Chapter 4. Concurrency in Android

This chapter does not focus specifically on Kotlin. Instead, it will introduce some of the issues that surround *concurrent programming* and that the rest of the book addresses. It will also introduce a few tools, already available to Android developers, for managing concurrent tasks.

Concurrent programming has a reputation as kind of a dark art: something that is done by self-proclaimed wizards and that novices touch at their peril. Certainly, writing correct concurrent programs can be quite challenging. This is particularly true because errors in concurrent programs don't always show up right away. It is nearly impossible to test for concurrency bugs and they can be extremely difficult to reproduce, even when they are known to exist.

A developer concerned about the hazards of concurrent programming would do well to remember these three things:

- Nearly everything you do, every day, *except* programming, is concurrent. You get along quite nicely in a concurrent environment. It is programming, where things happen in order, that is odd.
- If you are trying to understand the issues that concurrent programming presents, you are on the right path. Even an incomplete understanding of concurrency is better than copying sample code and crossing your fingers.
- Concurrent programming is just how Android works. Anything other than the most trivial Android application will require concurrent execution. Might as well get on with it and figure out what it's all about!

Before getting into specifics, let's define some terms.

The first term is *process*. A process is memory that an application can use, and one or more threads of execution. The memory space belongs to the process—no other processes can affect it.[1] An application usually runs as a single process.

That, of course, introduces the second term: *thread*. A thread is a sequence of instructions, executed one at a time, in order.

And this leads us to the term that, to some extent, drives the rest of this book: *thread safe*. A set of instructions is thread-safe if, when multiple threads execute it, no possible ordering of the instructions executed by the threads can produce a result that could not be obtained if each of the threads executed the code completely, in some order, one at a time. That's a little hard to parse, but it just says that the code produces the same results whether the multiple threads execute it all at the same time or, serially, one at a time. It means that running the program produces predictable results.

So how does one make a program thread-safe? There are lots and lots of ideas about this. We would like to propose one that is clear, relatively easy to follow, and always correct. Just follow one, fairly clear, fairly simple rule. We'll state the rule in a few pages. First, though, let's discuss in more detail what thread safety means.

Thread Safety

We've already said that thread-safe code cannot produce a result, when executed by multiple threads at the same time, that could not have been produced by some ordering of the threads executing one at a time. That definition, though, is not very useful in practice: no one is going to test all possible execution orders.

Perhaps we can get a handle on the problem by looking at some common ways that code can be *thread-unsafe*.

Thread-safety failures can be divided into a few categories. Two of the most important are *atomicity* and *visibility*.

Atomicity

Nearly all developers understand problems of atomicity. This code is not thread-safe:

```
fun unsafe() { globalVar++ }
```

Multiple threads executing this code can interfere with each other. Each thread executing this code might read the same value for `globalVar`—say, 3. Each might increment that value to get 4, and then each might update `globalVar` to have the value 4. Even if 724 threads executed the code, `globalVar` might, when all were through executing, have the value 4.

There is no possible way that each of those 724 threads could execute that code serially and have `globalVar` end up as 4. Because the result of executing the code concurrently can be different from any possible result generated by serial execution, this code is not thread-safe, according to our definition.

To make the code thread-safe, we need to make the read, increment, and write operations on the variable `globalVar`, together, *atomic*. An atomic operation is one that cannot be interrupted by another thread. If the read, increment, and write operations are atomic, then no two threads can see the same value of `globalVar`, and the program is guaranteed to behave as expected.

Atomicity is easy to understand.

Visibility

Our second category of thread-safety errors, visibility, is much more difficult to apprehend. This code is also not thread-safe:

```
var shouldStop = false

fun runner() {
    while (!shouldStop) { doSomething() }
}
```

```
fun stopper() { shouldStop = true }
```

A thread running the function `runner` may never stop, even though another thread runs `stopper`. The thread running `runner` may never notice that the value of `shouldStop` has changed to `true`.

The reason for this is optimization. Both the hardware (using registers, multilayer caches, etc.) and the compiler (using hoisting, reordering, etc.) do their very best to make your code run fast. In order to do this, the instructions that the hardware actually executes may not look much like the Kotlin source at all. In particular, while you think that `shouldStop` is a single variable, the hardware probably has at least two representations for it: one in a register and one in main memory.

You definitely want that! You would not want the loops in your code to depend on access to main memory instead of using caches and registers. Fast memory optimizes your code because it has access times that are several orders of magnitude faster than main memory.

To make the example code work, though, you have to explain to the compiler that it cannot keep the value of `shouldStop` in local memory (a register or cache). If, as proposed, there are multiple representations of `shouldStop` in different kinds of hardware memory, the compiler must be sure that the value kept in the fast, local representation of `shouldStop` is pushed to memory that is visible to all threads. This is called *publishing* the value.

`@Synchronized` is the way to do that. Synchronization tells the compiler that it must make sure that any side effects of the code executed within the synchronized block are visible to all other threads, before the executing thread leaves the block.

Synchronization, then, is not so much about hardware, or tricky and complicated criteria for what must be protected and what need not be. Synchronization is a contract between the developer and the compiler. If you don’t synchronize code, the compiler is free to make any optimization that it can prove safe, based on serial execution. If there is other code somewhere, running on a different thread, that makes the compiler's proof invalid, you must synchronize the code.

So, here's the rule. If you want to write code that is thread-safe, you just have to follow this one short, clear rule. Paraphrasing from Java's bible of parallel programming, *Java Concurrency in Practice*:[2] Whenever more than one thread accesses a given state variable, and one of them might write to it, they all must coordinate their access to it using synchronization.

Note, by the way, that that quote does not distinguish between read access and write access for synchronization. Unless it is guaranteed that *nobody* will mutate the shared state, all access, read or write, must be synchronized.

The Android Threading Model

As noted in Chapter 3, one of the implications of an MVC architecture is a single-threaded UI (the View and Controller). Although a multithreaded UI seems very tempting (surely a thread for the View and a thread for the Controller would work…), attempts to build them were abandoned back in the 1970s, when it became clear that they, inevitably, ended in a snarl of deadlocks.

Since the general adoption of MVC, the standard UI design is a queue serviced by a single thread (in Android, the *Main-*, or *UI-thread*). As illustrated in Figure 4-1, events—both those that originate with a user (clicks, taps, typing, etc.) and those that originate in the model (animation, requests to redraw/update the screen, etc.)—are enqueued and eventually processed in order by the single UI thread.

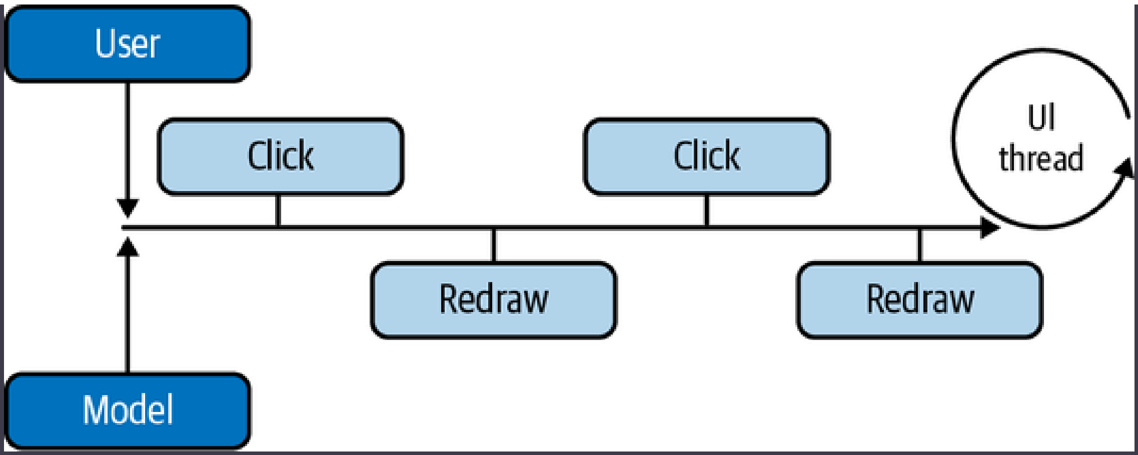

Figure 4-1. UI thread.

This is exactly how Android's UI works. An application's main thread (the application process's original thread) becomes its UI thread. As part of initialization, the thread enters a tight loop. For the rest of the life of the application, it removes tasks from the canonical UI queue one by one and executes them. Because UI methods are always run on a single thread, UI components make no attempt to be thread-safe.

That sounds great, right? A single-threaded UI and no worries about thread safety. There's a problem, though. To understand it, we'll have to switch out of our developer hats and talk a bit about the experience of the end users of Android devices. In particular, we'll need to look into some details of video display.

NOTE

Threads are said to *deadlock* when each holds a resource that the other requires: neither can make forward progress. For instance, one thread might hold the widget that displays a value and require the container that holds the value to be displayed. At the same time, another thread might hold the container and require the widget. Deadlocks can be avoided if all threads always seize resources in the same order.

Dropped Frames

We know, from long experience with motion pictures and TV, that the human brain can be tricked into perceiving motion as continuous, even when it is not. A series of still images shown rapidly, one after the other, can appear to the observer to be smooth, uninterrupted motion. The rate at which the images are displayed is known as the *frame rate*. It is measured in *frames per second* (fps).

The standard frame rate for movies is 24 fps. That has worked quite well for the entire Golden Age of Hollywood. Older televisions used a frame rate of 30 fps. As you might imagine, faster frame rates do an even better job of tricking the brain than slow ones. Even if you can't exactly put your finger on what you are sensing, if you watch a high frame rate video next to one with a lower frame rate, you will likely notice a difference. The faster one will feel "smoother."

Many Android devices use a frame rate of 60 fps. This translates to redrawing the screen once approximately every 16 milliseconds (ms). That means that the UI thread, the single thread handling UI tasks, must have a new image available, ready to be drawn on the screen every 16 ms. If producing the image takes longer than that, and the new image is not ready when the screen is redrawn, we say that the frame has been dropped.

It will be another 16 ms before the screen is redrawn again and a new frame becomes visible. Instead of 60 fps, a dropped frame lowers the frame rate to 30 fps, close to the threshold at which the human brain notices it. Just a few dropped frames can give a UI a choppy feeling that is sometimes called "jank."

Consider the queue of tasks shown in Figure 4-2, at Android's standard render rate of 60 fps.

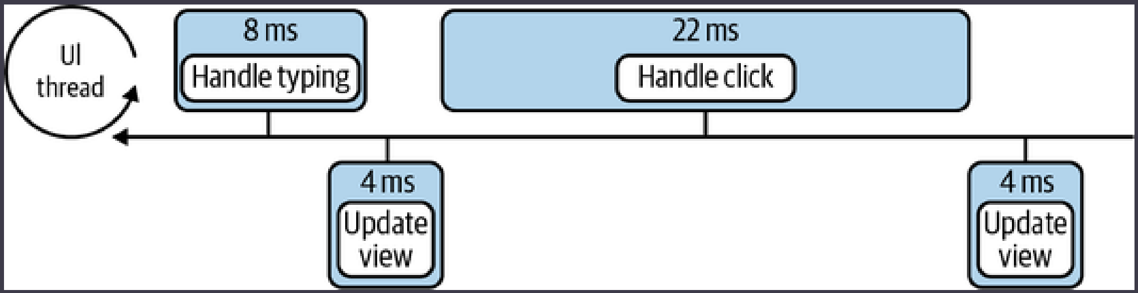

Figure 4-2. Tasks queued for the UI thread.

The first task, handling character input from the user, takes 8 ms to execute. The next task, updating the view, is part of an animation. In order to look smooth, the animation needs to be updated at least 24 times per second. The third task, though, handling a user click, takes 22 ms. The last task in the diagram is the next frame of the animation. Figure 4-3 shows what the UI thread sees.

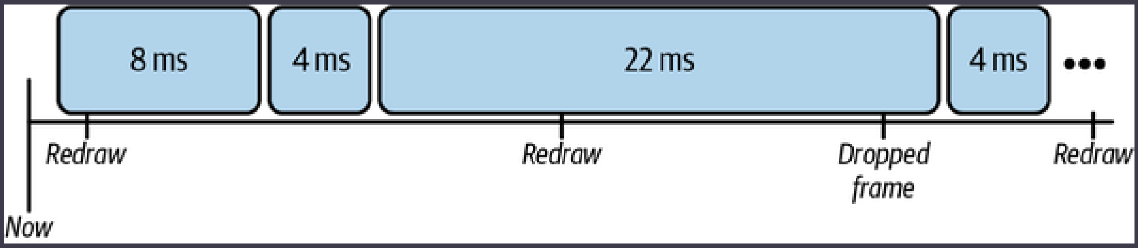

Figure 4-3. A dropped frame.

The first task completes in 8 ms. The animation draws a frame to the display buffer in 4 ms. The UI thread then starts to handle the click. A couple of milliseconds into handling the click, the hardware redraw takes place and the animation's frame is now visible on the screen.

Unfortunately, 16 ms later, the task to handle the click is still not complete. The task to draw the next frame of the animation, which is queued behind it, has not been processed. When the redraw happens, the contents of the display buffer are exactly as they were during the previous redraw. The animation frame has been dropped.

NOTE

Computer displays are usually managed using one or more display buffers. A *display buffer* is an area of memory in which user code "draws" the things that will be visible on the screen. Occasionally, at the *refresh interval* (approximately 16 ms for a 60 fps display), user code is briefly locked out of the buffer. The system uses the contents of the buffer to render the screen and then releases it back to the user code for further updates.

A few milliseconds later, when the click handling task is complete, the animation task gets its chance to update the display buffer. Even though the display buffer now contains the next frame of the animation, the screen will not be redrawn for several milliseconds. The frame rate for the animation has been cut in half, to 30 fps, dangerously close to visible flicker.

Some newer devices, like Google's Pixel 4, have the ability to refresh the screen at much higher frame rates. With a frame rate that is, for instance, twice as high (120 fps), even if the UI thread misses two frames in a row, it still only has to wait an extra 8 ms for the next redraw. The interval between the two renderings in this case is only around 24 ms; much better than the 32 ms cost of dropping a frame at 60 fps.

Though increased frame rate may help, an Android developer must be vigilant and make sure that an application drops as few frames as possible. If an app is in the middle of an expensive computation and that computation takes longer than expected to complete, it will miss the redraw time slot and drop the frame, and the application will feel janky.

This scenario is the reason why it is absolutely necessary to deal with concurrency in Android applications. Put simply, the UI is single-threaded and the UI thread must never be occupied for more than a few milliseconds

The only possible solution for a nontrivial application is to pass time-consuming work—database storage and retrieval, network interactions, and long-running computations—to some other thread.

Memory Leaks

We've already dealt with one complexity introduced by concurrency: thread safety. Android's component-based architecture adds a second, equally dangerous complexity: *memory leaks*.

A memory leak occurs when the object can't be freed (garbage-collected) even though it's no longer useful. At worst, memory leaks could result in an `OutOfMemoryError`, and an application crash. Even if things don't get that bad, though, running short on memory can force more frequent garbage collections that again cause "jank."

As discussed in Chapter 3, Android applications are particularly susceptible to memory leaks because the lifecycles of some of the most frequently used components—`Activity`s, `Fragment`s, `Service`s, and so on—are not under the control of the application. Instances of those components can all too easily turn into dead weight.

This is particularly true in a multithreaded environment. Consider offloading a task to a worker thread like this:

```
override fun onViewCreated(
    view: View,
    savedInstanceState: Bundle?
) {
    // DO NOT DO THIS!
    myButton.setOnClickListener {
        Thread {
            val status = doTimeConsumingThing()
            view.findViewById<TextView>(R.id.textview_second)
                .setText(status)
        }
            .start()
    }
}
```

The idea of moving the time-consuming work off the UI thread is a noble one. Unfortunately, the preceding code has several flaws. Can you spot them?

First, as mentioned earlier in this chapter, Android UI components are not thread-safe and cannot be accessed or modified from outside the UI thread. The call to `setText` in this code, from a thread other than the UI thread, is incorrect. Many Android UI components detect unsafe uses like this, and throw exceptions if they occur.

One way to address this problem is to return results to the UI thread using one of the Android toolkit methods for safe thread dispatch, as shown here. Note that this code *still* has flaws!

```
override fun onViewCreated(
    view: View,
    savedInstanceState: Bundle?
) {
    // DO NOT DO THIS EITHER!
    myButton.setOnClickListener {
        Thread {
            val status = doTimeConsumingThing()
            view.post {
                view.findViewById<TextView>(R.id.textview_second)
                    .setText(status)
            }
        }
            .start()
    }
}
```

That fixes the first issue (the UI method, `setText`, is now called from the Main thread) but the code is still not correct. Though the vagaries of the language make it hard to see the problem, it is that the thread, newly created in the `ClickListener`, holds an implicit reference to an Android-managed object. Since `doTimeConsumingThing` is a method on an `Activity` (or `Fragment`), the thread, newly created in the click listener, holds an *implicit* reference to that `Activity`, as shown in Figure 4-4.

Figure 4-4. A leaked activity.

It might be more obvious if the call to `doTimeConsumingThing` were written as `this.doTimeConsumingThing`. If you think about it, though, it is clear that there is no way to call the method `doTimeConsumingThing` on some object (in this case, an instance of an `Activity`) without holding a reference to that object. Now the `Activity` instance cannot be garbage-collected as long as the `Runnable` running on the worker thread holds a reference to it. If the thread runs for any significant amount of time, `Activity` memory has leaked.

This issue is considerably more difficult to address than the last. One approach assumes that tasks that are guaranteed to hold such an implicit reference for only a very short period of time (less than a second) may not cause a problem. The Android OS itself occasionally creates such short-lived tasks.

`ViewModel`s and `LiveData` ensure that your UI always renders the freshest data, and does it safely. Combined with Jetpack's `viewModelScope` and coroutines—both to be introduced shortly—all these things make it easier to control cancellation of background tasks that are no longer relevant, and ensure memory integrity and thread safety. Without the libraries, we'd have to correctly address all of these concerns ourselves.

NOTE

Careful design using Jetpacks' lifecycle-aware, observable `LiveData` containers, as described in Chapter 3, can help to eliminate both memory leaks and the danger of using an Android component that has completed its lifecycle.

Tools for Managing Threads

There is, actually, a third flaw in the code we just discussed; a deep design flaw.

Threads are very expensive objects. They are large, they affect garbage collection, and switching context among them is far from free. Creating and destroying threads, as the code in the example does, is quite wasteful, ill-advised, and likely to affect application performance.

Spawning more threads in no way makes an application able to accomplish more work: a CPU has only so much power. Threads that are not executing are simply an expensive way of representing work that is not yet complete.

Consider, for instance, what would happen if a user mashed `myButton`, from the previous example. Even if the operations that each of the generated threads performed were fast and thread-safe, creating and destroying those threads would slow the app to a crawl.

A best practice for applications is a thread policy: an application-wide strategy based on the number of threads that is actually useful, that controls how many threads are running at any given time. A smart application maintains one or more pools of threads, each with a particular purpose, and each fronted by a queue. Client code, with work to be done, enqueues tasks to be executed by the pool threads and, if necessary, recovers the task results.

The next two sections introduce two threading primitives available to Android developers, the `Looper`/`Handler` and the `Executor`.

Looper/Handler

The `Looper`/`Handler` is a framework of cooperating classes: a `Looper`, a `MessageQueue` and the `Message`s enqueued on it, and one or more `Handler`s.

A `Looper` is simply a Java `Thread` that is initialized by calling the methods `Looper.prepare()` and `Looper.start()` from its `run` method, like this:

```
var looper = Thread {
    Looper.prepare()
    Looper.loop()
}
looper.start()
```

The second method, `Looper.loop()`, causes the thread to enter a tight loop in which it checks its `MessageQueue` for tasks, removes them one by one, and executes them. If there are no tasks to be executed, the thread sleeps until a new task is enqueued.

NOTE

If you find yourself thinking that this sounds vaguely familiar, you are right. Android's UI thread is simply a `Looper` created from the application process's main thread.

A `Handler` is the mechanism used to enqueue tasks on a `Looper`'s queue, for processing. You create a `Handler` like this:

```
var handler = new Handler(someLooper);
```

The main thread's `Looper` is always accessible using the method `Looper.getMainLooper`. Creating a `Handler` that posts tasks to the UI thread, then, is as simple as this:

```
var handler = new Handler(Looper.getMainLooper);
```

In fact, this is exactly how the `post()` method, shown in the preceding example, works.

`Handler`s are interesting because they handle both ends of the `Looper`'s queue. In order to see how this works, let's follow a single task through the `Looper`/`Handler` framework.

There are several `Handler` methods for enqueuing a task. Here are two of them:

- `post(task: Runnable)`
- `send(task: Message)`

These two methods define two slightly different ways of enqueuing a task: sending messages and posting `Runnable`s. Actually, the `Handler` always enqueues a `Message`. For convenience, though, the `post...()` group of methods attach a `Runnable` to the `Message` for special handling.

In this example we use the method `Handler.post(task: Runnable)` to enqueue our task. The `Handler` obtains a `Message` object from a pool, attaches the `Runnable`, and adds the `Message` to the end of the `Looper`'s `MessageQueue`.

Our task is now awaiting execution. When it reaches the head of the queue, the `Looper` picks it up and, interestingly, hands it right back to the exact same `Handler` that enqueued it. The same `Handler` instance that enqueues a task is always the instance that runs it.

This can seem a bit perplexing until you realize that the `Handler` code that submitted the task might be running on any application thread. The `Handler` code that processes the task, however, is *always* running on the `Looper`, as shown in Figure 4-5.

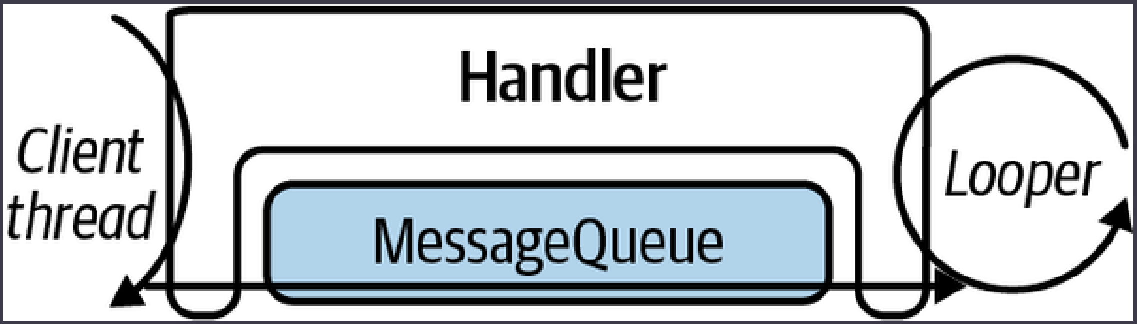

Figure 4-5. `Looper`/`Handler`.

The `Handler` method called by the `Looper` to handle a task first checks to see if the `Message` contains a `Runnable`. If it does—and because we used one of the `post...()` methods, our task does—the `Handler` executes the `Runnable`.

If we'd used one of the `send...()` methods, the `Handler` would have passed the `Message` to its own overridable method, `Handler.handleMessage(msg: Message)`. A subclass of `Handler` would, in that method, use the `Message` attribute `what` to decide which particular task it should perform, and the attributes `arg1`, `arg2`, and `obj` as task parameters.

The `MessageQueue` is, actually, a sorted queue. Each `Message` includes, as one of its attributes, the earliest time at which it may be executed. In the preceding two methods, `post` and `send`, simply use the current time (the message will be processed “now,” immediately).

Two other methods, though, allow tasks to be enqueued to be run at some time in the future:

- `postDelayed(runnable, delayMillis)`
- `sendMessageDelayed(message, delayMillis)`

Tasks created using these methods will be sorted into the `MessageQueue` to be executed at the indicated time.

NOTE

As noted, a `Looper` can only make a best effort at running a task at the requested time. While it will never run a delayed task before its time, if another task hogs the thread the task may run late.

`Looper`/`Handler`s are a fantastically versatile and efficient tool. The Android system makes extensive use of them, particularly the `send...()` calls, which do not do any memory allocation.

Note that a `Looper` can submit tasks to itself. Tasks that execute and then reschedule themselves after a given interval (using one of the `...Delayed()` methods) are one of the ways that Android creates animations.

Also note that because a `Looper` is single-threaded, a task that is only run on one particular `Looper` need not be thread-safe. There is no need for synchronization or ordering when a task, even a task that is run asynchronously, is run only on a single thread. As mentioned earlier, the entire Android UI framework, which runs only on the UI Looper, depends on this assumption.

Executors and ExecutorServices

Java introduced `Executor`s and `ExecutorService`s in Java 5, as part of a new Concurrency Framework. The new framework provided several higher-level concurrency abstractions that allowed developers to leave behind many of the details of threads, locks, and synchronization.

An `Executor` is, as its name suggests, a utility that executes tasks submitted to it. Its contract is the single method `execute(Runnable)`.

Java provides several implementations of the interface, each with a different execution strategy and purpose. The simplest of these is available using the method `Executors.newSingleThreadExecutor`.

A single-threaded executor is very similar to the `Looper`/`Handler` examined in the previous section: it is an unbounded queue in front of a single thread. New tasks are enqueued onto the queue and then removed and executed in order on the single thread that services the queue.

`Looper`/`Handler`s and single-threaded `Executor`s each have their own advantages. For instance, a `Looper`/`Handler` is heavily optimized, to avoid object allocation. On the other hand, a single-threaded `Executor` will replace its thread if that thread is aborted by a failing task.

A generalization of the single-threaded `Executor` is the `FixedThreadPoolExecutor`: instead of a single thread, its unbounded queue is serviced by a fixed number of threads. Like the single-threaded `Executor`, a `FixedThreadPoolExecutor` will replace threads when tasks kill them. A `FixedThreadPoolExecutor` does not guarantee task order, though, and will execute tasks simultaneously, hardware permitting.

The single-threaded scheduled `Executor` is Java's equivalent of the `Looper`/`Handler`. It's similar to a single-threaded `Executor` except that, like the `Looper`/`Handler`, its queue is sorted by execution time. Tasks are executed in time order, not submission order. As with the `Looper`/`Handler`, of course, long-running tasks can prevent subsequent tasks from being executed on time.

If none of these standard execution utilities meets your needs, you can create a custom instance of `ThreadPoolExecutor`, specifying details like the size and ordering of its queue, number of threads in its thread pool and how they are created, and what happens when the pool's queue is full.

There is one more type of `Executor` that deserves special attention—the `ForkJoinPool`. Fork-join pools exist because of the observation that sometimes a single problem can be broken down into multiple subproblems which can be executed concurrently.

A common example of this kind of problem is adding two same-size arrays together. The synchronous solution is to iterate, `i = 0 .. n - 1`, where `n` is the size of the array, and at each `i` to compute `s[i] = a1[i] + a2[i]`.

There is a clever optimization that is possible, though, if the task is divided into pieces. In this case, the task can be subdivided into `` n` `` subtasks, each of which computes `s[i] = a1[i] + a2[i]` for some `i`.

Note that an execution service creating subtasks it expects to process *itself* can enqueue the subtasks on a thread-local queue. Since the local queue is used predominantly by the single thread, there is almost never contention for the queue locks. Most of the time, the queue belongs to the thread—it alone puts things on and takes them off. This can be quite an optimization.

Consider a pool of these threads, each with its own fast, local queue. Suppose that one of the threads finishes all of its work and is about to idle itself, while at the same time another pool thread has a queue of 200 subtasks to execute. The idle thread steals the work. It grabs the lock for the busy thread's queue, grabs half of the subtasks, puts them in its own queue, and goes to work on them.

The work-stealing trick is most useful when concurrent tasks spawn their own subtasks. As we will see, it turns out that Kotlin coroutines are exactly such tasks.

Tools for Managing Jobs

Just as there can be economies of scale in the production of, say, cars, there are important optimizations that require the large-scale view of a system. Consider the use of the radio on a mobile phone.

When an application needs to interact with a remote service, the phone, normally in battery-saving mode, must power up its radio, connect to a nearby tower, negotiate a connection, and then transmit its message. Because connection negotiation is overhead, the phone holds the connection open for a while. The assumption is that, when one network interaction takes place, it is likely that others will follow. When more than a minute or so goes by without any use of the network, though, the phone goes back to its quiescent, battery-saving state.

Given this behavior, imagine what happens when several applications phone home, each at a different time. When the first app sends its ping, the phone powers its radio up, negotiates the connection, transmits a message for the app, waits a bit, and then goes back to sleep. Just as it goes back to sleep, though, the next application tries to use the network. The phone has to power back up, renegotiate a connection, and so on. If there are more than a handful of applications doing this, the phone radio is at full power essentially all the time. It is also spending a lot of that time renegotiating a network connection that it dropped just a few seconds ago.

No single application can prevent this kind of problem. It requires a system-wide view of battery and network use to coordinate multiple apps (each with its own requirements) and to optimize battery life.

Android 8.0 (API 26+) introduced limits on application resource consumption. Included in these limitations are the following:

- An application is in the foreground only when it has a visible activity or is running a foreground service. Bound and started `Service`s no longer prevent an application from being killed.

- Applications cannot use their manifest to register for implicit broadcasts. There are also limitations on sending broadcasts.

These constraints can make it difficult for an application to perform "background" tasks: synching with a remote, recording location, and so on. In most cases, the constraints can be mitigated using the `JobScheduler` or Jetpack's `WorkManager`.

Whenever medium to large tasks have to be scheduled more than a few minutes in the future, it is a best practice to use one of these tools. Size matters: refreshing an animation every few milliseconds, or scheduling another location check in a couple of seconds, is probably a fine thing to do with a thread-level scheduler. Refreshing a database from its upstream every 10 minutes is definitely something that should be done using the `JobScheduler`.

JobScheduler

The `JobScheduler` is Android's tool for scheduling tasks—possibly repeating tasks—in the future. It is quite adaptable and, in addition to optimizing battery life, provides access to details of system state that applications used to have to infer from heuristics.

A `JobScheduler` job is, actually, a bound service. An application declares a special service in its manifest to make it visible to the Android system. It then schedules tasks for the service using `JobInfo`.

When the `JobInfo`'s conditions are met, Android binds the task service, much as we described in "Bound Services". Once the task has been bound, Android instructs the service to run and passes any relevant parameters.

The first step in creating a `JobScheduler` task is registering it in the application manifest. That is done as shown here:

```
<service
    android:name=".RecurringTask"
    android:permission="android.permission.BIND_JOB_SERVICE"/>
```

The important thing in this declaration is the permission. Unless the service is declared with *exactly* the `android.permission.BIND_JOB_SERVICE` permission, the `JobScheduler` will not be able to find it.

Note that the task service is not visible to other applications. This is not a problem. The `JobScheduler` is part of the Android system and can see things that normal applications cannot.

The next step in setting up a `JobScheduler` task is scheduling it, as shown here, in the method `schedulePeriodically`:

```
const val TASK_ID = 8954
const val SYNC_INTERVAL = 30L
const val PARAM_TASK_TYPE = "task"
const val SAMPLE_TASK = 22158

class RecurringTask() : JobService() {
    companion object {
        fun schedulePeriodically(context: Context) {
            val extras = PersistableBundle()
            extras.putInt(PARAM_TASK_TYPE, SAMPLE_TASK)

            (context.getSystemService(Context.JOB_SCHEDULER_SERVICE)
                as JobScheduler)
                .schedule(
                    JobInfo.Builder(
                        TASK_ID,
                        ComponentName(
                            context,
                            RecurringTask::class.java
                        )
                    )
                        .setPeriodic(SYNC_INTERVAL)
                        .setRequiresStorageNotLow(true)
                        .setRequiresCharging(true)
                        .setExtras(extras)
                        .build()
                )
        }
    }

    override fun onStartJob(params: JobParameters?): Boolean {
        // do stuff
        return true;
    }

    override fun onStopJob(params: JobParameters?): Boolean {
        // stop doing stuff
        return true;
    }
}
```

This particular task will be run every `SYNC_INTERVAL` seconds but only if there is sufficient space on the device and if it is currently attached to an external power source. These are only two of the wide variety of attributes available for scheduling a task. The granularity and flexibility of scheduling is, perhaps, the `JobScheduler`'s most appealing quality.

Note that `JobInfo` identifies the task class to be run in much the same way that we identified the target for an `Intent` back in Chapter 3.

The system will call the task's `onStartJob` method based on the criteria set in the `JobInfo` when the task is eligible to run. This is why the `JobScheduler` exists. Because it knows the schedules and requirements for all scheduled tasks, it can optimize scheduling, globally, to minimize the impact, especially on the battery.

Beware! The `onStartJob` method is run on the main (UI) thread. If, as is very likely, the scheduled task is something that will take more than a few milliseconds, it must be scheduled on a background thread, using one of the techniques described previously.

If `onStartJob` returns `true`, the system will allow the application to run until either it calls `jobFinished` or the conditions described in the `JobInfo` are no longer satisfied. If, for instance, the phone running the `RecurringTask` in the previous example was unplugged from its power source, the system would immediately call the running task's `onStopJob()` method to notify it that it should stop.

When a `JobScheduler` task receives a call to `onStopJob()` it must stop. The documentation suggests that the task has a little bit of time to tidy up and terminate cleanly. Unfortunately, it is quite vague about exactly how much time is a "little bit." It is quite dire, though, in its warning that "You are solely responsible for the behavior of your application upon receipt of this message; your app will likely start to misbehave if you ignore it."

If `onStopJob()` returns `false`, the task will not be scheduled again, even if the criteria in its `JobInfo` are met: the job has been cancelled. A recurring task should always return `true`.

WorkManager

The `WorkManager` is an Android Jetpack library that wraps the `JobScheduler`. It allows a single codebase to make optimal use of modern versions of Android—those that support the `JobScheduler`—and still work on legacy versions of Android that do not.

While the services provided by the `WorkManager`, as well as its API, are similar to those provided by the `JobScheduler` that it wraps, they are one more step away from the details of implementation, and one abstraction more concise.

Where the `JobScheduler` encodes the difference between a task that repeats periodically and one that runs once in the `Boolean` return from the `onStopJob` method, the `WorkManager` makes it explicit; there are two types of tasks: a `OneTimeWorkRequest` and a `PeriodicWorkRequest`.

Enqueuing a work request always returns a token, a `WorkRequest` that can be used to cancel the task, when it is no longer necessary.

The `WorkManager` also supports the construction of complex task chains: "run this and that in parallel, and run the other when both are done." These task chains might even remind you of the chains we used to transform collections in Chapter 2.

The `WorkManager` is the most fluent and concise way to both guarantee that the necessary tasks are run (even when your application is not visible on the device screen) and to do so in a way that optimizes battery use.

Summary

In this chapter we introduced Android's threading model, and some concepts and tools to help you use it effectively. To summarize:

- A thread-safe program is one that behaves, no matter how concurrent threads execute it, in a way that could be reproduced if the same threads executed it serially.
- In the Android threading model, the UI thread is responsible for the following:
  - Drawing the view
  - Dispatching events resulting from user interaction with the UI
- Android programs use multiple threads in order to ensure that the UI thread is free to redraw the screen without dropping frames.
- Java and Android provide several language-level threading primitives:
  - A `Looper`/`Handler` is a queue of tasks serviced by a single, dedicated thread.
  - `Executor`s and `ExecutionService`s are Java constructs for implementing an application-wide thread-management policy.
- Android offers the architectural components `JobScheduler` and `WorkManager` to schedule tasks efficiently.

The following chapters will turn to more complex topics in Android and concurrency. In them we will explore how Kotlin makes managing concurrent processes clearer and easier and less error-prone.

[1] It is possible for processes to share some memory (as with Binder), but they do so in very controlled ways.

[2] Goetz et al., 2006. *Java Concurrency in Practice*. Boston: Addison-Wesley.