Chapter 3. Android Fundamentals

The first two chapters of this book were a whirlwind review of the Kotlin language. This chapter will review the environment in which we will use Kotlin: Android.

Android is an operating system, like Windows and MacOS. Unlike those two systems, Android is a Linux-based OS, like Ubuntu and Red Hat. Unlike Ubuntu and Red Hat, though, Android has been very heavily optimized for mobile devices—battery-powered mobile devices, in particular.

The most significant of these optimizations concerns what it means to be an application. In particular, as we will see, Android apps have much more in common with web applications than they do with familiar desktop applications.

But we'll get to that in a moment. First, let's look in a little more detail at the Android environment. We'll look at the operating system as a stack—kind of a layer cake.

The Android Stack

Figure 3-1 shows one way of looking at Android: as a stack of components. Each layer in the stack has a specific task and provides specific services; each uses the features of the layers beneath it.

Walking up from the bottom, the layers are:

- Hardware
- Kernel
- System services
- Android Runtime Environment
- Applications

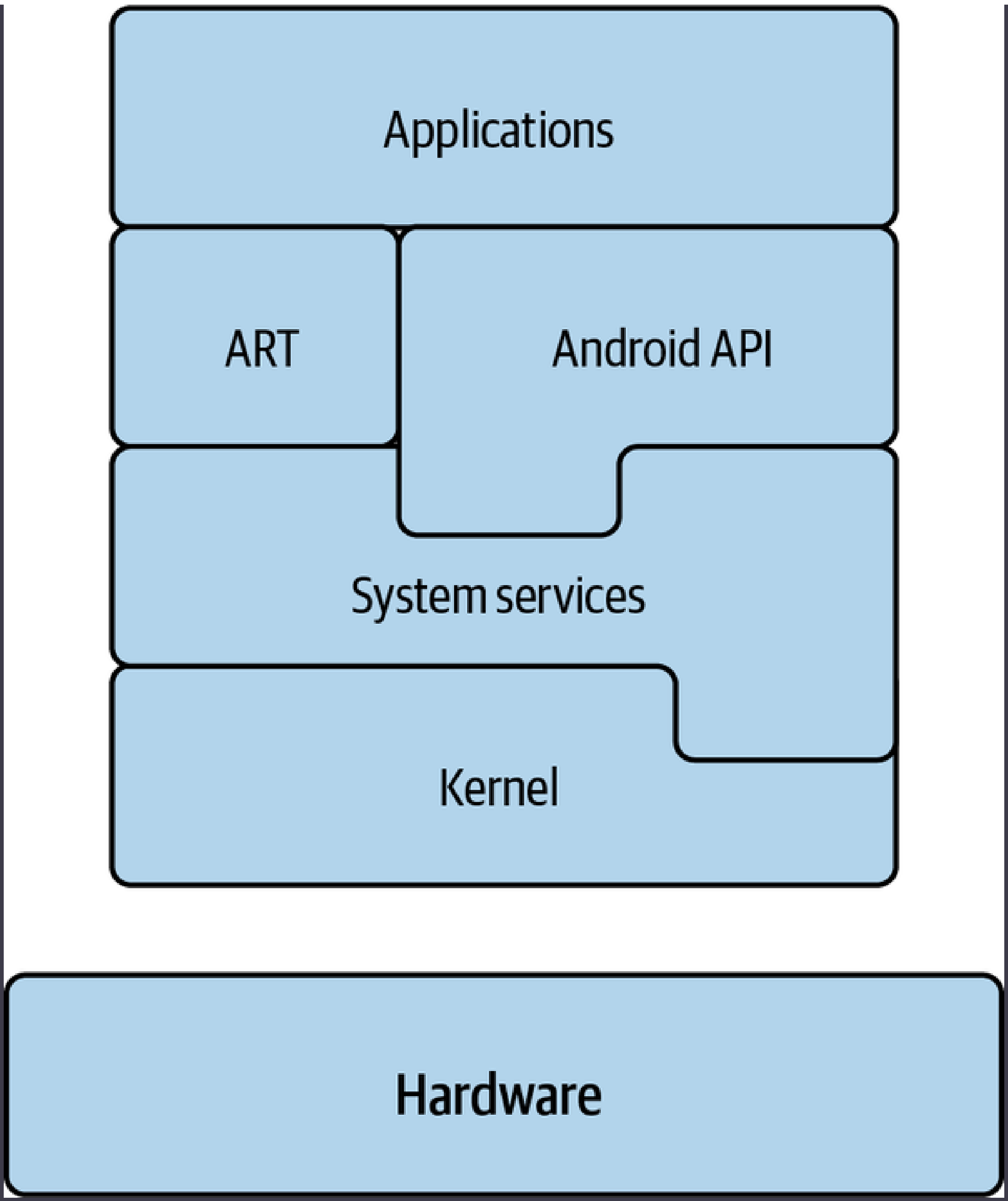

Figure 3-1. The Android stack.

Hardware

Beneath the Android stack, of course, is hardware: a piece of warm silicon. While the hardware is not part of the Android stack, it is important to recognize that the hardware for which Android was designed imposes some fairly tough constraints on the system. By far, the most significant of these constraints is power. Most common operating systems just assume an infinite power supply. The Android systems cannot.

Kernel

The Android operating system depends on the Linux kernel. A kernel is responsible for providing the basic services that developers expect: a filesystem, threads and processes, network access, interfaces to hardware devices, and so on. Linux is free and open source and, thus, a popular choice for hardware and device manufacturers.

Because it is based on Linux, Android bears some similarity to the common Linux distributions: Debian, Centos, etc. In the layers above the kernel, however, the similarity diminishes. While most common Linux distributions are based heavily on the GNU family of system software (and should, properly, be called GNU/Linux), Android's system software is quite a bit different. It is, in general, not possible to run common Linux applications directly on an Android system.

System Services

The system services layer is big and complex. It includes a wide variety of utilities, from code that runs as part of the kernel (drivers or kernel modules), and long-running applications that manage various housekeeping tasks (daemons), to libraries that implement standard functions like cryptography and media presentation.

This layer includes several system services that are unique to Android. Among them are Binder, Android's essential interprocess communication system; ART, which has replaced Dalvik as Android's analog of the Java VM; and Zygote, Android's application container.

Android Runtime Environment

The layer above the system services is the implementation of the *Android Runtime Environment*. The Android Runtime Environment is the collection of libraries that you use from your application by including them with `import` statements: *android.view*, *android.os*, and so on. They are the services provided by the layers below, made available to your application. They are interesting because they are implemented using two languages: usually Java and C or C++.

The part of the implementation that your application imports is likely to be written in Java. The Java code, however, uses the *Java Native Interface*

(JNI) to invoke native code, usually written in C or C++. It is the native code that actually interacts with the system services.

Applications

Finally, at the top of the stack are Android applications. Applications, in the Android universe, are actually part of the stack. They are made up of individually addressable components that other applications can “call.” The Dialer, Camera, and Contacts programs are all examples of Android applications that are used as services by other applications.

This is the environment in which an Android application executes. So let’s get back to looking at the anatomy of an application itself.

The Android Application Environment

Android applications are programs translated from a source language (Java or Kotlin) into a transportable intermediate language, DEX. The DEX code is installed on a device and interpreted by the ART VM, when the application is run.

Nearly every developer is familiar with the standard application environment. The operating system creates a “process”—a sort of virtual computer that appears to belong entirely to the application. The system runs the application code in the process, where it appears to have its own memory, its own processors, and so on, completely independent of other applications that might be running on the same device. The application runs until it, itself, decides to stop.

That’s not how Android works. Android doesn’t really think in terms of applications. For instance, Android apps don’t contain the equivalent of Java’s `public static void main` method, the method used to start typical Java applications. Instead, Android apps are libraries of components. The Android runtime, Zygote, manages processes, lifecycles, and so on. It calls an application’s components only when it needs them. This makes Android apps, as hinted earlier, very similar to web applications: they are an assemblage of components deployed into a container.

The other end of the lifecycle, terminating an application, is perhaps even more interesting. On other operating systems, abruptly stopping an application (`kill -9` or “Force Quit”) is something that happens rarely and only when the application misbehaves. On Android, it is the most common way for an application to be terminated. Nearly every running app will eventually be terminated abruptly.

As with most web app frameworks, components are implemented as subclasses of template base classes. Component subclasses override the methods that are called by the framework in order to provide application-specific behavior. Often, the superclass has important work to do when one of these template methods is called. In those cases, the overriding method in the subclass must call the superclass method that it overrides.

Android supports four types of components:

- Activity
- Service
- Broadcast receiver
- Content provider

Just as in a web app, the implementations of these components must be registered in a manifest: an XML file. Android’s manifest file is called, perhaps unsurprisingly, *AndroidManifest.xml*. The Android container parses this file as part of loading an application. The application components (not some overarching application) are the basic units of the Android app. They are individually addressable and may be published individually for use by other applications.

So, how does an application target an Android component? With an `Intent`.

Intents and Intent Filters

In Android, components are started with `Intent`s. An `Intent` is a small packet that names the component that it targets. It has some extra room in which it can indicate a specific action that it would like the receiving component to take and a few parameters to the request. One can think of an intent as a function call: the name of the class, the name of a particular function within that class, and the parameters to the call. The intent is delivered by the system to the target component. It is up to the component to perform the requested service.

It is interesting to note that, in keeping with its component-oriented architecture, Android doesn't actually have any way of starting an application. Instead, clients start a component, perhaps the `Activity` that is registered as main for an application whose icon a user just tapped on the Launcher page. If the application that owns the activity is not already running, it will be started as a side effect.

An intent can name its target explicitly, as shown here:

```
context.startActivity(
  Intent(context, MembersListActivity::class.java)))
```

This code fires an `Intent` at the `Activity` `MembersListActivity`. Note that the call, `startActivity` here, must agree with the type of the component being started: an `Activity` in this case. There are other, similar methods for firing intents at other kinds of components (`startService` for a `Service`, and so on).

The `Intent` fired by this line of code is called an *explicit intent* because it names a specific, unique class, in a unique application (identified by a `Context`, discussed in a moment), to which the `Intent` is to be delivered.

Because they identify a unique, specific target, explicit intents are faster and more secure than implicit ones. There are places that the Android system, for reasons related to security, requires the use of explicit intents. Even when they are not required, explicit intents should be preferred whenever possible.

Within an application, a component can always be reached with an explicit intent. A component from another application that is publicly visible can also always be reached explicitly. So why ever use an implicit intent? Because implicit intents allow dynamic resolution of a request.

Imagine that the email application you've had on your phone for years allows editing messages with an external editor. We now can guess that it does this by firing an intent that might look something like this:

```
val intent = Intent(Intent.ACTION_EDIT))
intent.setDataAndType(textToEditUri, textMimeType);
startActivityForResult(intent, reqId);
```

The target specified in this intention is *not* explicit. The `Intent` specifies neither a `Context` nor the fully qualified name of a component within a context. The intent is *implicit* and Android will allow any component at all to register to handle it.

Components register for implicit intents using an `IntentFilter`. In fact, the "Awesome Code Editor" that you happen to have installed just 15 minutes ago registers for exactly the intent shown in the preceding code, by including an `IntentFilter` like this in its manifest:

```
<manifest ...>
  <application
    android:label="@string/awesome_code_editor">
    ...>
    <activity
      android:name=".EditorActivity"
      android:label="@string/editor">
      <intent-filter>
        <action
          android:name="android.intent.action.EDIT" />
        <category
          android:name="android.intent.category.TEXT" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

As you can see, the intent filter matches the intent that the email application fires.

When Android installs the Awesome Code Editor application it parses the application manifest and notices that the `EditorActivity` claims to be able to handle an `EDIT` action for the category `android.intent.category.TEXT` (see more in the Android Developers documentation). It remembers that fact.

The next time your email program requests an editor, Android will include Awesome Code Editor in the list of editors it offers for your use. You have just upgraded your email program simply by installing another application. Talk about awesome!

NOTE

Android gradually increased restrictions on the use of implicit intents in recent releases. Because they can be intercepted by any randomly installed application, despite their power, implicit intents are not secure. Recent versions of Android have imposed strict new constraints on their use. In particular, as of v30, it is not possible to register for many implicit intents in the manifest.

Context

Because Android components are just subsystems run in a larger container, they need some way of referring to the container so that they can request services from it. From within a component, the container is visible as a `Context`. `Context`s come in a couple of flavors: component and application. Let's have a look at each of them.

Component context

We've already seen a call like this:

```
context.startActivity(
  Intent(context, MembersListActivity::class.java)))
```

This call uses a `Context` twice. First, starting an `Activity` is a function that a component requests from the framework, the `Context`. In this case, it called the `Context` method `startActivity`. Next, in order to make the intent explicit, the component must identify the unique package that contains the component it wants to start. The `Intent`'s constructor uses the `context` passed as its first argument to get a unique name for the application to which the `context` belongs: this call starts an `Activity` that belongs to this application.

The `Context` is an abstract class that provides access to various resources, including:

- Starting other components
- Accessing system services
- Accessing `SharedPreferences`, resources, and files

Two of the Android components, `Activity` and `Service`, are themselves `Context`s. In addition to being `Context`s, they are also components that the Android container expects to manage. This can lead to problems, all of which are variations on the code shown in Example 3-1.

Example 3-1. Do NOT do this!

```
class MainActivity : AppCompatActivity() {
  companion object {
    var context: Context? = null;
  }

  override fun onCreate() {
    if (context == null) {
      context = this  // NO!
    }
  }
  // ...
}
```

Our developer has decided that it would be really handy to be able to say things like `MainActivity.context.startActivity(...)` anywhere in their application. In order to do that, they've stored a reference to an `Activity` in a global variable, where it will be accessible for the entire life of the application. What could go wrong?

There are two things that could go wrong, one bad and the other horrible. Bad is when the Android framework knows that the `Activity` is no longer needed, and would like to free it up for garbage collection, but it cannot do so. The reference in that companion object will prevent the `Activity` from being released, for the entire lifetime of the application.

The `Activity` has been leaked. `Activity`s are large objects and leaking their memory is no small matter.

The second (far worse) thing, that could go wrong is that a call to a method on the cached `Activity` could fail catastrophically. As we will explain shortly, once the framework decides that an `Activity` is no longer being used, it discards it. It is done with it and will never use it again. As a result, the object may be put into an inconsistent state. Calling methods on it may lead to failures that are both difficult to diagnose and reproduce.

While the problem in that bit of code is pretty easy to see, there are variants that are much more subtle. The following code may have a similar problem:

```
override fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  // ...
  NetController.refresh(this::update)
}
```

It is harder to see, but the callback `this::update` is a reference to a method on `this`, the `Activity` that contains this `onCreate` method. Once `onCreate` completes, the `NetController` holds a reference to this `Activity` that does not honor its lifecycle and can incur either of the problems described earlier.

Application context

There is another kind of context. When Android starts an application, it usually creates a singleton instance of the `Application` class. That instance is a `Context` and, though it has a lifecycle, that lifecycle is essentially congruent with the lifecycle of the application. Because it is long-lived, it is quite safe to hold references to it in other long-lived places. This code, similar to the dangerous code shown earlier, is fairly safe because the `context` to which it stores a reference is the `ApplicationContext`:

```
class SafeApp : Application() {
  companion object {
    var context: Context? = null;
  }

  override fun onCreate() {
    if (context == null) {
      context = this
    }
  }
  // ...
}
```

Be sure to remember that, in order for the Android system to use the custom subclass of `Application` instead of its default, the `SafeApp` class must be registered in the manifest, like this:

```
<manifest ...>
  <application
    android:name=".SafeApp"
    ...>
    ...
  </application>
</manifest>
```

Now, when the framework creates the `ApplicationContext` it will be an instance of `SafeApp` instead of the instance of `Application` that it would have used otherwise.

There is another way to get the `ApplicationContext` as well. Calling the method `Context.getApplicationContext()` on any context at all, including the `ApplicationContext` itself, will always return the long-lived application context. But here's the bad news: the `ApplicationContext` is not a magic bullet. An `ApplicationContext` is not an `Activity`. Its implementations of `Context` methods behave differently from those of `Activity`. For instance, and probably most annoying, you cannot launch `Activity` from an `ApplicationContext`. There is a `startActivity` method on `ApplicationContext`, but it simply generates an error message in all but a very limited set of circumstances.

Android Application Components: The Building Blocks

Finally, we can narrow our focus to the components themselves, the essence of an application.

The lifecycles of Android application components are managed by the Android framework, which creates and destroys them according to its needs. Note that this absolutely includes instantiation! Application code must *never* create a new instance of a component.

Recall that there are four types of components:

- Activity
- Service
- Broadcast receiver
- Content provider

Remember, also, that the following descriptions are nothing more than brief overviews, perhaps calling attention to potential pitfalls or features of interest. The Android Developers documentation is extensive, complete, and authoritative.

The Activity and Its Friends

An `Activity` component manages a single page of an application's UI. It is Android's analog of a web application servlet. It uses Android's rich library of "widgets" to draw a single, interactive page. Widgets (buttons, text boxes, and the like) are the basic UI elements, and they combine a screen representation with the input collection that gives the widgets behavior. We'll discuss them in detail shortly.

As mentioned previously, it is important to understand that an `Activity` is not an application! Activities are ephemeral and guaranteed to exist only while the page that they manage is visible. When that page becomes invisible, either because the application presents a different page or because the user, for instance, takes a phone call, there is no guarantee that Android will preserve either the `Activity` instance or any of the state that it represents.

Figure 3-2 shows the state machine that controls the lifecycle of an `Activity`. The methods—shown as state transitions—come in pairs and are the bookends of the four states that an `Activity` may assume: *destroyed*, *created*, *started*, and *running*. The methods are called strictly in order. After a call to `onStart`, for instance, Android will make only one of two possible calls: `onResume`, to enter the next state, or `onStop`, to revert to the previous state.

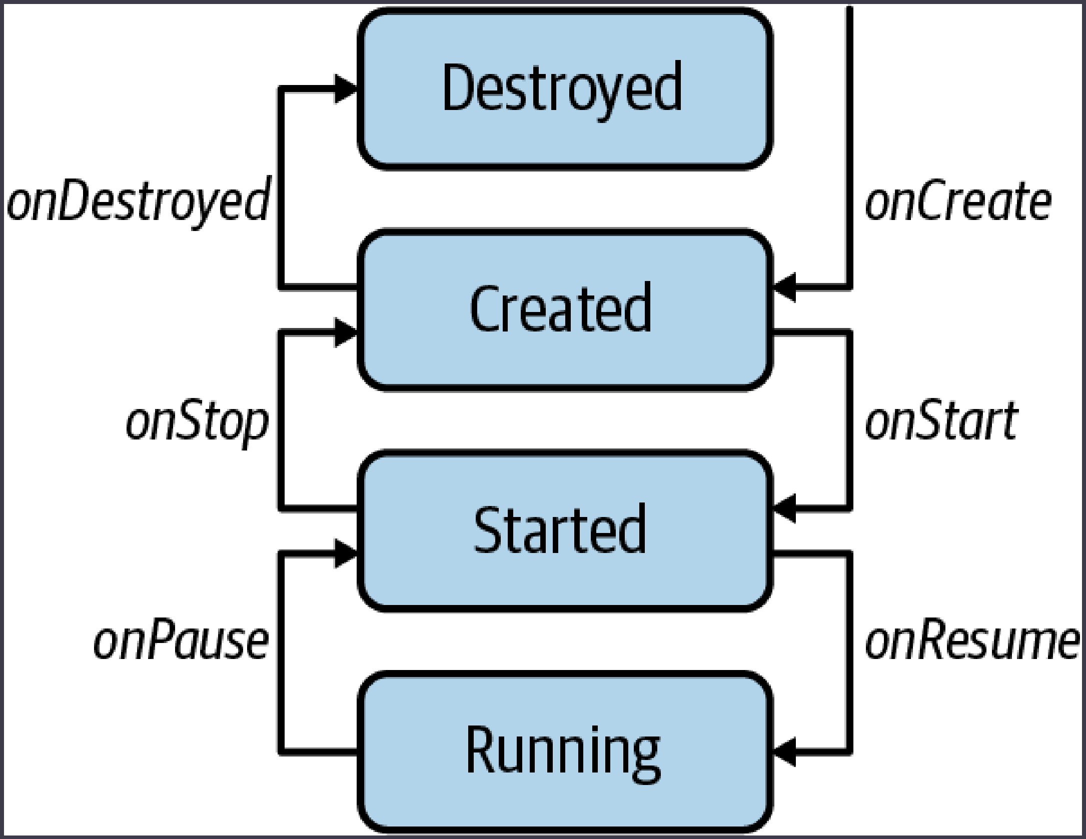

Figure 3-2. The `Activity` lifecycle.

The first pair of bookends are `onCreate` and `onDestroy`. Between them, an `Activity` is said to be *created*. When Android instantiates a new `Activity`, it calls its `onCreate` method nearly immediately. Until it does so, the `Activity` is in an inconsistent state and most of its functions will not work. Note, in particular, that most of an `Activity`'s functionality is, inconveniently, not available from its constructor.

The `onCreate` method is the ideal place to do any initialization that an `Activity` needs to do only once. This almost always includes setting up the view hierarchy (usually by inflating an XML layout), installing view controllers or presenters, and wiring up text and touch listeners.

`Activity`s, similarly, should not be used after the call to their `onDestroy` method. The `Activity` is, again, in an inconsistent state and the Android framework will make no further use of it. (It will not, for instance, call `onCreate` to revivify it.) Beware, though: the `onDestroy` method is not necessarily the best place to perform essential finalization! Android calls `onDestroy` only on a best-effort basis. It is entirely possible that an application will be terminated before all of an `Activity`s'. `onDestroy` methods have completed.

An `Activity` can be destroyed from within its own program by calling its `finish()` method.

The next pair of methods are `onStart` and `onStop`. The former, `onStart`, will only ever be called on an `Activity` that is in the created state. It moves the `Activity` to its on-deck state, called *started*. A started `Activity` may be partially visible behind a dialog or another app that only incompletely fills the screen. In started state, an `Activity` should be completely painted but should not expect user input. A well-written `Activity` will not run animations or other resource-hogging tasks while it is in the started state.

The `onStop` method will only be called on a started `Activity`. It returns it to the created state.

The final pair of methods are `onResume` and `onPause`. Between them, an `Activity`'s page is in focus on the device and the target of user input. It is said to be *running*. Again, these methods will only be called on an `Activity` that is in the started or running state, respectively.

Along with `onCreate`, `onResume` and `onPause` are the most important in the lifecycle of an `Activity`. They are where the page comes to life, starting, say, data updates, animations, and all of the other things that make a UI feel responsive.

TIP

It is a good practice to respect the pairing of these methods: a beginning method and an end method. If you start something running in the beginning method of the pair, stop it in the end method of the same pair. Trying to start, say, network polling in `onResume` and stop it in `onStop` is a recipe for hard-to-find bugs.

Fragments

`Fragment`s are an afterthought added to Android's stable of component-like features only at version 3 (Honeycomb, 2011). They can feel a bit "bolted on." They were introduced as a way of making it possible to share UI implementations across screens with shapes and sizes so different that it affects navigation: in particular, phones and tablets.

`Fragment`s are not `Context`s. Though they hold a reference to an underlying `Activity` for most of their lifecycle, `Fragment`s are not registered in the manifest. They are instantiated in application code and cannot be started with `Intent`s. They are also quite complex. Compare Figure 3-3, the state diagram for a `Fragment`, to that of an `Activity`!

A thorough discussion of how (or, for that matter, even whether) to use `Fragment`s is well outside the scope of this book. Briefly, however, one might think of a `Fragment` as something like an *iframe* on a web page: almost an `Activity` embedded in an `Activity`. They are complete, logical UI units that can be assembled in different ways to form a page.

As shown, `Fragment`s have lifecycles that are similar to (though more complex than) those of an `Activity`. However, a `Fragment` is only useful when it is attached to an `Activity`. This is the main reason that a `Fragment` lifecycle is more complex: its state can be affected by changes in the state of the `Activity` to which it is attached.

Also, just as an `Activity` is programmatically accessible in the inconsistent state before its `onCreate` method is called, so a `Fragment` is programmatically accessible before it is attached to an `Activity`. `Fragment`s must be used with great care before their `onAttach` and `onCreateView` methods have been called.

Figure 3-3. `Fragment` lifecycle.

The back stack

Android supports a navigation paradigm sometimes called *card-deck* navigation. Navigating to a new page stacks that page on top of the previous page. When a user presses a back button the current page is popped from the stack to reveal the one that previously held the screen. This paradigm is fairly intuitive for most human users: push new cards on top; pop them off to get back to where you were.

In Figure 3-4, the current `Activity` is the one named SecondActivity. Pushing the back button will cause the `Activity` named MainActivity to take the screen.

Note that, unlike a web browser, Android does not support *forward* navigation. Once the user pushes the back button, there is no simple navigational device that will allow them to return to the popped page. Android uses this fact to infer that it can destroy SecondActivity (in this case), should it need the resources.

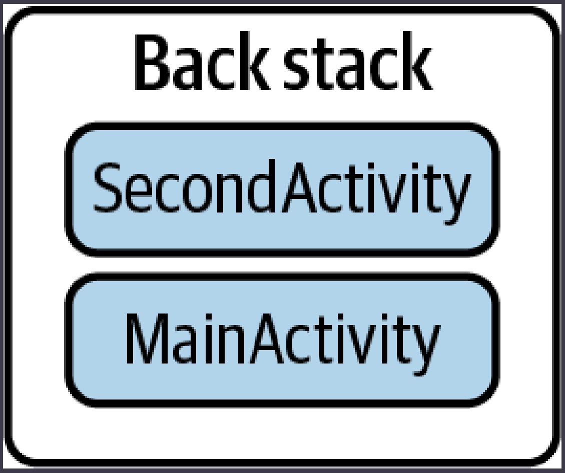

Figure 3-4. The back stack stores an `Activity`'s pages in last in, first out (LIFO) order.

`Fragment`s can also go on the back stack as part of a fragment transaction, as shown in Figure 3-5.

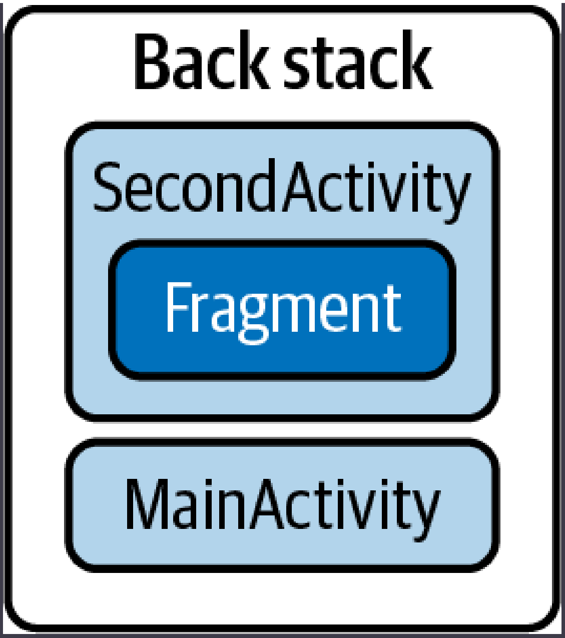

Figure 3-5. A `Fragment` transaction, on the back stack, will be reverted before the `Activity` that contains it is popped.

Adding a fragment to the back stack can be particularly useful when combined with tagging, as shown in the following code:

```
// Add the new tab fragment
supportFragmentManager.beginTransaction()
    .replace(
        R.id.fragment_container,
        SomeFragment.newInstance())
    .addToBackStack(FRAGMENT_TAG)
    .commit()
```

This code creates a new instance of `SomeFragment` and adds it to the back stack, tagged with the identifier `FRAGMENT_TAG` (a string constant). As shown in the following code, you can use `supportFragmentManager` to pop *everything* off the back stack, all the way to the tag:

```
manager.popBackStack(
    FRAGMENT_TAG,
    FragmentManager.POP_BACK_STACK_INCLUSIVE)
```

When the back stack is empty, pushing the back button returns the user to the Launcher.

Services

A `Service` is an Android component that is, almost exactly, an `Activity` with no UI. That may sound a bit odd, given that an `Activity`'s sole reason for existence is that it manages the UI!

Android was designed for hardware that is much different from that which is common now. The first Android phone, the HTC Dream, was announced in September of 2008. It had very little physical memory (192 MB) and did not support virtual memory at all. It could run no more than a handful of applications simultaneously. Android's designers needed a way to know when an application was not doing useful work so that they could reclaim its memory for other uses.

It's easy to figure out when an `Activity` is not doing useful work. It has only one job: to manage a visible page. If applications were composed only of `Activity`s, it would be easy to tell when one was no longer useful and could be terminated. When none of an application's `Activity`s are visible, the application is not doing anything useful and can be reclaimed. It's that simple.

The problem comes when an application needs to perform long-running tasks that are not attached to any UI: monitoring location, synchronizing a dataset over the network, and so on. While Android is definitely prejudiced toward "if the user can't see it, why do it?" it grudgingly acknowledges the existence of long-running tasks and invented `Service`s to handle them.

While `Service`s still have their uses, much of the work that they were designed to do, back on earlier versions of Android with its more limited hardware, can now be done using other techniques. Android's `WorkManager` is a terrific way to manage repeating tasks. There are also other, simpler and more maintainable ways of running tasks in the background, on a worker thread. Something as simple as a singleton class may be sufficient.

Service components still exist, though, and still have important roles to play.

There are, actually, two different kinds of `Service`: *bound* and *started.* Despite the fact that the `Service` base class is, confusingly, the template for both, the two types are completely orthogonal. A single `Service` can be either or both.

Both types of `Service` have `onCreate` and `onDestroy` methods that behave exactly as they do for an `Activity`. Since a `Service` has no UI, it does not need any of an `Activity`'s other templated methods.

Services do have other templated methods, though. Which of them a specific `Service` implements depends on whether it is started or bound.

Started Services

A *started* `Service` is initiated by sending it an `Intent`. While it is possible to create a started service that returns a value, doing so is inelegantly complex and probably indicative of a design that could be improved. For the most part, started services are fire-and-forget: something like "put this in the database" or "send this out to the net."

To start a service, send it an intent. The intent must name the service, probably explicitly by passing the current context and the service class. If the service provides multiple functions, of course, the intent may also indicate which of them it is intended to invoke. It might also supply parameters appropriate for the call.

The service receives the intent as the argument to a call from the Android framework, to the method `Service.onStart`. Note that this is not done in the "background"! The `onStart` method runs on the main/UI thread. The `onStart` method parses the `Intent` content and processes the contained request appropriately.

A well-behaved started `Service` will call `Service.stopSelf()` whenever it completes its work. This call is similar to `Activity.finish()`: it lets the framework know that the `Service` instance is no longer performing useful work and can be reclaimed. Modern versions of Android actually pay very little attention to whether a service has stopped itself or not. `Service`s are suspended and, possibly even terminated, using less voluntary criteria (see the Android Developers documentation).

Bound Services

A *bound* `Service` is Android's IPC mechanism. Bound services provide a communication channel between a client and a server that is process agnostic: the two ends may or may not be part of the same application. Bound services—or at least the communication channels they provide—are at the very heart of Android. They are the mechanism through which applications send tasks to system services.

A bound service, itself, actually does very little. It is just the factory for a `Binder`, a half-duplex IPC channel. While a complete description of the Binder IPC channels and their use is beyond the scope of this book, their structure will be familiar to users of any of the other common IPC mechanisms. Figure 3-6 illustrates the system.

Typically, a service provides a *proxy* that looks like a simple function call. The proxy *marshals* an identifier for the requested service (essentially, the function name) and its parameters, by converting them to data that can be transmitted over the connection: usually aggregates of very simple data types like integers and strings. The marshaled data is communicated, in this case via the Binder kernel module, to a *stub* provided by the bound service that is the target of the connection.

Figure 3-6. Binder IPC.

The stub *unmarshals* the data, converting it back into a function call to the service implementation. Notice that the proxy function and the service implementation function have the same signature: they implement the same interface (IService, as shown in Figure 3-6).

Android makes *extensive* use of this mechanism in the implementation of system services. Functions that are actually calls to remote processes are a fundamental part of Android.

An instance of the class `ServiceConnection` represents a connection to a bound service. The following code demonstrates its use:

```
abstract class BoundService<T : Service> : ServiceConnection {
    abstract class LocalBinder<out T : Service> : Binder() {
        abstract val service: T?
    }
    private var service: T? = null

    protected abstract val intent: Intent?

    fun bind(ctxt: Context) {
        ctxt.bindService(intent, this, Context.BIND_AUTO_CREATE)
    }

    fun unbind(ctxt: Context) {
        service = null
        ctxt.unbindService(this)
    }

    override fun onServiceConnected(name: ComponentName, binder: IBinder)
        service = (binder as? LocalBinder<T>)?.service
        Log.d("BS", "bound: ${service}")
    }

    override fun onServiceDisconnected(name: ComponentName) {
        service = null
    }
}
```

A subclass of `BoundService` provides the type of the service that will be bound, and an `Intent` that targets it.

The client side initiates a connection using the `bind` call. In response, the framework initiates a connection to the remote bound service object. The remote framework calls the bound service's `onBind` method with the intent. The bound service creates and returns an implementation of `IBinder` that is also an implementation of the interface the client requested. Note that this is often a reference to the bound service itself. In other words, the `Service` is often not only the factory but also the implementation.

The service side uses the implementation provided by the bound service to create the remote-side stub. It then notifies the client side that it's ready. The client-side framework creates the proxy and then finally calls the `ServiceConnection`'s `onServiceConnected` method. The client now holds a live connection to the remote service. Profit!

As one might guess from the presence of an `onServiceDisconnected` method, a client can lose the connection to a bound service at any time. Though the notification is usually immediate, it is definitely possible for a client call to a service to fail even before it receives a disconnect notification.

Like a started service, bound service code does not run in the background. Unless explicitly made to do otherwise, bound service code runs on the application's main thread. This can be confusing, though, because a bound service might run on the main thread of a *different* application.

If the code in a service implementation must run on a background thread, it is the service implementation that is responsible for arranging that. Client calls to a bound service, while asynchronous, cannot control the thread on which the service itself runs.

Services, like every other component, must be registered in the application manifest:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
  <application...>
    <service android:name=".PollService"/>
  </application>
</manifest>
```

Content Providers

A `ContentProvider` is a REST-like interface to data held by an application. Because it is an API, not simply direct access to data, a `ContentProvider` can exercise very fine-grained control over what it publishes and to whom it publishes it. External applications get access to a `ContentProvider` using a Binder IPC interface, through which the `ContentProvider` can obtain information about the querying process, the permissions it holds, and the type of access it requests.

Early Android applications often shared data simply by putting it into publicly accessible files. Even then, Android encouraged the use of `ContentProvider`s instead. More recent versions of Android, in the interests of security, have made it difficult to share files directly, making `ContentProvider`s more relevant.

NOTE

While `ContentProvider`s provide access to stored data, you must have some kind of data store from which to read and write the data. Android Jetpack offers the Room persistence library as an option. As described in its official documentation, Room provides "an abstraction layer to allow for more robust access while harnessing the full power of SQLite."

For more information on how to save data in a local database using Room, check out the Android Developers documentation.

One particularly interesting capability of a `ContentProvider` is that it can pass an open file to another program. The requesting program need not have any way to access the file directly using a file path. The `ContentProvider` can construct the file it passes in any way that it wants. By passing an open file, though, the `ContentProvider` moves itself out of the loop. It gives the requesting program direct access to the data. Neither the `ContentProvider` nor any other IPC mechanism remains between the client and the data. The client simply reads the file just as if it had opened that file itself.

An application publishes a `ContentProvider`, as usual, by declaring it in the application manifest:

```
<application...>
  <provider
   android:name="com.oreilly.kotlin.example.MemberProvider"
   android:authorities="com.oreilly.kotlin.example.members"
   android:readPermission="com.oreilly.kotlin.example.members.READ"/>
 </application>
```

This XML element says that the application contains the class named `com.oreilly.kotlin.example.MemberProvider`, which has to be a subclass of `android.content.ContentProvider`. The element declares that `MemberProvider` is the authority for any requests for data from the URL *content://com.oreilly.kotlin.example.members*. Finally, the declaration mandates that requesting applications must hold the permission “com.oreilly.kotlin.example.members.READ” in order to get any access at all and that even then they will get only read access.

`ContentProvider`s have exactly the API one would expect from a REST interface:

`query()`
: This fetches data from a particular table.

`insert()`
: This inserts a new row within a content provider and returns the content URI.

`update()`
: This updates the fields of an existing row and returns the number of rows updated.

`delete()`
: This deletes existing rows and returns the number of rows deleted.

`getType()`
: This returns the MIME data type for the given Content URI.

The `ContentProvider` for `MemberProvider` would probably implement only the first of these methods, because it is read-only.

Broadcast Receivers

The original concept for a `BroadcastReceiver` was as a kind of data bus. Listeners could subscribe in order to get notification of events that were of interest. As the system has come of age, however, `BroadcastReceiver`s have proved to be too expensive and too prone to security problems to be used pervasively. They remain mostly a tool used by the system to signal applications of important events.

Perhaps the most common use of a `BroadcastReceiver` is as a way of starting an application, even if there has been no user request to do so.

The `Intent` `android.intent.action.BOOT_COMPLETED` is broadcast by the Android system once the OS is stable, after a system restart. An application could register to receive this broadcast, like this:

```
<receiver android:name=".StartupReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
```

If an application does this, its `StartupReceiver` will be started, to receive the `BOOT_COMPLETED` `Intent` broadcast when the OS is rebooted. As noted earlier, a side effect of starting the `StartupReceiver` is that the application that contains the receiver is also started.

Applications have used this as a way of creating a *daemon*: an app that is always running. While a hack and fragile (even in early Android, behavior changed from version to version), this trick worked well enough that many, many applications used it. Even as Android version 26 introduced some fairly radical changes in background process management (`BroadcastReceiver`s cannot be registered for implicit broadcasts in their manifests; they must instead register them dynamically using `Context.registerReceiver`), developers continue to find ways to use it.

NOTE

There are exceptions to the Android 26 implicit intent rule. Receiving SMS messaging, changing locale, detecting USB devices, and a few other intents are *permitted*, and applications may register for them, in their manifests. `ACTION_USB_ACCESSORY_ATTACHED`, `ACTION_CONNECTION_STATE_CHANGED`, and our dear old friend `ACTION_BOOT_COMPLETED` are among the permitted intents. For more, check out the Android Developers documentation.

`Activity`, `Service`, `ContentProvider`, and `BroadcastReceiver` are the four components that are the essential building blocks of an Android application. As Android has grown and improved, it has introduced new abstractions that obscure these basic mechanisms. A modern Android application may use only one or two of these building blocks directly, and many developers will never code a `ContentProvider` or a `BroadcastReceiver`.

The essential lesson here, which bears repeating, is that an Android app is not an “application” in the traditional sense. It is more like a web application: a collection of components that provide services to a framework when requested to do so.

Android Application Architectures

So far, in this chapter we’ve discussed the Android system architecture. While understanding that architecture is essential for any serious Android developer, it is not sufficient for understanding how to write resilient, bug-free Android programs. As evidence of this, one need only look at the many tools and abstractions that have been tried and abandoned over the years of Android’s existence. Time and experience, though, have honed the Android playbook and made the path to a robust, maintainable application much easier to follow.

MVC: The Foundation

The original pattern for applications with a UI was called Model–View–Controller (MVC). The innovation that the pattern introduced was a guarantee that the view—what was rendered on the screen—was always consistent. It did this by insisting on a unidirectional cycle for data flow.

It all starts with the user. They see something on the screen (the *View*: I told you it was a cycle!) and, in response to what they see, take some action. They touch the screen, type something, speak, whatever. They do something that will change the state of the application.

Their input is fielded by the *Controller*. The Controller has two responsibilities. First, it orders the user’s input. For any given user event—say, tapping the “stop” button—all other user events happen either before that tap or after it. No two events are ever processed at the same time.

NOTE

The implication that the Controller is single-threaded is one of the most important aspects of the original MVC pattern. Prior multithreaded strategies (including Java's Abstract Window Toolkit [AWT]) often produced a nightmare of deadlocks as messages traveling in opposite directions—from the user and to the user—tried to seize the same locks in different orders.

The Controller's second responsibility is to translate user input into operations on a *Model*.

The Model is the business logic of an application. It probably combines some kind of persistent data store and perhaps a network connection with rules for combining and interpreting the input from the Controller. In the ideal MVC architecture, it is the only component that holds the current state of the application.

The Model, again, ideally is allowed to send only one message to the View: "things have changed." When the View receives such a message it does its job. It requests the application state from the Model, interprets it, and renders it on the screen. What it renders is always a consistent snapshot of the Model. At this point, the user can see the new state and take new actions in response. The cycle continues.

While the MVC pattern was fairly revolutionary when it was introduced, there is room for improvement.

Widgets

As we mentioned earlier in the context of the `Activity` component, a widget is a single class that combines a View component with a Controller component. After the preceding discussion of the MVC pattern and its emphasis on separating the two, it may seem odd to find classes like `Button`, `TextBox`, and `RadioButton` that clearly combine the two.

Widgets do not break MVC architecture. There is still, in each widget, distinct View and Controller code. The Controller portion of a widget never talks directly to the View, and the View does not receive events from the

Controller. The sections are independent; they are just bundled together into a single handy container.

Combining the two functions just seems fairly obvious. What is the use of the image of a button, that can be placed anywhere on the screen, if it doesn't respond to clicks? It just makes sense that the renderer for the UI components, and the mechanism that handles input for it, be part of the same component.

The Local Model

With the advent of the Web, browsers, and the long delay required for an entire MVC cycle, developers began to see the need for keeping the state of the screen as a separate, UI-Local Model. Developers have, over time, referred to this component using several names, depending on other features of the design pattern in which it is being used. To avoid confusion, we will refer to it, for the rest of this chapter, as the *Local Model*.

The use of a Local Model gives rise to a new pattern that is sort of a two-layer MVC—it has even been referred to as the "Figure Eight" pattern. When the user takes an action, the Controller updates the Local Model instead of the Model, because a Model update may be a network connection away. The Local Model is not business logic. It represents, as simply as possible, the state of the View: which buttons are on, which are off, what text is in which box, and the color and length of the bars in the graph.

The Local Model does two things in response to an action. First it notifies the View that things have changed so that the View can rerender the screen from the new Local Model state. In addition, though, with code that is analogous to the simple MVC's Controller, the Local Model forwards the state changes to the Model. In response, the Model eventually notifies—this time the Local Model—that there has been an update and that the Local Model should sync itself. This probably results in a second request that the View update itself.

Android Patterns

In Android, regardless of the pattern, an `Activity` object—or possibly its cousin, a `Fragment`—takes the role of the View. This is more or less mandated by the structure of the `Activity` object: it is the thing that owns the screen and it is the thing that has access to the widgets that comprise the view. Over time, though, as is appropriate for an MVC-based UI, `Activity` objects have gotten simpler and simpler. In a modern Android application, it is likely that an `Activity` will do little more than inflate the view, delegate events inbound from the user to the Local Model, and observe Local Model state that is of interest, redrawing itself in response to updates.

Model–View–Intent

One of the oldest versions of MVC adopted by the Android community was called Model–View–Intent. The pattern decouples the `Activity` from a Model by using `Intent`s and their payloads. While this structure produces excellent component isolation, it can be quite slow and the code for constructing the `Intents` quite bulky. Although it is still used successfully, newer patterns have largely supplanted it.

Model–View–Presenter

A goal for all of these MVC-based patterns is to loosen the coupling among the three components and to make information flow unidirectionally. In a naive implementation, though, the View and the Local Model each hold a reference to the other. Perhaps the View gets an instance of the Local Model from some sort of factory and then registers with it. Though subtle—and regardless of the apparent direction in which information flows—holding a reference to an object of a specific type is coupling.

Over the past few years, there have been several refinements to the MVC pattern in an attempt to reduce this coupling. While these refinements have often resulted in better code, the distinctions among them, and the very names used to identify them, have not always been clear.

One of the earliest refinements replaces the View and Local Model references to each other with references to interfaces. The pattern is often called Model–View–Presenter (MVP). In implementations of this pattern, the Local Model holds a reference, not to the View `Activity`, but simply to the implementation of some interface. The interface describes the minimal set of operations that the Local Model can expect from its peer. It has, essentially, no knowledge that the View is a View: it sees only operations for updating information.

The View proxies user input events to its Presenter. The Presenter, as described earlier, responds to the events, updating Local Model and Model state as necessary. It then notifies the View that it needs to redraw itself. Because the Presenter knows exactly what changes have taken place, it may be able to request that the View update only affected sections, instead of forcing a redraw of the entire screen.

The most important attribute of this architecture, however, is that the Presenter (this architecture's name for the Local Model) can be unit tested. Tests need only mock the the interface that the View provides to the Presenter to completely isolate it from the View. Extremely thin views and testable Presenters lead to much more robust applications.

But it is possible to do even better than this. The Local Model might hold no references to the View at all!

Model–View–ViewModel

Google, with its introduction of Jetpack, supports an architecture called Model–View–ViewModel (MVVM). Because it's supported, internally, by the modern Android framework, it is the most common and most discussed pattern for modern Android apps.

In the MVVM pattern, as usual, either an `Activity` or a `Fragment` takes the role of the View. The View code will be as simple as it is possible to make it, often contained entirely within the `Activity` or `Fragment` subclass. Perhaps some complex views will need separate classes for image rendering or a `RecyclerView`. Even these, though, will be instantiated and installed in the view, directly by the `Activity` or `Fragment`.

The ViewModel is responsible for wiring together the commands necessary to update the View and the backend Model. The novel feature of this pattern is that a single interface, `Observable`, is used to transmit changes in the state of the Local Model to the View.

Instead of the multiple Presenter interfaces used in the MVP pattern, the ViewModel represents viewable data as a collection of `Observable`s. The View simply registers as an observer for these observables and reacts to notifications of changes in the data they contain.

The Jetpack library calls these `Observable`s `LiveData`. A `LiveData` object is an observable data holder class with a single generified interface that notifies subscribers of changes in the underlying data.

Like MVP, MVVM makes mocking and unit testing easy. The important new feature that MVVM introduces is lifecycle awareness.

The keen reader will have noticed that the version of the MVP pattern described earlier does *exactly* the thing we warned against in Example 3-1: it stores the reference to an `Activity`, an object with an Android-controlled lifecycle, in a long-lived object! Applications are left to their own devices to make sure the reference doesn't outlive the target object.

The Jetpack-supported implementation of the MVVM pattern dramatically reduces this problem. In its implementation, the only references to the View are the subscriptions to the `LiveData` observables. The `LiveData` objects identify `Fragment` and `Activity` observers, and unregister them, automatically when their lifecycle ends.

Applications built with JetPack's version of MVVM can be quite elegant. For a broad variety of applications, the View will contain a single, simple, declarative method that draws the screen. It will register that method as an observer for ViewModel observables. The ViewModel translates user input into calls to the backend Model and updates its observables in response to notifications from the Model. It's that simple.

Summary

Congratulations, you've successfully covered an intimidating amount of information in a very short chapter!

Remember that much of this material is foundational. It is not important that you master all of the information presented here. In fact, it's quite possible that you will never touch, for instance, a `ContentProvider` or a `BroadcastReceiver`. Use what is practical for you, and approach mastering items only as they become useful.

Here are some key points to take with you:

- An Android app is not an “application” in the traditional sense. It is more like a web application: a collection of components that provide services to a framework, when requested to do so.
- The Android OS is a very specialized Linux distribution. Each application is treated as an individual “user” and has its own private file storage.
- Android has four kinds of components. They are: `Activity`s, `Service`s, `ContentProvider`s, and `BroadcastReceiver`. `Activity`s, `Service`s, and the `ContentProvider`s must be registered and possibly given permission within the Android manifest:
  - `Activity`s are the UI of an Android application. They start their lifecycle at `onCreate`, are live to user interaction after `onResume`, and may be interrupted (`onPause`) at any time.
  - `Fragment`s are complex beasts with lifecycles all their own. They can be used to organize independent UI containers, within a UI page.
  - `Service`s can be started services and/or bound. API 26 started introducing restrictions for background use of services, so the general rule is that if the user interacts with a task in any way, a service ought to be made into a foreground service.
  - Unless a `BroadcastReceiver` is using implicit intent that is explicitly allowed by the system with the action, it is probably necessary to register the broadcast receiver dynamically from application code.
- Use the `Activity Context` carefully. Activities have a lifecycle that is not under the control of your application. A reference to an `Activity` *must* respect the actual lifecycle of the Activity.
- General software architectures in Android, like MVI, MVP, and MVVM, are designed to keep `Fragment`s and `Activity`s lean and encourage better separation of concern and testing and while being “lifecycle-aware.”

Now that we’ve reviewed the ground rules and explored the playing field, our journey to achieving structured coroutines in Kotlin officially starts.

In the following chapter, we begin to apply this foundation to examining memory and threading in Android. Understanding the details of Android’s organization will reveal the issues that the coming chapters set out to solve.