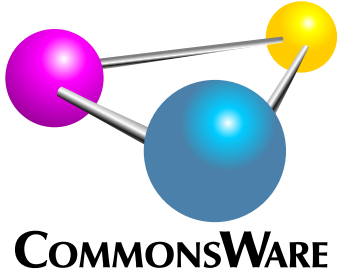
Version 0.1

Elements of Kotlin Coroutines

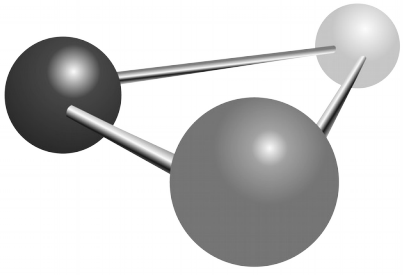
Mark L. Murphy



CommonsWare

Elements of Kotlin Coroutines

by Mark L. Murphy



COMMONSWARE

Elements of Kotlin Coroutines
by Mark L. Murphy

Printing History:
August 2019 Version 0.1

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 versions of Kotlin 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 0.1?

Everything!

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 o 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 August 2023*. 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 already is supporting coroutines in some of the Android Jetpack Kotlin extension libraries ("Android KTX") and will expand upon those in the coming years.

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:

```
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`;

```
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 (Throwable t) {
  // 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:

```
someCoroutineScope.launch {
  doSomething();

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

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

```
  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:

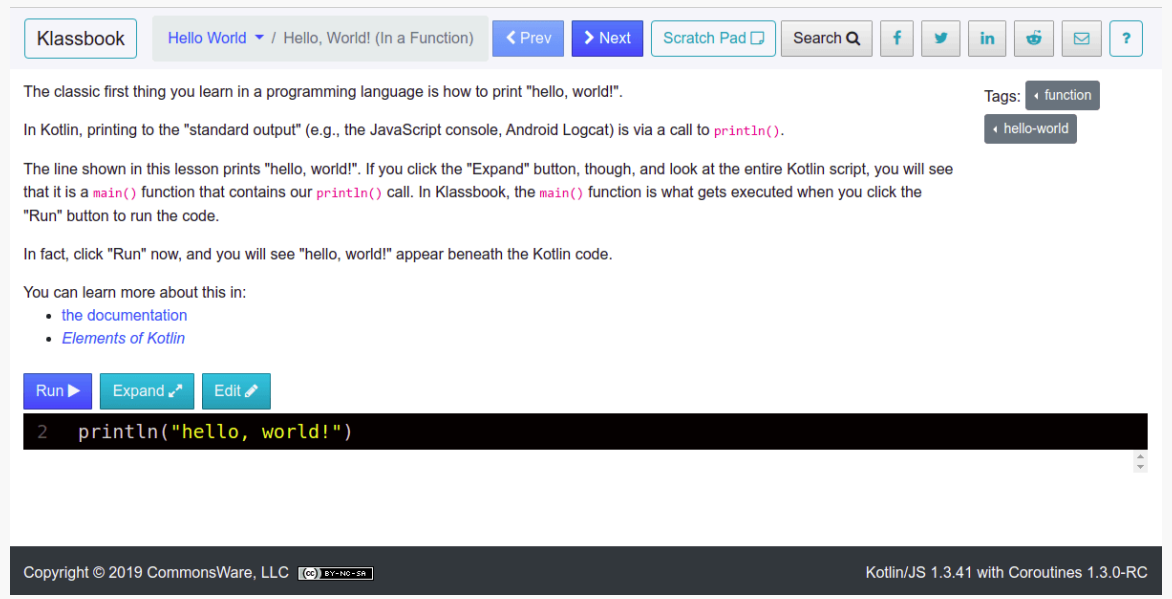


Figure 1: Klassbook Sample, As Initially Displayed

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:

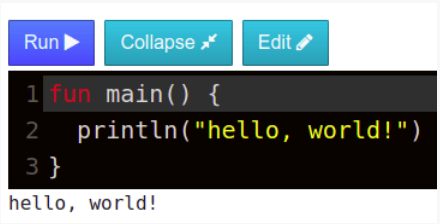


Figure 3: Klassbook Sample, with Expanded Snippet

If you want to pay 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:

```
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 versions `1.2.2` and `1.3-RC` of these dependencies. Note that these versions are somewhat independent of the overall Kotlin version (1.3).

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 `startForTime()` 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` or `runOnUiThread()` on an `Activity`, to have that `Runnable` code be executed on that 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 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()` global 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 thread pool. 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

```
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.

Parallel Coroutines

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

```
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)
  }
}
```

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

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 in parallel. 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:

```
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 somewhat in parallel, 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 otherwise, lest their app 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.

NOTE: Flows are still an experimental API at the moment, though they should be ready for production use very soon.

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.

Channels and flows 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 flow usually models a cold stream, while a channel usually models a hot stream.

Dependencies

Channels are part of current stable versions of Kotlin.

Flows, though, are still in a pre-release state. You will need a pre-release version of `1.3.0` of the coroutines dependencies, such as `1.3.0-RC`, to have access to flows. However, no new dependencies are required.

`Flow` Basics

As you might expect, a flow in Kotlin is represented by a `Flow` object, and a channel is represented by a `Channel` 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:

```
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

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

```
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 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.

```
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 the previous chapter, 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 great detail later in the book.

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

```
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:

```
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 "async()" in the Klassbook)

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 later in the book, when we look at `Job`.

`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:

```
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:

```
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:

```
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:

```
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`. `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.

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 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 fail to have 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, 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 injection 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 in an upcoming chapter.

`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 "A Simple Coroutine Sample" in the Klassbook)

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.

Android developers can think of `launch(Dispatchers.Main)` as being a bit like `post()` on `Handler` or `View`. `post()` takes a `Runnable` and will 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.

Some Context for Coroutines

Let’s go back to the first coroutine example:

```
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:

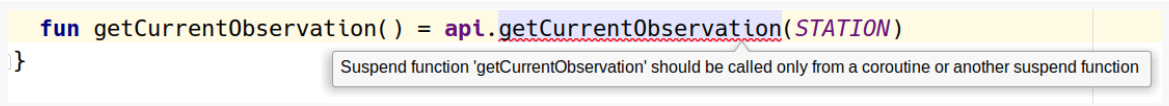


Figure 4: Android Studio Error About Missing `suspend` Keyword

Conversely, having `suspend` on a function that does not need one may result in a compiler warning and IDE "lint" message:

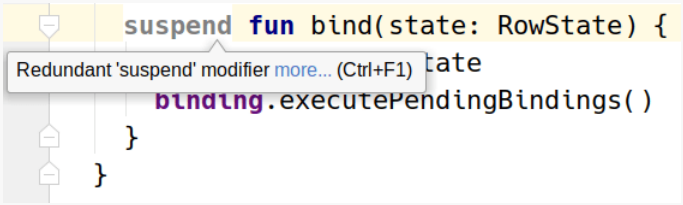


Figure 5: Android Studio Warning About Pointless `suspend` Keyword

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)
  }
}
```

(from "Contexts and Jobs" in the Klassbook)

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 "Jobs and Parents" in the Klassbook)

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:

```
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 the second message, only the first one. That is because the coroutine itself 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`:

```
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)
  }
}
```

(from "Starting Lazy Coroutines" in the Klassbook)

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")
}
```

(from "async()" in the Klassbook)

`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")
}
```

(from "join()" in the Klassbook)

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:

```
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)
  }
}
```

(from "cancelAndJoin()" in the Klassbook)

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()`:

```
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")
  }
}
```

(from "Canceling a Parent Cancels Its Children" in the Klassbook)

`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:

```
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)
  }
}
```

(from "Jobs Cancel When They Crash" in the Klassbook)

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() {
  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")
```

```
  }
}
```

(from "Parent Jobs Cancel When a Child Crashes" in the Klassbook)

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)
  }
}
```

(from "TimeoutCancellationException" in the Klassbook)

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)
  }
}
```

(from "withTimeoutOrNull()" in the Klassbook)

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 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()`:

```
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:

```
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:

```
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 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.

```
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:

```
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)
  }
}
```

(from "TimeoutCancellationException" in the Klassbook)

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)
  }
}
```

(from "invokeOnCompletion()" in the Klassbook)

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 be added in a future version of this book!

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:

```
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))
```

(from "flowOf()" in the Klassbook)

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 in the next chapter.

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 in an upcoming chapter.

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.

```
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!")
}
```

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

(from "single() and singleOrNull()" in the Klassbook)

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 `singleForNull()`, 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.

```
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)
  }
}
```

(from "first() and Flow" in the Klassbook)

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`:

```
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))
  }
}
```

(from "toList() and toSet()" in the Klassbook)

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... with one limitation.

Allowed: `suspend` Function and `withContext()`

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

```
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)
}
```

(from "suspend Functions and Flows" in the Klassbook)

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.

Not Allowed: `withContext()` in `flow()`

You might think that 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 is not allowed, though, leading to a compile error.

Allowed: `flowOn()`

That compile error gives us a hint of what we should do:

```
Using 'withContext(CoroutineContext, suspend () -> R): Unit' is an error.
withContext in flow body is deprecated, use flowOn instead
```

If you want to say that the body of the `flow()` lambda expression be invoked using a different dispatcher, 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. Specifically, you can call:

- `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) {
    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!
```

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)
- `switchMap()` to cancel the original `Flow` and switch to a new `Flow` created by a function that you supply
- `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 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:

```
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`:

```
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:

```
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 "catch()" in the Klassbook)

Tuning Into Channels

This chapter will be added in an upcoming version of this book!

Bridging to Callback APIs

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

Creating Custom Scopes

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

Testing Coroutines

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

Coroutines and Android

Applying Coroutines to Your UI

This chapter will be added in an upcoming version of this book!

Tying Scopes to Components

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

Using Coroutines with the Jetpack

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

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.

WARNING: The resulting coroutines-based app will be using a few pre-release artifacts. Consider this a "preview of coming attractions", more than a recommendation to use pre-release artifacts in production.

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:

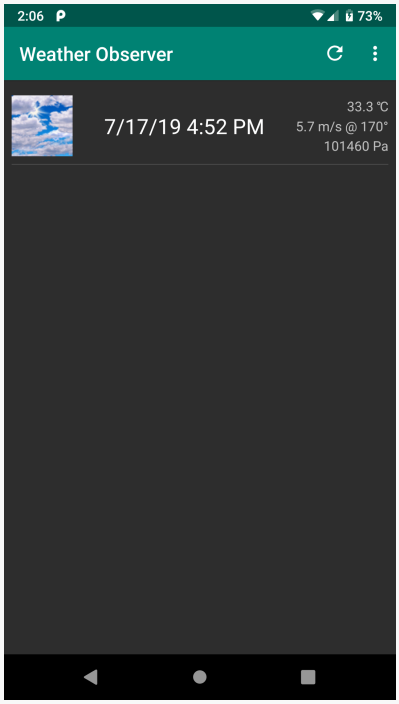


Figure 6: Weather Workshop Sample App, Showing Weather Info

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`:

```
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`:

```
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(
```

```
    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 `ObservationEntity`objects. 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.

`ObservationRepository`

The gateway to `ObservationRemoteDataSource` and `ObservationDatabase` is the `ObservationRepository`. This provides the public API that our UI uses to request weather observations:

```
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(
          "Unexpected windDirection unit: ${response.properties.windDirection.unitCode}"
        )
      response.properties.windSpeed.unitCode != "unit:m_s-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:

```
package com.commonsware.coroutines.weather

import android.app.Application
import com.jakewharton.threetenabp.AndroidThreeTen
import okhttp3.OkHttpClient
import org.koin.android.ext.android.startKoin
import org.koin.android.ext.koin.androidContext
import org.koin.androidx.viewmodel.ext.koin.viewModel
import org.koin.dsl.module.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()) }
    factory { 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(this, listOf(koinModule))
  }
}
```

(from app/src/main/java/com/commonsware/coroutines/weather/KoinApp.kt)

`MainMotor`

The app uses a variation on the model-view-intent (MVI) 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:

```
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))

      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()
        .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 MVI GUI architecture pattern in the "Adding Some Architecture" chapter of *Elements of Android Jetpack*!

`MainMotor` itself is an `AndroidViewModel` 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 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"
    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 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"
      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"
    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>
```

(from app/src/main/res/layout/row.xml)

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:

```
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)
  }
}
```

(from app/src/main/java/com/commonsware/coroutines/weather/BindingAdapters.kt)

The `MainActivity` Kotlin file not only contains the activity code, but also the adapter and view-holder for our `RecyclerView`:

```
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.Observer
import androidx.recyclerview.widget.*
import com.commonsware.coroutines.weather.databinding.RowBinding
import kotlinx.android.synthetic.main.activity_main.*
import org.koin.android.ext.android.inject

class MainActivity : AppCompatActivity() {
  private val motor: MainMotor by inject()
```

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

  val adapter = ObservationAdapter()

  observations.layoutManager = LinearLayoutManager(this)
  observations.adapter = adapter
  observations.addItemDecoration(
    DividerItemDecoration(
      this,
      DividerItemDecoration.VERTICAL
    )
  )

  motor.states.observe(this, Observer { state ->
    when (state) {
      MainViewState.Loading -> progress.visibility = View.VISIBLE
      is MainViewState.Content -> {
        progress.visibility = View.GONE
        adapter.submitList(state.observations)
      }
      is MainViewState.Error -> {
        progress.visibility = View.GONE
        Toast.makeText(
          this@MainActivity, state.throwable.localizedMessage,
          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`).

Google has recently upgraded Room to support `Flow` as a return type. 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-ktx:$room_version"
```

(from app/build.gradle)

This uses a `room_version` constant that we have set up to synchronize the versions of our various Room dependencies:

```
def room_version = "2.1.0"
```

(from app/build.gradle)

NOTE: If you wish to copy code from the book and paste it into your IDE, and you are using the PDF edition of the book, use Adobe Reader as your PDF viewer. Other PDF viewers seem to have difficulty with copying code snippets.

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>
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRemoteDataSource.kt)

to:

```
  suspend fun getCurrentObservation(@Path("stationId") stationId: String): ObservationResponse
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRemoteDataSource.kt)

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)
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRemoteDataSource.kt)

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()
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRemoteDataSource.kt)

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(
  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)

Step #5: Altering `ObservationDatabase`

Changing our database access is a matter of updating two function declarations on `ObservationStore`, replacing a `Completable` return type with `suspend`. So, change:

```
  @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:

```
  @Insert(onConflict = OnConflictStrategy.IGNORE)
  suspend fun save(entity: ObservationEntity)

  @Query("DELETE FROM observations")
  suspend fun clear()
```

Step #6: Adjusting `ObservationRepository`

Our `ObservationRepository` now needs revised `refresh()` and `clear()` functions, to use the new `suspend` functions from the data sources.

First, add `suspend` to `refresh()` and `clear()` — and remove the `Completable` return type — from the `IObservationRepository` interface that `ObservationRepository` implements:

```
interface IObservationRepository {
  fun load(): Observable<List<ObservationModel>>
  suspend fun refresh()
  suspend fun clear()
}
```

In terms of `ObservationRepository` itself, fixing `clear()` is easy — just take this:

```
override fun clear() = db.observationStore().clear()
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRepository.kt)

...and add the `suspend` keyword, giving you this:

```
override suspend fun clear() = db.observationStore().clear()
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRepository.kt)

`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) }
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRepository.kt)

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:

```
override suspend fun refresh() {
  db.observationStore().save(convertToEntity(remote.getCurrentObservation()))
}
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRepository.kt)

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()`.

Step #7: Modifying `MainMotor`

The last step is to update `MainMotor` to call the new `suspend` 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-viewmodel-ktx:2.1.0-rc01"
```

(from app/build.gradle)

Then, replace the existing `refresh()` and `clear()` functions with:

```
  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)
      }
    }
  }
```

(from app/src/main/java/com/commonsware/coroutines/weather/MainMotor.kt)

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.

At this point, if you run the app, it should work as it did before, just with somewhat less RxJava.

Step #8: Upgrading to `Flow` Support

Room `2.2.0-alpha02`, released in early August 2019, added support for `Flow` as a return type for `@Dao` functions. Right now, though, the project is using Room `2.1.0`.

So, adjust the value of `room_version` to be `2.2.0-alpha02`:

```
def room_version = "2.2.0-alpha02"
```

(from app/build.gradle)

This in turn will pull in newer versions of the coroutines dependencies, ones that contain `Flow` support.

Step #9: Amending `ObservationStore` for `Flow`

Now we can change the return type of `load()` to use `Flow` instead of `Observable`:

```
  @Query("SELECT * FROM observations ORDER BY timestamp DESC")
  fun load(): Flow<List<ObservationEntity>>
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationDatabase.kt)

And you can get rid of the RxJava-related `import` statements, both `Observable` and ones lingering from before (e.g., `Completable`).

The resulting `ObservationStore` should look like:

```
@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()
}
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationDatabase.kt)

...and `ObservationDatabase` overall should look like:

```
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()
  }
}
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationDatabase.kt)

Step #10: Updating `ObservationRepository` for `Flow`

`ObservationRepository` had been working with an `Observable` from

`ObservationStore` — now we need to adjust it to work with a `Flow` instead. Fortunately, this is a very minor change.

Modify the `IObservationRepository` interface to use `Flow` for the return value for `load()`:

```
interface IObservationRepository {
  fun load(): Flow<List<ObservationModel>>
  suspend fun refresh()
  suspend fun clear()
}
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRepository.kt)

Then, on `ObservationRepository` itself, replace the current `load()` function with:

```
  override fun load(): Flow<List<ObservationModel>> = db.observationStore()
    .load()
    .map { entities -> entities.map { it.toModel() }
  }
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRepository.kt)

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
```

(from app/src/main/java/com/commonsware/coroutines/weather/ObservationRepository.kt)

This just:

- Switches the return type from `Observable` to `Flow`
- Uses the `Flow.map()` extension function instead of the `map()` method on `Observable`

You can remove any RxJava-related `import` statements (e.g., `Observable`, `Single`).

At this point, `ObservationRepository` should look like:

```
package com.commonsware.coroutines.weather

import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.map
```

```
interface IObservationRepository {
  fun load(): Flow<List<ObservationModel>>
  suspend fun refresh()
  suspend fun clear()
}

class ObservationRepository(
  private val db: ObservationDatabase,
  private val remote: ObservationRemoteDataSource
) : IObservationRepository {
  override fun load(): Flow<List<ObservationModel>> = db.observationStore()
    .load()
    .map { entities -> entities.map { it.toModel() }
  }

  override suspend fun refresh() {
    db.observationStore().save(convertToEntity(remote.getCurrentObservation()))
  }

  override suspend 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(
          "Unexpected windDirection unit: ${response.properties.windDirection.unitCode}"
        )
      response.properties.windSpeed.unitCode != "unit:m_s-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)

Step #11: Collecting our `Flow` in `MainMotor`

Finally, `MainMotor` now needs to switch from observing an `Observable` to collecting a `Flow`.

Replace the `init` block in `MainMotor` with this:

```
  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)
      }
    }
  }
```

(from app/src/main/java/com/commonsware/coroutines/weather/MainMotor.kt)

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))

      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()
          .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)
      }
    }
  }
}
```

(from app/src/main/java/com/commonsware/coroutines/weather/MainMotor.kt)

And, at this point, if you run the app, it should work as it did before, just without RxJava.

Step #12: 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`:

```
  @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:

```
@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))
}
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

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:

```
@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))
}
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

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`:

```
@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 #13: 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 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.3.0-RC'
```

(from app/build.gradle)

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:

```
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()
  }
}
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainDispatcherRule.kt)

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:

```
  @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)
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

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 {
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

Next, in the `initialLoad()` function, change:

```
    whenever(repo.load()).thenReturn(Observable.just(listOf(TEST_MODEL)))
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

...to:

```
    whenever(repo.load()).thenReturn(flowOf(listOf(TEST_MODEL)))
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

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()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

...with:

```
    mainDispatcherRule.dispatcher.runCurrent()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

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:

```
@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))
}
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

Now, we need to fix `initialLoadError()`. Change:

```
  whenever(repo.load()).thenReturn(Observable.error(TEST_ERROR))
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

...to:

```
  whenever(repo.load()).thenReturn(flow { throw TEST_ERROR })
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

`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:

```
  androidSchedulerRule.scheduler.triggerActions()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

...with:

```
  mainDispatcherRule.dispatcher.runCurrent()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

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:

```
@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))
}
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

Next up is the `refresh()` test function. Replace:

```
  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:

```
  val channel = Channel<List<ObservationModel>>()

  whenever(repo.load()).thenReturn(channel.consumeAsFlow())
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

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 {
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

Next, replace:

```
    testSubject.onNext(listOf(TEST_MODEL))
    androidSchedulerRule.scheduler.triggerActions()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

...with:

```
    channel.offer(listOf(TEST_MODEL))
    mainDispatcherRule.dispatcher.runCurrent()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

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()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

...with:

```
    channel.offer(listOf(TEST_MODEL, TEST_MODEL_2))
    mainDispatcherRule.dispatcher.runCurrent()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

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

  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))
}
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

Finally, we need to fix the `clear()` test function. This will use the same techniques that we used for the `refresh()` test function.

Replace:

```
  val testSubject: PublishSubject<List<ObservationModel>> =
    PublishSubject.create()

  whenever(repo.load()).thenReturn(testSubject)
  whenever(repo.clear()).thenReturn(Completable.complete())
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

...with:

```
    val channel = Channel<List<ObservationModel>>()

    whenever(repo.load()).thenReturn(channel.consumeAsFlow())
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

Next, replace:

```
    testSubject.onNext(listOf(TEST_MODEL))
    androidSchedulerRule.scheduler.triggerActions()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

...with:

```
    channel.offer(listOf(TEST_MODEL))
    mainDispatcherRule.dispatcher.runCurrent()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

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()
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

At this point, `clear()` should have no compile errors and should look like:

```
@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))
  }
```

(from app/src/androidTest/java/com/commonsware/coroutines/weather/MainMotorTest.kt)

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`.

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!