(context, ObservationDatabase::class.java, DB_NAME)
        .build()
  }
}
```

In terms of threading, just as Room handles our RxJava scheduler, Room sets up the coroutines dispatcher used for the code-generated functions implementing `ObservationStore`. Specifically, it uses a dispatcher wrapped around the same thread pool that it would have used for its RxJava `Scheduler`.

Step #6: Adjusting `ObservationRepository`

Our `ObservationRepository` now needs to be updated to use the new function signatures from the data sources.

First, add `suspend` to `refresh()` and `clear()` — and remove the `Completable` return type — to the `IObservationRepository` interface that `ObservationRepository` implements. Also, switch `load()` to return a `Flow`:

```
interface IObservationRepository {
  fun load(): Flow<List<ObservationModel>>
  suspend fun refresh()
  suspend fun clear()
}
```

Now, we need to adjust the implementation of `ObservationRepository` itself.

First, replace the current `load()` function with:

```
  override fun load(): Flow<List<ObservationModel>> =
    db.observationStore()
      .load()
      .map { entities ->
        entities.map { it.toModel() }
      }
```

Note that you will need to add an `import` for the `map()` extension function on `Flow`, since even though syntactically it is the same as what we had before, the `map()`

implementation is now an extension function:

```
import kotlinx.coroutines.flow.map
```

Our revisions to the load() function just:

- Switches the return type from Observable to Flow
- Uses the Flow.map() extension function instead of the map() method on Observable

That one was easy, as it is a read operation. For write operations — refresh() and clear() — we need to make certain that they actually complete and do not get canceled just because, say, the user rotates the screen.

The approach we will use here involves creating a custom application-level CoroutineScope with a SupervisorJob. The SupervisorJob will prevent an exception in one coroutine from canceling any other coroutines in that scope. And, by simply not clearing the CoroutineScope, we avoid any premature unexpected cancellation of our coroutines.

To that end, add another single line to the koinModule declaration in KoinApp:

```
single(named("appScope")) { CoroutineScope(SupervisorJob()) }
```

This declares a new singleton instance of a CoroutineScope with a SupervisorJob that we can inject into other declarations. We use the named("appScope") Koin qualifier to identify this scope, just in case sometime later we want to have other injectable CoroutineScope objects.

Then, modify ObservationRepository to take a CoroutineScope as a constructor parameter:

```
class ObservationRepository(
  private val db: ObservationDatabase,
  private val remote: ObservationRemoteDataSource,
  private val appScope: CoroutineScope
) : IObservationRepository {
```

Back in KoinApp, our ObservationRepository single now needs that parameter, so adjust its declaration to be:

```
    single<IObservationRepository> {
      ObservationRepository(
        get(),
        get(),
        get(named("appScope"))
      )
    }
```

We use get(named("appScope")) to retrieve the appScope object that we declared earlier in the module.

At this point, KoinApp should look like:

```kotlin
package com.commonsware.coroutines.weather

import android.app.Application
import com.jakewharton.threetenabp.AndroidThreeTen
import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.SupervisorJob
import okhttp3.OkHttpClient
import org.koin.android.ext.koin.androidContext
import org.koin.android.ext.koin.androidLogger
import org.koin.androidx.viewmodel.dsl.viewModel
import org.koin.core.context.startKoin
import org.koin.core.qualifier.named
import org.koin.dsl.module
import org.threeten.bp.format.DateTimeFormatter
import org.threeten.bp.format.FormatStyle

class KoinApp : Application() {
  private val koinModule = module {
    single(named("appScope")) { CoroutineScope(SupervisorJob()) }
    single { OkHttpClient.Builder().build() }
    single { ObservationDatabase.create(androidContext()) }
    single { ObservationRemoteDataSource(get()) }
    single<IObservationRepository> {
      ObservationRepository(
        get(),
        get(),
        get(named("appScope"))
      )
    }
    single { DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT) }
    viewModel { MainMotor(get(), get(), androidContext()) }
  }

  override fun onCreate() {
```

```
    super.onCreate()

    AndroidThreeTen.init(this);

    startKoin {
      androidLogger()
      androidContext(this@KoinApp)
      modules(koinModule)
    }
  }
}
```

Now, we can actually rewrite `refresh()` and `clear()` on `ObservationRepository`.

Fixing `clear()` is relatively easy — just take this:

```
  override fun clear() = db.observationStore().clear()
```

<div align="right">(from <u>app/src/main/java/com/commonsware/coroutines/weather/ObservationRepository.kt</u>)</div>

...and change it to this:

```
  override suspend fun clear() = withContext(appScope.coroutineContext) {
    db.observationStore().clear()
  }
```

`db.observationStore().clear()` is our DAO operation to clear the table. We wrap that in `withContext(appScope.coroutineContext)`, to have our `appScope` manage this coroutine.

`refresh()` is a bit more involved. The current code is:

```
  override fun refresh() = remote.getCurrentObservation()
      .subscribeOn(Schedulers.io())
      .map { convertToEntity(it) }
      .flatMapCompletable { db.observationStore().save(it) }
```

<div align="right">(from <u>app/src/main/java/com/commonsware/coroutines/weather/ObservationRepository.kt</u>)</div>

That takes our `Single` response from the repository and applies two RxJava operators to it:

- `map()` to convert the `ObservationResponse` into an `ObservationEntity`, and
- `flatMapCompletable()` to `save()` the entity in Room as part of this RxJava chain

Change that to:

```kotlin
override suspend fun refresh() = withContext(appScope.coroutineContext) {
  db.observationStore()
    .save(convertToEntity(remote.getCurrentObservation()))
}
```

Now, we have a more natural imperative-style syntax, getting our response, converting it to an entity, and passing the entity to save(). The suspend keyword is required, since we are calling suspend functions, as we needed with refresh(). And, we use the same withContext(appScope.coroutineContext) wrapper as we did with clear().

You can remove any RxJava-related import statements (e.g., Observable, Single).

At this point, ObservationRepository should look like:

```kotlin
package com.commonsware.coroutines.weather

import kotlinx.coroutines.CoroutineScope
import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.withContext

interface IObservationRepository {
  fun load(): Flow<List<ObservationModel>>
  suspend fun refresh()
  suspend fun clear()
}

class ObservationRepository(
  private val db: ObservationDatabase,
  private val remote: ObservationRemoteDataSource,
  private val appScope: CoroutineScope
) : IObservationRepository {
  override fun load(): Flow<List<ObservationModel>> =
    db.observationStore()
      .load()
      .map { entities ->
        entities.map { it.toModel() }
      }

  override suspend fun refresh() = withContext(appScope.coroutineContext) {
    db.observationStore()
      .save(convertToEntity(remote.getCurrentObservation()))
  }

  override suspend fun clear() = withContext(appScope.coroutineContext) {
    db.observationStore().clear()
  }

  private fun convertToEntity(response: ObservationResponse): ObservationEntity {
    when {
```

```kotlin
    response.properties.temperature.unitCode != "unit:degC" ->
      throw IllegalStateException(
        "Unexpected temperature unit: ${response.properties.temperature.unitCode}"
      )
    response.properties.windDirection.unitCode != "unit:degree_(angle)" ->
      throw IllegalStateException(
        "Unexpected windDirection unit: ${response.properties.windDirection.unitCode}"
      )
    response.properties.windSpeed.unitCode != "unit:km_h-1" ->
      throw IllegalStateException(
        "Unexpected windSpeed unit: ${response.properties.windSpeed.unitCode}"
      )
    response.properties.barometricPressure.unitCode != "unit:Pa" ->
      throw IllegalStateException(
        "Unexpected barometricPressure unit: ${response.properties.barometricPressure.unitCode}"
      )
  }

  return ObservationEntity(
    id = response.id,
    icon = response.properties.icon,
    timestamp = response.properties.timestamp,
    temperatureCelsius = response.properties.temperature.value,
    windDirectionDegrees = response.properties.windDirection.value,
    windSpeedMetersSecond = response.properties.windSpeed.value,
    barometricPressurePascals = response.properties.barometricPressure.value
  )
 }
}

data class ObservationModel(
  val id: String,
  val timestamp: String,
  val icon: String,
  val temperatureCelsius: Double?,
  val windDirectionDegrees: Double?,
  val windSpeedMetersSecond: Double?,
  val barometricPressurePascals: Double?
)
```

Step #7: Modifying `MainMotor`

The next step is to update `MainMotor` to call the new functions from the repository.

However, to do that, we need a `CoroutineScope`. The ideal scope is `viewModelScope`, which is tied to the life of the `ViewModel`. However, for that, we need another dependency. Add this line to the `dependencies` closure of `app/build.gradle`:

```
implementation 'androidx.lifecycle:lifecycle-livedata-ktx:2.2.0'
```

Then, replace the existing `refresh()` and `clear()` functions with:

```kotlin
fun refresh() {
  viewModelScope.launch(Dispatchers.Main) {
    try {
      repo.refresh()
    } catch (t: Throwable) {
      _states.value = MainViewState.Error(t)
    }
  }
}

fun clear() {
  viewModelScope.launch(Dispatchers.Main) {
    try {
      repo.clear()
    } catch (t: Throwable) {
      _states.value = MainViewState.Error(t)
    }
  }
}
```

Exceptions get thrown normally with suspend functions, so we can just use ordinary try/catch structures to catch them and publish an error state for our UI.

Finally, MainMotor now needs to switch from observing an Observable to collecting a Flow for getting our forecast.

Replace the init block in MainMotor with this:

```kotlin
init {
  viewModelScope.launch(Dispatchers.Main) {
    try {
      repo.load()
        .map { models ->
          MainViewState.Content(models.map {
            RowState.fromModel(it, formatter, context)
          })
        }
        .collect { _states.value = it }
    } catch (ex: Exception) {
      _states.value = MainViewState.Error(ex)
    }
  }
}
```

We launch() a coroutine that uses load() to get the list of ObservationModel objects. We then map() those into our view-state, then use collect() to "subscribe"

to the `Flow`, pouring the states into our `MutableLiveData`. If something goes wrong and Room throws an exception, that just gets caught by our `try/catch` construct, so we can update the `MutableLiveData` with an error state.

You can also remove the `sub` property and the `onCleared()` function, as we no longer are using the `CompositeDisposable`. Plus, you can remove all RxJava-related `import` statements, such as the one for `CompositeDisposable` itself.

At this point, `MainMotor` should look like:

```
package com.commonsware.coroutines.weather

import android.content.Context
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.launch
import org.threeten.bp.OffsetDateTime
import org.threeten.bp.ZoneId
import org.threeten.bp.format.DateTimeFormatter

data class RowState(
  val timestamp: String,
  val icon: String,
  val temp: String?,
  val wind: String?,
  val pressure: String?
) {
  companion object {
    fun fromModel(
      model: ObservationModel,
      formatter: DateTimeFormatter,
      context: Context
    ): RowState {
      val timestampDateTime = OffsetDateTime.parse(
        model.timestamp,
        DateTimeFormatter.ISO_OFFSET_DATE_TIME
      )
      val easternTimeId = ZoneId.of("America/New_York")
      val formattedTimestamp =
        formatter.format(timestampDateTime.atZoneSameInstant(easternTimeId))
```

```kotlin
      return RowState(
        timestamp = formattedTimestamp,
        icon = model.icon,
        temp = model.temperatureCelsius?.let {
          context.getString(
            R.string.temp,
            it
          )
        },
        wind = model.windSpeedMetersSecond?.let { speed ->
          model.windDirectionDegrees?.let {
            context.getString(
              R.string.wind,
              speed,
              it
            )
          }
        },
        pressure = model.barometricPressurePascals?.let {
          context.getString(
            R.string.pressure,
            it
          )
        }
      )
    }
  }
}

sealed class MainViewState {
  object Loading : MainViewState()
  data class Content(val observations: List<RowState>) : MainViewState()
  data class Error(val throwable: Throwable) : MainViewState()
}

class MainMotor(
  private val repo: IObservationRepository,
  private val formatter: DateTimeFormatter,
  private val context: Context
) : ViewModel() {
  private val _states =
    MutableLiveData<MainViewState>().apply { value = MainViewState.Loading }
  val states: LiveData<MainViewState> = _states

  init {
    viewModelScope.launch(Dispatchers.Main) {
      try {
        repo.load()
```

```kotlin
        .map { models ->
          MainViewState.Content(models.map {
            RowState.fromModel(it, formatter, context)
          })
        }
        .collect { _states.value = it }
    } catch (ex: Exception) {
      _states.value = MainViewState.Error(ex)
    }
  }
}

fun refresh() {
  viewModelScope.launch(Dispatchers.Main) {
    try {
      repo.refresh()
    } catch (t: Throwable) {
      _states.value = MainViewState.Error(t)
    }
  }
}

fun clear() {
  viewModelScope.launch(Dispatchers.Main) {
    try {
      repo.clear()
    } catch (t: Throwable) {
      _states.value = MainViewState.Error(t)
    }
  }
}
}
```

And, at this point, if you run the app, it should work as it did before, just without RxJava.

Step #8: Reviewing the Instrumented Tests

So far, we have avoided thinking about tests. This tutorial practices TDD (Test Delayed Development), but, it is time to get our tests working again.

First, let's take a look at our one instrumented test: MainMotorTest. As the name suggests, this test exercises MainMotor, to confirm that it can take ObservationModel objects and emit RowState objects as a result.

`MainMotorTest` makes use of two JUnit4 rules:

- `InstantTaskExecutorRule` from the Android Jetpack, for having all normally-asynchronous work related to `LiveData` occur on the current thread
- A custom `AndroidSchedulerRule` that applies an RxJava `TestScheduler` to replace the stock `Scheduler` used for `AndroidSchedulers.mainThread()`:

```
package com.commonsware.coroutines.weather

import io.reactivex.Scheduler
import io.reactivex.android.plugins.RxAndroidPlugins
import org.junit.rules.TestWatcher
import org.junit.runner.Description

class AndroidSchedulerRule<T : Scheduler>(val scheduler: T) : TestWatcher() {
  override fun starting(description: Description?) {
    super.starting(description)

    RxAndroidPlugins.setMainThreadSchedulerHandler { scheduler }
  }

  override fun finished(description: Description?) {
    super.finished(description)

    RxAndroidPlugins.reset()
  }
}
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/AndroidSchedulerRule.kt)

`MainMotorTest` also uses Mockito and Mockito-Kotlin to create a mock implementation of our repository, by mocking the `IObservationRepository` interface that defines the `ObservationRepository` API:

```
@RunWith(AndroidJUnit4::class)
class MainMotorTest {
  @get:Rule
  val instantTaskExecutorRule = InstantTaskExecutorRule()
  @get:Rule
  val androidSchedulerRule = AndroidSchedulerRule(TestScheduler())

  private val repo: IObservationRepository = mock()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

The `initialLoad()` test function confirms that our `MainMotor` handles a normal startup correctly. Specifically, we validate that our initial state is `MainViewState.Loading`, then proceeds to `MainViewState.Content`:

```kotlin
@Test
fun initialLoad() {
  whenever(repo.load()).thenReturn(Observable.just(listOf(TEST_MODEL)))

  val underTest = makeTestMotor()
  val initialState = underTest.states.value

  assertThat(initialState is MainViewState.Loading, equalTo(true))
  androidSchedulerRule.scheduler.triggerActions()

  val state = underTest.states.value as MainViewState.Content

  assertThat(state.observations.size, equalTo(1))
  assertThat(state.observations[0], equalTo(TEST_ROW_STATE))
}
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

Of particular note:

- We use `Observable.just()` to mock the response from `load()` on the repository
- We use the `TestScheduler` that we set up using `AndroidSchedulerRule` to advance from the loading state to the content state

The `initialLoadError()` test function demonstrates tests if `load()` throws some sort of exception, to confirm that we wind up with a `MainViewState.Error` result:

```kotlin
@Test
fun initialLoadError() {
  whenever(repo.load()).thenReturn(Observable.error(TEST_ERROR))

  val underTest = makeTestMotor()
  val initialState = underTest.states.value

  assertThat(initialState is MainViewState.Loading, equalTo(true))
  androidSchedulerRule.scheduler.triggerActions()

  val state = underTest.states.value as MainViewState.Error

  assertThat(TEST_ERROR, equalTo(state.throwable))
}
```

In this case, we use `Observable.error()` to set up an `Observable` that immediately throws a test error.

The `refresh()` test function confirms that we can call `refresh()` without crashing and that `MainMotor` can handle a fresh set of model objects delivered to it by our `Observable`. We simulate our ongoing Room-backed `Observable` by a `PublishSubject`, so we can provide initial data and can later supply "fresh" data, simulating the results of Room detecting our refreshed database content:

```kotlin
@Test
fun refresh() {
  val testSubject: PublishSubject<List<ObservationModel>> =
    PublishSubject.create()

  whenever(repo.load()).thenReturn(testSubject)
  whenever(repo.refresh()).thenReturn(Completable.complete())

  val underTest = makeTestMotor()
  val initialState = underTest.states.value

  assertThat(initialState is MainViewState.Loading, equalTo(true))

  testSubject.onNext(listOf(TEST_MODEL))
  androidSchedulerRule.scheduler.triggerActions()

  underTest.refresh()

  testSubject.onNext(listOf(TEST_MODEL, TEST_MODEL_2))
  androidSchedulerRule.scheduler.triggerActions()

  val state = underTest.states.value as MainViewState.Content

  assertThat(state.observations.size, equalTo(2))
  assertThat(state.observations[0], equalTo(TEST_ROW_STATE))
  assertThat(state.observations[1], equalTo(TEST_ROW_STATE_2))
}
```

Note that we use `Completable.complete()` as the mocked return value from `refresh()` on our repository, so `MainMotor` gets a valid `Completable` for use.

Finally, the `clear()` test function confirms that we can call `clear()` without crashing and that `MainMotor` can handle an empty set of model objects delivered to

it by our `Observable`:

```kotlin
@Test
fun clear() {
  val testSubject: PublishSubject<List<ObservationModel>> =
    PublishSubject.create()

  whenever(repo.load()).thenReturn(testSubject)
  whenever(repo.clear()).thenReturn(Completable.complete())

  val underTest = makeTestMotor()
  val initialState = underTest.states.value

  assertThat(initialState is MainViewState.Loading, equalTo(true))

  testSubject.onNext(listOf(TEST_MODEL))
  androidSchedulerRule.scheduler.triggerActions()

  underTest.clear()

  testSubject.onNext(listOf())
  androidSchedulerRule.scheduler.triggerActions()

  val state = underTest.states.value as MainViewState.Content

  assertThat(state.observations.size, equalTo(0))
}
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

This is not a complete set of tests for `MainMotor`, but it is enough to give us the spirit of testing the RxJava/Jetpack combination.

Step #9: Repair `MainMotorTest`

Of course, `MainMotorTest` is littered with compile errors at the moment, as we changed `MainMotor` and `ObservationRepository` to use coroutines instead of RxJava. So, we need to make some repairs.

First, add this dependency to `app/build.gradle`:

```
androidTestImplementation 'com.natpryce:hamkrest:1.7.0.0'
```

This is a release candidate of the `kotlinx-coroutines-test` dependency. It gives us a `TestCoroutineDispatcher` that we can use in a similar fashion as to how we used

TestScheduler with RxJava.

Next, add this `MainDispatcherRule` alongside `MainMotorTest` in the `androidTest` source set:

```kotlin
package com.commonsware.coroutines.weather

import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.ExperimentalCoroutinesApi
import kotlinx.coroutines.test.TestCoroutineDispatcher
import kotlinx.coroutines.test.resetMain
import kotlinx.coroutines.test.setMain
import org.junit.rules.TestWatcher
import org.junit.runner.Description

// inspired by https://medium.com/androiddevelopers/easy-coroutines-in-android-viewmodelscope-25bffb605471

@ExperimentalCoroutinesApi
class MainDispatcherRule(paused: Boolean) : TestWatcher() {
  val dispatcher =
    TestCoroutineDispatcher().apply { if (paused) pauseDispatcher() }

  override fun starting(description: Description?) {
    super.starting(description)

    Dispatchers.setMain(dispatcher)
  }

  override fun finished(description: Description?) {
    super.finished(description)

    Dispatchers.resetMain()
    dispatcher.cleanupTestCoroutines()
  }
}
```

This is another JUnit rule, implemented as a `TestWatcher`, allowing us to encapuslate some setup and teardown work. Specifically, we create a `TestCoroutineDispatcher` and possibly configure it to be paused, so we have to manually trigger coroutines to be executed. Then, when a test using the rule starts, we use `Dispatchers.setMain()` to override the default `Dispatchers.Main` dispatcher. When a test using the rule ends, we use `Dispatchers.resetMain()` to return to normal operation, plus clean up our `TestCoroutineDispatcher` for any lingering coroutines that are still outstanding.

Then, in `MainMotorTest`, replace these `AndroidSchedulerRule` lines:

```kotlin
@get:Rule
val androidSchedulerRule = AndroidSchedulerRule(TestScheduler())
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

...with these lines to set up the `MainDispatcherRule`:

```
@get:Rule
val mainDispatcherRule = MainDispatcherRule(paused = true)
```

At this point, we are no longer using `AndroidSchedulerRule`, so you can delete that class.

However, our use of `MainDispatcherRule` has a warning, indicating that this is an experimental API. So, add `@ExperimentalCoroutinesApi` to the declaration of `MainMotorTest`:

```
@ExperimentalCoroutinesApi
@RunWith(AndroidJUnit4::class)
class MainMotorTest {
```

Next, in the `initialLoad()` function, change:

```
whenever(repo.load()).thenReturn(Observable.just(listOf(TEST_MODEL)))
```

<div align="right">(from <u>app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt</u>)</div>

...to:

```
whenever(repo.load()).thenReturn(flowOf(listOf(TEST_MODEL)))
```

Here, we use `flowOf()`, which is the `Flow` equivalent of `Observable.just()`. `flowOf()` creates a `Flow` that emits the object(s) that we provide — here we are just providing one, but `flowOf()` accepts an arbitrary number of objects.

Then, replace:

```
androidSchedulerRule.scheduler.triggerActions()
```

<div align="right">(from <u>app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt</u>)</div>

...with:

```
mainDispatcherRule.dispatcher.runCurrent()
```

This replaces our manual trigger of the `TestScheduler` with an equivalent manual trigger of the `TestCoroutineDispatcher`.

At this point, `initialLoad()` should have no more compile errors and should look

like:

```kotlin
@Test
fun initialLoad() {
  whenever(repo.load()).thenReturn(flowOf(listOf(TEST_MODEL)))

  val underTest = makeTestMotor()
  val initialState = underTest.states.value

  assertThat(initialState is MainViewState.Loading, equalTo(true))
  mainDispatcherRule.dispatcher.runCurrent()

  val state = underTest.states.value as MainViewState.Content

  assertThat(state.observations.size, equalTo(1))
  assertThat(state.observations[0], equalTo(TEST_ROW_STATE))
}
```

Now, we need to fix initialLoadError(). Change:

```kotlin
whenever(repo.load()).thenReturn(Observable.error(TEST_ERROR))
```

<div align="right">(from <u>app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt</u>)</div>

...to:

```kotlin
whenever(repo.load()).thenReturn(flow { throw TEST_ERROR })
```

Observable.error() creates an Observable that throws the supplied exception when it is subscribed to. The equivalent is to use the flow() creator function and simply throw the exception yourself — that will occur when something starts consuming the flow.

Then, replace:

```kotlin
androidSchedulerRule.scheduler.triggerActions()
```

<div align="right">(from <u>app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt</u>)</div>

...with:

```kotlin
mainDispatcherRule.dispatcher.runCurrent()
```

As before, this replaces our manual trigger of the TestScheduler with an equivalent manual trigger of the TestCoroutineDispatcher.

At this point, `initialLoadError()` should have no more compile errors and should look like:

```kotlin
@Test
fun initialLoadError() {
  whenever(repo.load()).thenReturn(flow { throw TEST_ERROR })

  val underTest = makeTestMotor()
  val initialState = underTest.states.value

  assertThat(initialState is MainViewState.Loading, equalTo(true))
  mainDispatcherRule.dispatcher.runCurrent()

  val state = underTest.states.value as MainViewState.Error

  assertThat(TEST_ERROR, equalTo(state.throwable))
}
```

Next up is the `refresh()` test function. Replace:

```kotlin
val testSubject: PublishSubject<List<ObservationModel>> =
  PublishSubject.create()

whenever(repo.load()).thenReturn(testSubject)
whenever(repo.refresh()).thenReturn(Completable.complete())
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

...with:

```kotlin
val channel = Channel<List<ObservationModel>>()

whenever(repo.load()).thenReturn(channel.consumeAsFlow())
```

As with RxJava, we need to set up a `Flow` where we can deliver multiple values over time. One way to do that is to set up a `Channel`, then use `consumeAsFlow()` to convert it into a `Flow`. We can then use the `Channel` to `offer()` objects to be emitted by the `Flow`. So, we switch our mock `repo.load()` call to use this approach, and we get rid of the `repo.refresh()` mock, as `refresh()` no longer returns anything, so stock Mockito behavior is all that we need.

You will see that `consumeAsFlow()` has a warning. This is part of a `Flow` preview API and has its own warning separate from the one that we fixed by adding `@ExperimentalCoroutinesApi` to the class. To fix this one, add `@FlowPreview` to the class:

```
@FlowPreview
@ExperimentalCoroutinesApi
@RunWith(AndroidJUnit4::class)
class MainMotorTest {
```

Next, replace:

```
testSubject.onNext(listOf(TEST_MODEL))
androidSchedulerRule.scheduler.triggerActions()
```

<div align="right">(from <u>app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt</u>)</div>

...with:

```
channel.offer(listOf(TEST_MODEL))
mainDispatcherRule.dispatcher.runCurrent()
```

Now we offer() our model data to the Channel to be emitted by the Flow, and we advance our TestCoroutineDispatcher to process that event.

Then, replace:

```
testSubject.onNext(listOf(TEST_MODEL, TEST_MODEL_2))
androidSchedulerRule.scheduler.triggerActions()
```

<div align="right">(from <u>app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt</u>)</div>

...with:

```
channel.offer(listOf(TEST_MODEL, TEST_MODEL_2))
mainDispatcherRule.dispatcher.runCurrent()
```

This is the same basic change as the previous one, just for the second set of model objects.

At this point, refresh() should have no compile errors and should look like:

```
@Test
fun refresh() {
  val channel = Channel<List<ObservationModel>>()

  whenever(repo.load()).thenReturn(channel.consumeAsFlow())

  val underTest = makeTestMotor()
  val initialState = underTest.states.value
```

```kotlin
    assertThat(initialState is MainViewState.Loading, equalTo(true))

    channel.offer(listOf(TEST_MODEL))
    mainDispatcherRule.dispatcher.runCurrent()

    underTest.refresh()

    channel.offer(listOf(TEST_MODEL, TEST_MODEL_2))
    mainDispatcherRule.dispatcher.runCurrent()

    val state = underTest.states.value as MainViewState.Content

    assertThat(state.observations.size, equalTo(2))
    assertThat(state.observations[0], equalTo(TEST_ROW_STATE))
    assertThat(state.observations[1], equalTo(TEST_ROW_STATE_2))
  }
```

Finally, we need to fix the clear() test function. This will use the same techniques that we used for the refresh() test function.

Replace:

```kotlin
    val testSubject: PublishSubject<List<ObservationModel>> =
      PublishSubject.create()

    whenever(repo.load()).thenReturn(testSubject)
    whenever(repo.clear()).thenReturn(Completable.complete())
```

<div align="right">(from <u>app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt</u>)</div>

...with:

```kotlin
    val channel = Channel<List<ObservationModel>>()

    whenever(repo.load()).thenReturn(channel.consumeAsFlow())
```

Next, replace:

```kotlin
    testSubject.onNext(listOf(TEST_MODEL))
    androidSchedulerRule.scheduler.triggerActions()
```

<div align="right">(from <u>app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt</u>)</div>

...with:

```
channel.offer(listOf(TEST_MODEL))
mainDispatcherRule.dispatcher.runCurrent()
```

Then, replace:

```
testSubject.onNext(listOf())
androidSchedulerRule.scheduler.triggerActions()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

...with:

```
channel.offer(listOf())
mainDispatcherRule.dispatcher.runCurrent()
```

At this point, clear() should have no compile errors and should look like:

```kotlin
@Test
fun clear() {
  val channel = Channel<List<ObservationModel>>()

  whenever(repo.load()).thenReturn(channel.consumeAsFlow())

  val underTest = makeTestMotor()
  val initialState = underTest.states.value

  assertThat(initialState is MainViewState.Loading, equalTo(true))

  channel.offer(listOf(TEST_MODEL))
  mainDispatcherRule.dispatcher.runCurrent()

  underTest.clear()

  channel.offer(listOf())
  mainDispatcherRule.dispatcher.runCurrent()

  val state = underTest.states.value as MainViewState.Content

  assertThat(state.observations.size, equalTo(0))
}
```

And, most importantly, if you run the tests in MainMotorTest, they should all pass.

At this point, you can remove all import statements related to RxJava, such as Observable.

229

Step #14: Remove RxJava

At this point, we no longer need RxJava in the project. So, you can remove the following lines from the dependencies closure of app/build.gradle:

```
implementation "androidx.room:room-rxjava2:$room_version"
implementation "io.reactivex.rxjava2:rxandroid:2.1.1"
implementation 'io.reactivex.rxjava2:rxkotlin:2.3.0'
implementation "com.squareup.retrofit2:adapter-rxjava2:$retrofit_version"
implementation 'nl.littlerobots.rxlint:rxlint:1.7.4'
```

(note: those lines are not contiguous in app/build.gradle, but rather are scattered within the dependencies closure)

After this change — and the obligatory "sync Gradle with project files" step — both the app and the tests should still work.

And you're done!