Chapter 12. Trimming Down Resource Consumption with Performance Optimizations

In the previous chapter, you became familiar with ways to examine what's going on "under the hood" using popular Android profiling tools. This final chapter highlights a medley of performance optimization considerations. There's no one-size-fits-all approach, so it is helpful to become aware of potential performance pitfalls (and solutions). However, performance issues can sometimes be the result of many compounding problems that individually may not seem noteworthy.

Performance considerations allow you to examine concerns that may impact your application's ability to scale. If you can use any of these strategies as "low-hanging fruit" in your code base, it's well worth going for the biggest win with the smallest amount of effort. Not every section of this chapter will be suitable for every project you work on, but they are still useful considerations to be aware of when writing any Android application. These topics range from view system performance optimizations to network data format, caching, and more.

We are aware that the View system is to be replaced by Jetpack Compose: however, the View system is not going anywhere for years, even with Jetpack. The first half of this chapter is dedicated to view topics every project could benefit from: potential optimizations for the Android View system. The way you set up view hierarchies can end up having a substantial impact on performance if you are not careful. For this reason, we look at two easy ways to optimize view performance: reducing view hierarchy complexity with `ConstraintLayout`, and creating drawable resources for animation/customized backgrounds.

Achieving Flatter View Hierarchy with ConstraintLayout

As a general rule, you want to keep your view hierarchies in Android as flat as possible. Deeply nested hierarchies affect performance, both when a view first inflates and when the user interacts with the screen. When view hierarchies are deeply nested, it can take longer to send instructions back up to the root `ViewGroup` containing all your elements and traverse back down to make changes to particular views.

In addition to the profiling tools mentioned in Chapter 11, Android Studio offers *Layout Inspector*, which analyzes your application at runtime and creates a 3D rendering of the view elements stacked on the screen. You

can open Layout Inspector by clicking the bottom corner tab of Android Studio, as shown in Figure 12-1.

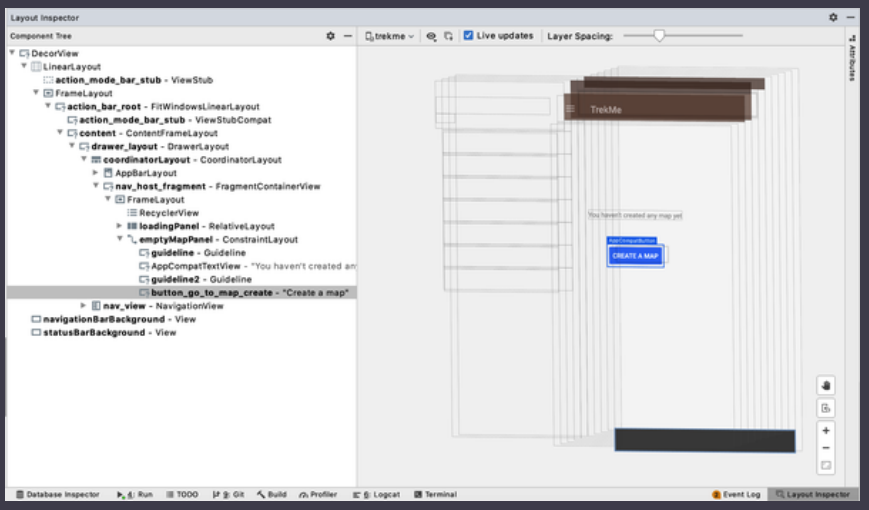

Figure 12-1. Layout Inspector allows you to rotate the 3D rendering for devices running API 29+.

When child components are drawn, they are drawn on top of the parent `View`, stacking one on top of the other. Layout Inspector does provide a *Component Tree* pane to the left so that you are able to drill down the elements and inspect their properties. To better understand what happens when users interact with Android UI widgets, Figure 12-2 shows a bird's-eye view of the very same layout hierarchy provided in the Component Tree.

Even for a relatively simple layout, a view hierarchy can grow in complexity pretty quickly. Managing many nested layouts can come with additional costs such as increased difficulty managing touch events, slower GPU rendering, and difficulty guaranteeing the same spacing/size of views across different-sized screens.

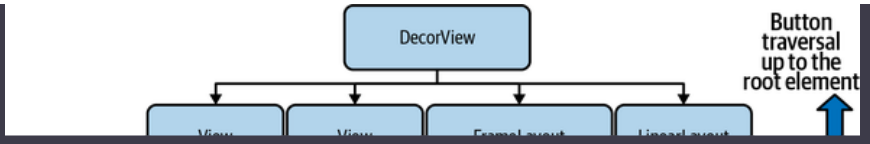

Figure 12-2. The elements of a running activity stretched out in their entirety.

On top of the visual changes your app might call for, the Android OS could also be affecting view properties on its own. Changes on view properties, called by either you or the OS, could trigger a re-layout of your view hierarchy. Whether this happens or not depends on how views are implemented (by yourself or by an external dependency), how often layout components trigger dimension resizing, and where they are located in the view hierarchy.

Not only must we worry about hierarchy complexity, but we also must be mindful of avoiding certain types of views that could end up costing our application twice the number of traversals necessary to send instructions to the Android OS. Some older layout types in Android are prone to "double taxation" when relative positioning is enabled:

`RelativeLayout`

Without fail, this always traverses its child elements at least twice: once for layout calculations for each position and size and once to finalize positioning.

`LinearLayout`

This sets its orientation to horizontal or sets `android:setMeasureWithLargestChildEnabled="true"` while in vertical orientation; both cases make two passes for each child element.

`GridLayout`

This can end up making double traversals if the layout uses weight distribution or sets `android:layout_gravity` to any valid value.

The cost of double taxation can become far more severe when any one of these cases is located closer to the root of the tree, and can even cause exponential traversals. The deeper the view hierarchy is, the longer it takes for input events to be processed and for views to be updated accordingly.

As a good practice, it's best to lower the negative impact of view re-layout on app responsiveness. To keep hierarchies flatter and more robust, Android advocates using `ConstraintLayout`. `ConstraintLayout` helps create a responsive UI for complex layouts with a flat-view hierarchy.

There are a few rules of `ConstraintLayout` to remember:

- Every view must have at least one horizontal and one vertical constraint.
- The Start/End of a view may only chain itself to the Start/End of other views.
- The Top/Bottom of a view may only chain itself to the Top/Bottom of other views.

Android Studio's design preview shows how the parent ties the view to the designated end of the screen, as shown in Figure 12-3.

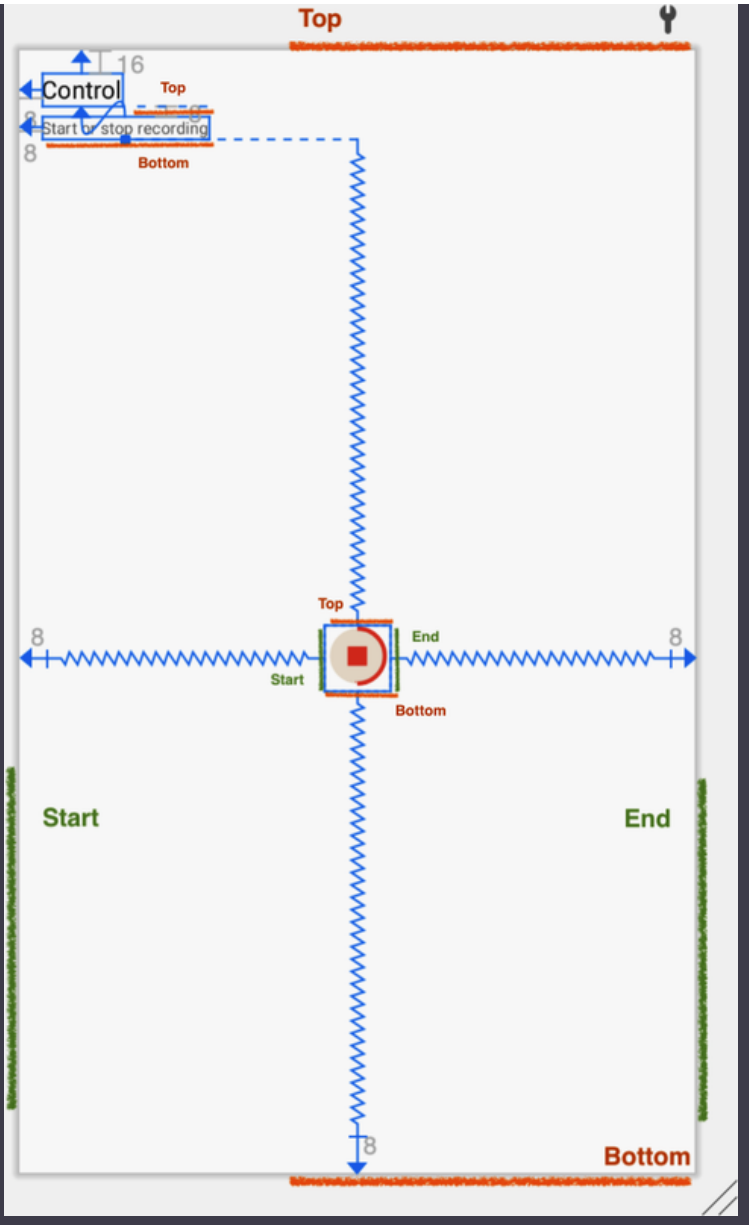

Figure 12-3. In this particular `ConstraintLayout`, the spinner button constrains all parent sides to the center of the screen. The text elements in the upper-left corner are only constrained to the top and left sides of the parent.

When highlighted, the zigzagged lines appear on a view to indicate where a side is constrained to. A zigzag indicates a constraint one way to a view while a squiggly line indicates that the two views constrain to each other.

This book does not cover additional useful features of `ConstraintLayout`, like barriers, guidelines, groups, and creating constraints. The best way to get to know `ConstraintLayout` is to experiment with the elements yourself in *Split View* within the design panel, as shown in Figure 12-4.

Figure 12-4. The Split View of the design panel shows half code and half design for layout files.

Using `ConstraintLayout`, especially when `ViewGroup` elements might be deeply nested or inefficient, is an easy way to address potential perfor-

mance bottlenecks at runtime for any Android application. In the next section, we shift focus on performance optimizations from views themselves to view animations.

Reducing Programmatic Draws with Drawables

Another potential performance issue for any Android project is programmatic draws at runtime. Once in a while, Android developers run into a view element which does not have access to certain properties in a layout file. Suppose you wanted to render a view with rounded corners only on the top two corners. One way to approach this is with a programmatic draw via a Kotlin extension function:

```
fun View.roundCorners(resources: Resources, outline: OutLine?) {
    val adjusted = TypedValue.applyDimension(
        TypedValue.COMPLEX_UNIT_SP,
        25,
        resources?.displayMetrics
    )
    val newHeight =
        view.height.plus(cornerRadiusAdjusted).toInt()
    this.run { outline?.setRoundRect(0, 0, width, newHeight, adjusted)}
}
```

This is fine and valid; however, too many programmatic draws can end up choking the RenderThread and subsequently block the UI thread from being able to process further events until runtime drawings complete. Furthermore, the cost of altering views programmatically becomes higher if a particular view needs to resize to meet constraints. Resizing a view element at runtime means you won't be able to use the `LayoutInflater` to adjust how the elements fit with the new dimensions of the original altered view.

You can offload overhead that would otherwise occur by using drawables, which are stored in the */drawables* folder in your resource assets. The following code shows how a `Drawable` XML file achieves the same goal of rounding the top two corners of a view element:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape = "rectangle">
    <corners android:topLeftRadius="25dp" android:topRightRadius="25dp"/>
    <stroke android:width="1dp" android:color="#FFF"/>
    <solid android:color="#FFF"/>
</shape>
```

You can then add the name of the file as a `Drawable` type to the background attribute in the View's layout file the name of the `Drawable` file:

```
android:background="@drawable/rounded_top_corners_backgr
```

In the previous section, we briefly touched on the initial stages of how user interaction sends instructions to the Android OS. To understand where animations come in, we will now dive a little further into the full process of how Android renders the UI. Let's consider the case where a user in TrekMe presses the "Create a Map" button.

The stages we cover in the remainder of this section show how the OS processes user events with a screen and how it is able to execute draw instructions from software to hardware. We explain all the phases the Android OS performs in a draw up to where animations occur in the *Sync* stage, as shown in Figure 12-5.

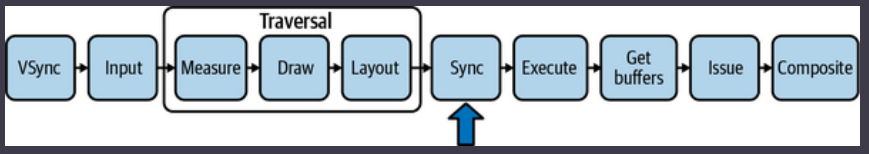

Figure 12-5. Animation occurs at the Sync stage, after traversal is performed.

VSync represents the time given between frame draws on the screen. In an app, when a user touches a view element on the screen, *input handling* occurs. In the *Input* stage, the Android OS makes a call to *invalidate* all the parent view element nodes up the tree by copying a set of instructions to keep track of dirtied state. Invalidation does not redraw the view itself, but rather, indicates to the system later on which marked view must be redrawn later. This is done by propagating the copied information up the view hierarchy so that it can all be executed on the way back down at a later stage. Figure 12-6 shows what invalidation looks like after user input occurs when someone touches a button: traversing up the node, then copying a set of `DisplayList` instructions up each parent view. Even though the arrow points down the elements, indicating child elements, the traversal and the copying of `getDisplayList()` actually goes up to the root before going back down.

Figure 12-6. The `DisplayList` object is a set of compact instructions used to instruct which views need to be redrawn on the Canvas. These instructions are copied up every parent view element to the root hierarchy during invalidation and then executed during traversal.

The Android UI system then schedules the next stage, known as *traversal*, which contains its own subset of rendering stages:

Measure

This calculates `MeasureSpecs` and passes it to the child element for measuring. It does this recursively, all the way down to the leaf nodes.

Layout

This sets the view position and sizing of a child layout.

Draw

> This renders the views using a set of instructions given by a set of `DisplayList` instructions.

In the next stage, *Sync*, the Android OS syncs the `DisplayList` info between the CPU and GPU. When the CPU starts talking to the GPU in Android, the JNI takes its set of instructions in the Java Native layer within the UI thread and sends a synthetic copy, along with some other information, to the GPU from the RenderThread. The RenderThread is responsible for animations and offloading work from the UI thread (instead of having to send the work to the GPU). From there, the CPU and GPU communicate with each other to determine what instructions ought to be executed and then combined visually to render on the screen. Finally, we reach the *Execute* stage, where the OS finally executes `DisplayList` operations in optimized fashion (like drawing similar operations together at once). "Drawn Out: How Android Renders" is an excellent talk that provides more detail on Android rendering at the system level.[1]

As of Android Oreo, animations, such as circular reveals, ripples, and vector drawable animations, live only in the `RenderThread`, meaning that these kinds of animations are nonblocking for the UI thread. You can create these animations with custom drawables. Consider the case where we wish to animate a shadowed ripple in the View background whenever a user presses some kind of `ViewGroup`. You can combine a set of drawables to make this happen, starting with `RippleDrawable` type `Drawable` to create the ripple animation itself:

```
<?xml version="1.0" encoding="utf-8"?>
<ripple xmlns:android="http://schemas.android.com/apk/res/android"
        android:color="@color/primary">
    <item android:id="@android:id/mask">
        <shape android:shape="rectangle">
            <solid android:color="@color/ripple_mask" />
        </shape>
    </item>
</ripple>
```

`RippleDrawable`, whose equivalent on XML is `ripple`, requires a color attribute for ripple effects. To apply this animation to a background, we can use another drawable file:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
        android:shape="rectangle">
    <solid android:color="@color/background_pressed" />
</shape>
```

We can use DrawableStates, a set of framework-provided states that can be specified on a `Drawable`. In this case, we use DrawableStates on a selector to determine the animation as well as whether the animation oc-

curs on press or not. Finally, we create a `Drawable` used to render different states. Each state is represented by a child drawable. In this case, we apply the ripple drawable animation only when the view has been pressed:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android"
    android:enterFadeDuration="@android:integer/config_shortAnimTime"
    android:exitFadeDuration="@android:integer/config_shortAnimTime">
    <item
        android:state_pressed="true" android:state_enabled="true"
        android:drawable="@drawable/background_pressed_ripple"/>
    <item
        android:state_pressed="false"
        android:drawable="@android:color/transparent"/>
</selector>
```

NOTE

As mentioned in the beginning of the chapter, the view system build around Jetpack Compose is completely different from the view system in Android, with its own sets of UI management, graphics, runtime/compile time behavior, and more. If Jetpack Compose is done with programmatic draws, would that mean using Jetpack Compose is not efficient for drawing? While XML currently renders faster than Compose rendering itself, optimizations are underway for closing the gap on render time. However, you should keep in mind the major advantage Compose holds is the ability to update, or recompose, Composable views quickly and far more efficiently than the current Android view framework.

We're done talking about view performance optimizations, and we'll move on to more performance optimization tips around various parts of an Android application for the remainder of the chapter.

Minimizing Asset Payload in Network Calls

In Android, it's important to use minimal payload to avoid slower loads, battery drainage, and using too much data. In the previous chapter, we started looking at network payload data formats. Both images and serialized data formats are the usual suspects for causing the most bloat, so it's important to check your payload's data format.

If you don't need transparency for the images you work with in your Android project, it's better to work with JPG/JPEG since this format intrinsically doesn't support transparency and compresses better than PNG. When it comes to blowing up bitmaps for thumbnails, it probably makes sense to render the image in much lower resolution.

In the industry, JSON is commonly used as the data payload in networking. Unfortunately, JSON and XML payloads are horrible for compression since the data format accounts for spaces, quotes, returns, acmes, and more. Binary serialization formats like *protocol buffers*, an accessible data format in Android which might serve as a cheaper alternative. You can define the data structs, which Protobuf is able to compress much smaller than XML and JSON data. Check out Google Developers for more on protocol buffers.

Bitmap Pooling and Caching

TrekMe uses Bitmap pooling to avoid allocating too many `Bitmap` objects. Bitmap pooling reuses an existing instance, when possible. Where does this "existing instance" come from? After a `Bitmap` is no longer visible, instead of making it available for garbage collection (by just not keeping a reference on it), you can put the no-longer-used `Bitmap` into a "bitmap pool." Such a pool is just a container for available bitmaps for later use. For example, TrekMe uses a simple in-memory dequeue as a bitmap pool. To load an image into an existing bitmap, you have to specify which bitmap instance you want to use. You can do that using the `inBitmap` parameter[2] of `BitMapFactory.Options`:

```
// we get an instance of bitmap from the pool
 BitmapFactory.Options().inBitmap = pool.get()
```

It's worth noting that image-loading libraries like Glide can save you from having to handle bitmap craziness yourself. Using these libraries results in bitmap caching for free in your applications. In cases where network calls are slow, fetching a fresh instance of a `Bitmap` could be costly. This is when fetching from a bitmap cache can save a lot of time and resources. If a user revisits a screen, the screen is able to load almost immediately instead of having to make another network request. We can distinguish two kinds of caches: *in-memory* and *filesystem* caches. In-memory caches provide the fastest object retrieval, at the cost of using more memory. Filesystem caches are typically slower, but they do have a low memory footprint. Some applications rely on in-memory LRU cache,[3] while others use filesystem-based cache or a mix of the two approaches.

As an example, if you perform HTTP requests in your application, you can use *OkHttp* to expose a nice API to use a filesystem cache. *OkHttp* (which is also included as a transitive dependency of the popular library, *Retrofit*) is a popular client library widely used in Android for networking. Adding caching is relatively easy:

```
val cacheSize = 10 * 1024 * 1024
val cache = Cache(rootDir, cacheSize)

val client = OkHttpClient.Builder()
```

```
                .cache(cache)
                .build()
```

With *OkHttp* client building, it is easy to create configurations with custom interceptors to better suit the use case of an application. For example, interceptors can force the cache to refresh at a designated interval. Caching is a great tool for a device working with limited resources in its environment. For this reason, Android developers ought to use cache to keep track of calculated computations.

TIP

A nice open source library that supports both *in-memory* and *filesystem* cache is Dropbox Store.

Reducing Unnecessary Work

For your application to consume resources frugally, you want to avoid leaving in code that is doing unnecessary work. Even senior developers commonly make these kinds of mistakes, causing extra work and memory to be allocated unnecessarily. For example, custom views in Android require particular attention. Let's consider a custom view with a circular shape. For a custom view implementation, you can subclass any kind of `View` and override the `onDraw` method. Here is one possible implementation of `CircleView`:

```
// Warning: this is an example of what NOT to do!
class CircleView @JvmOverloads constructor(
    context: Context,
) : View(context) {

    override fun onDraw(canvas: Canvas) {
       super.onDraw(canvas)
       canvas.save()
       // Never initialize object allocation here!
       val paint: Paint = Paint().apply {
           color = Color.parseColor("#55448AFF")
           isAntiAlias = true
       }
       canvas.drawCircle(100f, 100f, 50f, paint)
       canvas.restore()
   }
}
```

The `onDraw` method is invoked every time the view needs to be redrawn. That can happen quite frequently, especially if the view is animated or moved. Therefore, you should never instantiate new objects in `onDraw`. Such mistakes result in unnecessarily allocating a lot of objects, which puts high pressure on the garbage collector. In the previous example, a

new `Paint` instance is created every time the rendering layer draws `CircleView`. You should never do that.

Instead, it is better to instantiate the `Paint` object once as a class attribute:

```
class CircleView @JvmOverloads constructor(
    context: Context,
) : View(context) {

    private var paint: Paint = Paint().apply {
        color = Color.parseColor("#55448AFF")
        isAntiAlias = true
    }
        set(value) {
            field = value
            invalidate()
        }

    override fun onDraw(canvas: Canvas) {
       super.onDraw(canvas)
       canvas.save()
       canvas.drawCircle(100f, 100f, 50f, paint)
       canvas.restore()
   }
}
```

Now the `paint` object is allocated only once. For the purposes of this existing class, sometimes the `paint` value would be set to different colors. However, if the assignment is not dynamic, you can take it a step further by evaluating the `paint` value lazily.

You want to keep your inject balanced and your dependencies light whenever possible. For repositories, services, and other singleton dependencies (dependencies that are single objects in memory, like `object`), it makes sense to make use of `lazy` delegation so that there is a singleton instance rather than copies of the same object sitting in the heap.

Consider the code we examined earlier in "Detecting Memory Leaks with LeakCanary":

```
class MapImportViewModel @ViewModelInject constructor(
    private val settings: Settings,
    private val app: Application
): ViewModel() {
    /* removed for brevity */

    fun unarchiveAsync(item: ItemData) {
        viewModelScope.launch {
            val inputStream = app.contentResolve.
                openInputStream(item.uri) ?: return@launch
            val rootFolder = settings.getAppDir() ?: return@launch
            val outputFolder = File(rootFolder, "imported")
```

```
                /* removed for brevity */
            }
        }
    }
```

In this class, the `settings` dependency is injected using Hilt—you can tell that by the `@ViewModelInject`. At the time we wrote this example, we were using Hilt 2.30.1-alpha and only dependencies available in the activity scope could be injected into the `ViewModel`. In other words, a newly created `MapImportViewModel` is always injected into the same `Settings` instance, as long as the activity isn't re-created. So the bottom line is: a dependency injection framework such as Hilt can assist you in scoping the lifecycle of your dependencies. In TrekMe, `Settings` is scoped in the application. Therefore, `Settings` is technically a singleton.

NOTE

Hilt is a dependency injection (DI) framework that provides a standard way to use DI in your application. The framework also has the benefit of managing lifecycles automatically, and has extensions available for use with Jetpack components like ViewModels and WorkManager.

The avoidance of unnecessary work expands into every scope of Android development. When drawing objects to render on the UI, it makes sense to recycle already-drawn pixels. Likewise, since we know that making network calls in Android drains the battery, it's good to examine how many calls are made and how frequently they're called. Perhaps you have a shopping cart in your application. It may make good business sense to make updates to the remote server so that a user can access their cart cross-platform. On the other hand, it may also be worth exploring updating a user's cart in local storage (save for a periodic network update). Of course, these kinds of business decisions exist outside the scope of this book, but technical consideration can always help to make for more thoughtful features.

Using Static Functions

When a method or a property isn't tied to any class instance (e.g., doesn't alter an object state), it sometimes makes sense to use *static functions/properties*. We'll show different scenarios where using static functions is more appropriate than using inheritance.

Kotlin makes it very easy to use static functions. A `companion object` within a class declaration holds static constants, properties, and functions that can be referenced anywhere in the project. For example, an Android service can expose a static property `isStarted`, which can only be modified by the service itself, as shown in Example 12-1.

Example 12-1. `GpxRecordingService.isStarted`

```
class GpxRecordingService {

    /* Removed for brevity */

    companion object {
        var isStarted: Boolean = false
            private set(value) {
                EventBus.getDefault().post(GpxRecordServiceStatus(value))
                field = value
            }
    }
}
```

In Example 12-1, `GpxRecordingService` can internally change the value of `isStarted`. While doing so, an event is sent through the event bus, notifying all registered components. Moreover, the status of the `GpxRecordingService` is accessible from anywhere in the app as a read-only `GpxRecordingService.isStarted` property. But remember to avoid accidentally saving an `Activity`, `Fragment`, `View`, or `Context` to a static member: that could end in a hefty memory leak!

Minification and Obfuscation with R8 and ProGuard

It is a common practice to *minify*, or shrink, release builds for production so that unused code and resources can be removed. Minifying your code allows you to ship smaller APKs to Google PlayStore more securely. *Minification* shrinks your code by removing unused methods. Minifying your code also gives you the power of *obfuscation* as an additional security feature. Obfuscation garbles the names of classes/fields/methods and removes debugging attributes in order to discourage reverse engineering.

For Android users, R8 is now the default minification tool provided by the Android Gradle plug-in 5.4.1+. ProGuard, R8's stricter and more powerful predecessor, had a heavier focus on optimizing heavy reflection like the ones found in Gson. In comparison, the newer minification tool R8 does not support this feature. However, R8 is successful in achieving smaller compression and optimization for Kotlin.

Configurations can be done through `proguardFile` (you will see an example at the end of the section). R8 reads the rules provided for the `proguardFile` and executes shrinking and obfuscation accordingly. You can then assign a *proguardFile* to a certain flavor and build type in *build.gradle*:

```
buildTypes {
    release {
```

```
            minifyEnabled true
            shrinkResources true
            proguardFile getDefaultProguardFile('proguard-android-optimize.txt'),'progua
        }
    }
```

It's common practice to shrink your APK to upload to the PlayStore. However, it's important to be watchful and prevent unintentionally shrinking/obfuscating code that might need to be used by a third-party library at runtime. Kotlin uses metadata in Java classes for Kotlin constructs. However, when R8 shrinks Kotlin classes, it is unable to keep state with the Kotlin metadata. In a best-case scenario, shrinking/obfuscating such classes might cause wonky behavior; in a worst-case scenario, it might cause inexplicable crashes.

To demonstrate a scenario where ProGuard accidentally obfuscates too much application code, we observe some wonky behavior on the popular open source library, Retrofit. Perhaps your application works perfectly fine in debugging mode, but in release mode, a networking call inexplicably returns a `NullPointerException`. Unfortunately, Kotlin Gson models go blank even while annotating properties/fields with Retrofit's `@SerializedName`, thanks to Kotlin reflection. As a result, you must add a rule in your proguard file to prevent the Kotlin model class from obfuscating. Oftentimes, you may end up having to include your model classes by adding them directly in your `proguardFile`. Here is an example of adding model domain classes to a `proguardFile` so that release builds don't accidentally obfuscate the aforementioned classes:

```
# Retrofit 2.X
-dontwarn retrofit2.**
-keep class retrofit2.** { *; }
# Kotlin source code whitelisted here
-keep class com.some.kotlin.network.model.** { *; }
-keepattributes Signature
-keepattributes Exceptions
-keepclasseswithmembers class * {
    @retrofit2.http.* <methods>;
}
```

A good piece of advice is: always test the release build!

Summary

This chapter covered the following important performance optimization tips:

- In the Android view framework, deeply nested view hierarchies take longer to draw and traverse than flatter hierarchies. Consider using `ConstraintLayout`, where you can flatten nested views.

- In the Android view framework, it is better to move programmatic draws and animations to drawable resources to offload the work on the RenderThread at runtime.
- Using JSON and XML formats for network data payload is horrible for compression. Use protocol buffers for much smaller data compression.
- Avoid unnecessary work whenever possible: make sure you're not ringing off unnecessary network calls for constant updates, and try to recycle drawn objects.
- Optimizations in performance and memory can come from taking an honest look at the code you write. Are you unintentionally creating objects within a loop that could be created once outside a loop? What expensive operations could be reduced to less-intensive operations?
- You can use a ProGuard file to make your application as small as possible and add custom rules for shrinking, obfuscating, and optimizing your app.

Let's face it: Android can be a challenge to keep up with. It's OK to take information in stride as it becomes relevant for you. Such a strategy guarantees learning opportunities that stay with you for a long time. No matter where you're at in your journey, one of your best resources for both Kotlin and Android (besides this book) is the open source community. Both Android and Kotlin are living, breathing communities from which you can ascertain the newest and most relevant information. To keep yourself current, you can turn to additional resources like Twitter, Slack, and KEEP. You may well also find that you can return to this book to revisit popular, evergreen problems that show up in Android from time to time. We hope you enjoyed this book.

1 Chet Haase and Romain Guy. "Drawn Out: How Android Renders." Google I/O '18, 2017.

2 The instance of `Bitmap` that you supply must be a mutable bitmap.

3 LRU stands for Least Recently Used. As you can't cache objects indefinitely, caching is always related to an eviction strategy to maintain the cache at a target or acceptable size. In an LRU cache, the "oldest" objects are evicted first.

Support | Sign Out

©2022 O'REILLY MEDIA, INC. TERMS OF SERVICE | PRIVACY POLICY