Pro Android Wearables

Building Apps for Smartwatches

Wallace Jackson

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

Welcome to the *Pro Android Wearables* book, where you will learn how to develop applications for smartwatch devices. There will be a follow-on book called *Pro Android IoT* (Internet of Things), which will cover the other Android APIs such as Android TV, Android Auto, and Android Glass, so in this book I can focus only on an exploring the smartwatch device market.

The reason that smartwatches, along with iTV sets, are continuing to explode is a case of basic economics. There are now dozens of manufacturers, including traditional watch brands, such as Citizen, Rolex, Casio, Tag Heuer, Timex and Fossil, making smartwatches, as well as all of the major consumer electronics giants, including Sony, Samsung, LGE, ASUS, Huawei and Motorola, who now have multiple smartwatch models. This generates incredibly massive competition, which drives down pricing, making this value proposition difficult to argue with. I Google searched Android Wear watches today and found two of the most impressive smartwatches, the Motorola MOTO 360 and the ASUS ZenWatch, priced at less than $200. For a computer on your wrist, made with rose gold and calf leather (ZenWatch) or a beautiful carbon black steel bracelet (MOTO), that is an exceptionally reasonable price point. I expect smartwatches to range from $150 to $450 and to continue to generate increasing sales into the future, while adding screen resolution (480 to 640 pixels), processor cores (two to four), and system memory (1 to 2 GB).

This book will cover how to develop applications for an exploding smartwatch market, and it includes the new **Watch Faces API** released by Google that allows developers to create their application as the watch face design itself! Since that is what a watch is used for, I will discuss the Watch Faces API in detail, so that your smartwatch applications can offer their functions to the users while also telling them the time, date, weather, activity, notifications, and so forth. You will learn how to use Google Play Services and make Android Wear applications that have components running on your smartwatch, as well as on the smartphone or tablet, called a **companion activity application**.

Chapter 1 looks at Android Wear and wearable concepts and design considerations, before you set up the Wear production workstation, including your IDE, SDKs, and New Media Content Development applications in Chapter 2. I will discuss the new features of Android Wear in Chapter 3, before you learn about the IntelliJ IDEA, and create a foundation for

your Wear project in Chapter 4. In Chapter 5 you will set up the IntelliJ IDEA, also known as Android Studio, for production readiness, by making sure all the SDKs and emulators are up to date and creating AVDs to use for round or square watch face testing.

In Chapter 6, you will get ready to start coding by looking at the Android Watch Faces API and all of its functionality and UI design considerations. In Chapter 7, you will actually code your Watch Face Service and Watch Face Engine classes. These drive the Watch Face infrastructure which you will be putting into place in subsequent chapters.

In Chapter 8 you will put your Watch Face Timing Engine into place, learning about the Time and TimeZone classes, as well as implementing a BroadcastReceiver object to manage the implementation of these classes (objects). In Chapter 9, you will implement core Watch Faces API methods that control different watch face rendering styles and event processing.

In Chapter 10 you will learn about vector graphics and how to "render" designs on the smartwatch face using the onDraw() method, and in Chapter 11 you will learn about raster graphics and how to use BitmapDrawable objects along with PNG32 bitmap assets to add digital imagery to your smartwatch designs. In Chapter 12 you will learn digital imaging techniques that will allow you to optimize the number of colors used to accommodate different smartwatch display color limitations, so you can get the most photorealistic results for your smartwatch application design.

In Chapter 13 you will learn about the Google Mobile Services (GMS) APIs and how to access Google Play Services so that your Wear apps can do even more than they can using the native Android and Android Wear APIs. In Chapter 14 you will implement Android Wear Data APIs in your code to create a Watch Face Utility class to manage your users' settings.

In Chapter 15 you will learn how to set up a testing environment for real-world hardware devices and learn about the Android Debug Bridge, or ADB, as well as how to implement USB device drivers for your hardware devices.

In Chapter 16, you will learn how to dealing with API deprecation and class and method call code updates, as you remove the deprecated Time class and replace it with the Calendar and GregorianCalendar class code to make your application more efficient.

Finally, Chapter 17 goes over the Android IoT APIs and other Wear API features to consider for your smartwatch applications, such as voice recognition and location tracking using the Speech and GPS APIs, respectively. With the information in this book, you will be well on your way to developing smartwatch applications using Android Wear and Android Watch Faces APIs!

Chapter 1

Introduction to Android Wearables: Concepts, Types, and Material Design

Welcome to the *Pro Android Wearables* book! This book will show you how to develop Android Studio applications for those Android devices that are outside your normal smartphones and tablets. This book also includes Android development for devices that can be worn on your person, which is why these Android devices are commonly called "wearables."

If you are looking to develop for Android appliances, such as iTV sets, 4K iTV, game consoles, robots, or other appliances, then the book you want is the *Pro Android IoT* (Apress, 2016) book. That title covers development for Android devices known as "Internet of Things," which include devices that are not worn on your person and are beyond the more normal tablets and phones.

This chapter will look at the different types of wearables, as well as their features and popular usage, and you will learn about the terminology, concepts, differentiating factors, and important technology surrounding wearables and take a closer look at what types of Android applications you can develop for wearable devices. I'll get all of this wearables-related learning out of the way here so you can focus on setting up your workstation in Chapter 2, and then get into Android wearables features in Chapter 3. I will also explain the distinction between wearable devices and Android wearable peripherals.

I'll also discuss the new material design features that have been added to Android 5, as these are available for wearables' application development, and you will see all the cool things you can do with these!

Wearable Technology Defined: What Is a Wearable?

The term *wearables*, as well as the terms *wearable technology* and *wearable devices*, is indicative of consumer electronics technology based on embedded computer hardware that is built inside products that are worn on the outside of one's body. This includes clothing and accessories, including jewelry, such as watches, and protective wear, such as glasses, as well as items of clothing such as socks, shoes, hats, and gloves and sports, health, and fitness products that can be comfortably worn somewhere on your body.

Wearable devices often have some modicum of communications capability, and this will allow the device wearer to access information in real time. Data input capability is a necessary feature for wearable devices, as it allows users to access the features of the wearable device and use it to run the applications you are going to be learning to develop over the course of this book. Data input is usually in the form of touch screen interfaces, voice recognition (also known as voice input), or sometimes through use of hardware buttons built right into the wearable device itself.

Thanks to the cloud, local storage is not necessary for a wearable device, although some feature micro SD (secure digital) cards or store data on linked, companion devices. Wearable devices are able to perform many of the same application tasks as mobile phones and tablets; in fact, many wearable devices require that the wearable device be "married" to another Android device (more on this later on in the chapter) within the operating range of Bluetooth 4.x technology.

Android wearable devices tend to be more sophisticated on the "sensor input" side of the equation than hand-held technologies on the market today. This is because wearable devices will provide sensory and scanning features not typically seen with smartphone or tablet devices. Examples of this include features such as biofeedback and the tracking of physiological functions, such as pulse, heart rate, workout intensity, sleep monitoring, and so on.

Examples of wearable hardware devices include smartwatches, smartglasses, textiles, also called smart fabrics, hats, caps, shoes, socks, contact lenses, ear rings, headbands, hearing aids, and jewelry, such as rings, bracelets, and necklaces. Wearable technology tends to refer to things that can be put on and taken off effortlessly. It's important to note that there are versions of the wearables concept that are more radical in nature, for instance the implanted devices, such as microchips or even smart-tattoos. I will not be covering application development for any of these nonmainstream device types in this book. Because the general public will primarily be using smartwatches, I'll be focusing on that type of wearable device. Ultimately whether a device is worn on, or incorporated into, the body, the purpose of these Android wearable devices is providing constant, convenient, portable, seamless, and hands-free access to consumer electronics.

Wearable Application Development: What Types of Apps?

The uses of wearable technology are limited only by your imagination, and the implications of these applications will be far reaching, which is why you have purchased this *Pro Android Wearables* book in the first place!

Wearable applications will influence a wide spectrum of industry verticals in a large number of ways. Some of the many industries that are embracing Android wearable devices include the health care, pharmaceutical, education, fitness, entertainment, music, aging, disability, transportation, finance, enterprise, insurance, automotive, and gaming industries, and the list grows larger daily.

The goal of wearable applications in each of these industries should be to seamlessly incorporate functional, portable electronics and computers into an individual's daily life. Prior to their presence in the consumer market, wearable devices were primarily utilized in military operations as well as in the pharmaceutical, health care, sports, and medicine industries.

More than ten years ago, medical engineers started talking about a wearable device that could monitor the health of their patients in the form of a smart-shirt or smart-wristband. At least one of these is a mainstream technology today, and its aim is to monitor vital signs and send that biofeedback information to an application or a web site for data tracking and analysis.

The types of Android applications that you can create for use on wearable devices is limited only to your imagination. This is because wearables are "paired" with more advanced hardware and thus have those same hardware capabilities that smartphones or tablets have, plus some sensors that smartphones and tablets do not have!

One of the logical application verticals is health and welfare monitoring; because of these heart-rate sensors, Android wearables applications can be created that help with health-related areas of users' lives, such as their work out in the gym, tracking their diet on the go while working or traveling, getting enough sleep, walking or running enough miles in a day, and similar applications that will help users improve their health or at least stay healthy.

Another logical application vertical is social media, as the current trend these days is staying connected to everyone at all times of the day, while also making new friends or business associates. Androids are connected to the Internet, via Wi-Fi or 4G, at all times, so these types of wear apps are also going to be very popular for use on wearable devices.

Of course, games, watch faces or skins, and entertainment consumption will also be a massive application vertical for wearable devices. Let's look at this aspect so you can get some idea about how to apply what you'll be learning!

Android Wearable Fun: Augmented Reality, Edutainment, and Gamification

Although wearables technology could potentially have a significant impact in the areas of social media connectivity, health, dieting, or fitness, the area of wearable technology also promises to have a major impact on the casual gaming, audio video (AV) entertainment, edutainment, and augmented reality (AR) markets. Wearable applications that make everyday tasks into fun to play games, commonly termed *gamification*, is also a major market opportunity.

AR, originally called mixed reality, can leverage wearable technology. AR uses i3D OpenGL capabilities, found in the Android platform, to create a realistic and immersive 3D environment that syncs up with the real world around you in real time, thanks to Java 7 code in your Android 5 application. Whereas Android 4.4 and earlier used Java 6, Android 5 uses

Java 7. Mixing virtual reality or interactive 3D (i3D) with actual reality, using digital video, digital imaging, or global positioning satellite (GPS) location-based technology, is not new by any stretch of the imagination. AR apps are the most advanced type of wearable apps.

AR delivered through the use of wearable devices has been contemplated since before the turn of the century. What's important is that AR hardware prototypes are morphing from bulky technology, such as the massive goggles used by the Oculus Rift, into small, lightweight, comfortable, highly mobile 3D virtual reality systems.

The next most complex type of application that will soon be appearing on a wearable device will be wearable games. You can expect to see casual games created for smartwatches and smartglasses on the market very soon. Careful optimization is the key to creating a game application that will work well on a wearable device, and I will be covering that topic within this book.

Another complex type of entertainment application for a wearable device is the AV application. Playing digital audio or digital video on a wearable device also requires careful optimization, as well as a user who owns a good pair of Bluetooth audiophile headphones, which fortunately are made by more than a dozen major electronics manufacturers these days.

Finally, one of the more complex types of wearable applications is custom smartwatch faces or skins. These turn the watch face that a user looks at all day long into something they want their watch to look like. Of course you can also create loads of text-based apps, like office utilities or handy recipe managers, for instance; these will work great on wearables!

The future of Android wearables applications needs to reflect the seamless integration of fashion, functionality, practicality, visual, and user interface (UI) design. I'll discuss how to do this throughout the book, after you have put together the development workstation in Chapter 2 and created the emulators in Chapter 5, so you have a foundation in place for wearable development.

Mainstream Wearables: Smartwatches and Smartglasses

There are two primary (i.e., mainstream) types of wearable devices that popular consumer electronics manufacturers are scrambling to manufacture.

The smartwatch is currently the most popular type of wearable device, with hundreds of affordable models already available in the marketplace. As the centuries have passed by, watches have become the international fashion statement. Thus, it is no surprise that this is the most popular and useful type of Android wearable device.

The other type of popular wearable genre is smartglasses, and dozens of products have already been released by companies such as Google (Glass) and Vuzix (M100). Let's take a closer look at these two types of wearables hardware, since they are going to be the majority of the device types that run the pro Android wearables applications you will be creating during the course of this book.

Smartwatches: Round Watch Face vs. Square Organic Light-emitting Diode

The smartwatch wearable genre of consumer electronics has the most products out there, with dozens of branded manufacturers and actual product models numbering in the hundreds, with all but one (Apple Watch) running one flavor of Android or another. For this reason, I'll focus on these in this book.

There is also a smartwatch from Samsung, the Galaxy Gear 2, which uses the Tizen OS, utilizing Linux, HTML5, JavaScript, and CSS3 for app development. I won't be covering these in this book, instead focusing on the Samsung Gear S.

Android 5 Wear Software Development Kit (SDK) supports two different smartwatch face types, round and square, as watches normally come in these two configurations. You'll create Android virtual devices (AVDs) (software emulators) for both of these smartwatch types in Chapter 5.

Smartglasses: Glasses and Other Smartglasses Manufacturers

The smartglasses wearables consumer electronics product genre is the next fastest growing wearables products genre. Brand eyewear manufacturers are scrambling to get into this wearables space, so look for smartglasses from Luxottica soon. For this reason, this will be the secondary focus in this book.

Smartglasses will generally run the Google Glass Development Kit (GDK) or Android 4.x, and you can expect Android 5 smartglasses wearables in 2015. There are a number of smartglasses companies, including Google, Vuzix, GlassUp, Sony, six15, and Ion. Google has of course stopped producing the glasses, promising new and better products in the future.

Wearable Application Programming Interfaces

There are two primary application programming interfaces (APIs), both of which run under Google Android Studio, that can be used to access features for wearable devices that are not yet standard in the version of Android 5 that spans across mainstream devices, such as smartphones and tablets. The smartwatch API is called *Wear SDK* and the smartglasses API is called *Glass GDK*. It's important to note that some wearable devices can run the full Android operating system (OS).

The smartwatch example of this would be the Neptune Pine Smartwatch, which runs a full Android OS, and the Google Glass product does not require that you use Glass GDK unless you need to use special features of Google Glass or want to develop "native" glass apps. In other words, Google Glass will run Android apps that run on normal smartphones and tablets.

This means that Neptune Pine and Google Glass can run the same application you develop for other Android devices. Newer versions of this Neptune Pine product line will utilize the Wear SDK, which is largely what I will be covering within the scope of this book.

Android Studio 1.0: Android Wear SDK

Android Wear SDK is the API created by Google for use with Android wearables (wearable devices that run on Android).

It was launched at the beginning of 2015 via the Android developer web site along with several customized "vertical" APIs, including Android Auto SDK (for automobiles), Android TV SDK (for 2K or 4K iTV sets), and an Android Wear SDK (for smartwatches). For now, Wear SDK is targeted at smartwatches, but later it may expand to other wearables such as shoes, hats, and the like.

Android Wear SDK provides a unified Android wearables development platform that can span across multiple smartwatch products. Before the Wear SDK was available, a smartwatch manufacturer had to either provide its own APIs, like the Sony SmartWatch One and Two did back in 2014, or support the full Android 4 OS, like the Neptune Pine did during 2014.

It is important to note here that this Android Wear SDK does not provide a separate operating system, but in fact is an extension of the Android 5 OS that requires a portion of the Android wearable application to run on your host Android device. This would normally be an Android smartphone, as that is the most portable device and the device type that connects to a wide variety of networks and carriers.

Because the Android smartwatch represents the majority of the wearable application marketplace, I will focus the majority of this book on that area of wearable application development, although I'll also cover Google Glass in Chapter 17.

Google Glass Development Kit: GDK for Android or Mirror

When developing Google Glass wearable applications, you have two different GDK API options. You can use these separately or in conjunction with each other. Additionally, there is a third option of simply using the Android OS without either of the APIs. Let's take a closer look at Glass development.

Google Glass's Android Studio GDK: The Glass Development Kit

The Google Android GDK is an add-on to the Android OS APIs (also known as the Android 5 SDK), which allows you to build what Google terms *Glassware*. Glassware comprises the Google Glass wearable apps that run directly on Google Glass hardware.

In general, you would want to use this Android GDK development approach if you need direct access to unique hardware features of the Google Glass, real-time interaction with the Google Glass hardware for your users, or an off-line capability for your application if no Internet, Wi-Fi, 2G, 3G, or 4G cellular network is available.

By using the Android SDK in conjunction with the GDK, you can leverage the wide array of Android APIs while, at the same time, designing a great user experience for Google Glass owners. Unlike the Mirror API, Glassware built using this Android GDK runs on Glass itself. This allows access to Glass hardware's low-level, proprietary (unique to Google Glass) product features.

Develop Google Glass Apps Using Only the Android Environment

Google designed the Glass platform to make their existing Android SDK work on Glass. What this means is that you may use all of the existing Android development tools, which you'll be downloading and installing in Chapter 2, and your Glassware can be delivered using the standard Android application package (APK) format.

This opens up a lot of those other pro Android development book titles, such as *Pro Android Graphics* (Apress 2013) or *Pro Android UI* (Apress 2014), which will show you how to make visual Android applications that work well on Google Glass devices. This is because the Google Glass product is designed to run the full Android OS, and, therefore, any applications that will run on it. This means you can develop an application that will run across all of the Android devices out there, including Google Glass. This allows a code once, deliver anywhere (highly optimized) development work process for your app, as long as users don't need to run on smartwatches, other than Pine!

Using RESTful Services with Google Glass: The Mirror API

There's another API that works with Google Glass and is not tied to Google Android OS at all. This is what is known as the Google Glass Mirror API, and it is what is commonly known as a RESTful API.

The Mirror API allows developers to build Glassware for Google Glass using any programming language they choose. RESTful services provide easy access to web-based APIs that will do most of the data transfer heavy lifting for the application developer.

In general, you would want to utilize this Mirror API if you need to use a cross-platform infrastructure such as HTML or PHP, need to access built-in functions for the Google Glass product, or need platform independence from the Android OS. This would be how you would use Google Glass with iOS or Windows, for instance.

Hybrid Glass Applications: Mixing Android GDK and the Mirror API

Finally, it's interesting to note that developers can also create "hybrid" Google Glass applications. This is because, as you may have suspected, the Google Glass Mirror API can interface with the Google Glass Android GDK.

This is done by using a menu item to invoke Mirror API code that sends an Intent object to the Android GDK API and then to the GDK application code. I'll be using Intents, which are an Android platform-specific Java Object type, in this book. Intents are used to communicate among applications, menus, devices, activities, and APIs, such as the Mirror API. You can even use this hybrid development model to leverage existing web properties that can launch enhanced i3D experiences that run directly on Google Glass.

True Android or Android Peripheral: Bluetooth Link

In the world of pro Android wearables, and in some cases even in the world of pro Android appliances, there is often a distinction that you will need to be aware of as a developer that the marketers of Android products will have a tendency to want to hide. This is because the cost to manufacture one type of Android device will be quite high (miniaturization), while the cost to manufacture another type of Android device will be quite low, so profit margins will be higher, especially if the public can be convinced the product is running an Android OS, when in fact it's not actually doing so!

This is quite evident in Android wearables product segments, which include smartwatches and smartglasses, such as Google Glass. A couple of smartwatches are True Android devices; that is they have a computer processing unit (CPU), memory, storage, an OS, Wi-Fi, and the like, right inside of the smartwatch. A good example of one of these True Android devices is the Neptune Pine.

True smartwatch devices would actually be like having a full smartphone on your wrist and would be offered by telecommunications carriers just as smartphones currently are. This True Android smartwatch would be your only Android device, you would not need to carry a smartphone anymore. Embedded computer miniaturization advances will eventually allow all smartwatches to do the same things that the Neptune Pine did in 2014, placing a full-blown Android device on someone's wrist.

In case you are wondering, I borrowed this "true" Android description from the TrueHD (HDTV) industry term. TrueHD is 1920-by-1080 resolution, and it is a necessary descriptive modifier because there is another lower 1280-by-720 resolution in the marketplace that is called just HD (I call it pseudo-HD).

Other smartwatches are not True Android devices and could be described as more of a "peripheral" to your existing smartphone, phablet, or tablet, and these use Bluetooth technology to become an *extension* of an Android device that has the CPU (processor), memory (application runtime), data storage, and telecommunication (Wi-Fi access and 4G LTE cellular network) hardware.

Peripheral Android devices would obviously require a different application development work process and have different data optimization and testing procedures to achieve an optimal performance and user experience.

Obviously, because this book will be looking at developing for some of these more popular Android wearable devices, I will be getting into this "remote Android peripheral" development issue in greater detail and taking a look at how to design and optimize wearable peripheral apps.

I just want you to be aware that there are two completely different ways to approach Android development now: on-board, or native Android apps, and remote or two-way (back-end) communication Android app functionality.

With the advent of Bluetooth wearables and second-screen technology (which is covered in the book *Pro Android IoT* [Apress, 2016]), this is going to become an important distinction in Android applications' development as time goes on, and these extension Android products continue emerging into the market.

The bottom line is that you need to know which consumer electronics device is hosting the CPU, memory, storage, and telecommunications hardware, which consumer electronics device is hosting the touch screen, display, and input, and which technology (and how fast it is) is connecting those two together. This is important for how well you'll be able to optimize app performance, as user experience (UX) is based on how *responsive* and easy an app is to use.

Wearable Apps Design: Android 5 Material Design

Android 5, released in 2014 along with the Android TV, Wear, and Auto SDK add-ons, features an all new UI paradigm. Google calls it *Material Design* because it is more 3D savvy. Texturing or "skinning" a 2D or 3D object involves using what are commonly termed *materials* in the media design industry.

Google created Material Design to be a far-reaching UI design guideline for end-user interaction, animated motions, and visual design across the Google Chrome OS and Android OS platforms, as well as across consumer electronics devices that run Chrome OS (HTML5, CSS3, and JavaScript on top of Linux) or Android 5 OS (Java 7, CSS3, and XML on top of a 64-bit Linux Kernel).

One of the cool features in the Android 5 OS, which you may have learned about if you read the book *Android Apps for Absolute Beginners* (Apress 2014), is support for Material Design in apps across all types of Android 5 devices.

You will learn about using Material Design for Android wearables apps in this chapter, as well as throughout the rest of this book. Android 5, also known as Android API Level 21 (and later), offers some new components and new OS functionality, specifically for Material Design.

This includes a new Android 5 theme, Android View widgets for new viewing capabilities, improved shadows and animation APIs, and improved Drawables, including better vector scaling, 32-bit PNG tinting using the 8-bit alpha, and the automatic Color Extraction API. I will be covering all of these in more detail during the rest of this chapter.

The Android Material Design Themes: Light and Dark

This Android 5 Material Design Theme provides the new Android 5 conforming style to use for your Android 5 apps. Because the Android Wear SDK is a part of Android Studio 1 (Android 5 plus IntelliJ), these new themes will apply to pro Android wearables as well. You will be installing Android Studio, as well as some other open source content development software in Chapter 2, and exploring Android Studio 1.x and IntelliJ in Chapter 3 and Java 7 in Chapter 4. If you're developing for Android 4, you will use the HOLO Theme**;** if you are developing for Android 5, you will use the Material Theme.

Both Theme.Holo and Theme.Material offer a dark and a light version. You can customize the look of the Material Theme to match a brand identity using the custom color palette you define. You can tint an Android Action Bar as well as the Android Status Bar by using Material Theme attributes.

Your Android 5 widgets have a new UI design and touch feedback animations. You can customize these touch feedback animations, as well as the Activity transitions for your app. An Activity in Android is one logical section or UI screen for your application. I am assuming you already have knowledge of Android lingo, because this is an intermediate to advanced level (pro) book.

Defining the Wearable Material Theme: Using the Style Attribute

Just as in the previous versions of Android, the material theme is defined using the Android Style attribute. Examples of the various material themes would be defined using XML, using the following XML 1.0 markup:

```
@android:style/Theme.Material                    (the default dark UI version)
@android:style/Theme.Material.Light              (the light UI version)
@android:style/Theme.Material.Light.DarkActionBar (a light version with a dark version
                                                 Actionbar)
```

As I mentioned, the Theme.Material UI style (or theme) is only available in Android 5, API Level 21 and above. The v7 Support Libraries provide themes with Material Design styles for some pre-5 View widgets and support for customizing the color palette prior to Android 5.

Defining the Wearable Material Theme Color Palette: The Item Tag

If you wanted to customize your Material Design theme's primary color to fit your wearables app branding, you would define your custom colors using the **<item>** tag, nested inside a **<style>** tag, nested inside a **<resources>** tag inside a **themes.xml** file. You create an **AppTheme** with parent attributes inherited from the **Theme.Material** parent theme and add your custom color references using the **colors.xml** file that holds the hexadecimal color data using an XML markup structure. This should look something like this:

```
<resources>
    <style name="AppTheme" parent="android:Theme.Material">
        <item name="android:colorPrimary">@color/primary</item>
        <item name="android:colorPrimaryDark">@color/primary_dark</item>
        <item name="android:colorAccent">@color/accent</item>
    </style>
</resources>
```

Again, I assume you know basic Android (Java and XML) development here. The style name used for the app here is **AppTheme**, it references a **parent** style of **Theme.Material** and sets custom color values, set in a **colors.xml** file, using **<item>** tags containing your main theme style constants—**colorPrimary**, **colorPrimaryDark**, and **colorAccent**. Android is hard wired to use these theme constants, so all you have to do is reference custom color values to them.

Customizing a Wearable Material Theme Status Bar: *statusBarColor*

You can also easily customize the application Status Bar for **Theme.Material**, and you can specify another color that fits the wearable application brand and will provide a decent amount of color contrast to show the white status icons. To set the custom color for your application Status Bar, add an **android:statusBarColor** attribute when you extend a **Theme.Material** style definition. Using the previous example, your XML should look like this:

```
<resources>
    <style name="AppTheme" parent="android:Theme.Material">
        <item name="android:colorPrimary">@color/primary_color</item>
        <item name="android:colorPrimaryDark">@color/primary_dark</item>
        <item name="android:colorAccent">@color/accent_color</item>
        <item name="android:statusBarColor">@color/status_bar</item>
    </style>
</resources>
```

The **statusBarColor** constant will *inherit* the value of the **colorPrimaryDark** constant if you do not provide one specifically, as is seen above. You can also can draw behind the Status Bar using the *alpha channel* component of an Android 5 **#AARRGGBB** 32-bit hexadecimal color data value. If you want to delve into Android graphics, check out the book *Pro Android Graphics* (Apress 2013).

For example, if you wanted to show the Status Bar as completely transparent, you would use an **@android:color/transparent** constant, which sets the alpha channel to zero (off or **#00000000**). However, this would not be a good UI design practice, as you could have a background with white in it behind the Status Bar, which would then render the Status Bar icons invisible.

So what you would really want to do is create a **tinted** Status Bar over a background (image, photo, 3D, 2D, artwork). You should use a dark gradient to ensure the white status icons are visible. To do this, you would set the **statusBarColor** to transparent and also set the **WindowManager** object's **windowTranslucentStatus** attribute to a data value of true using an Android **WindowManager.LayoutParams** class (objects) **FLAG_TRANSLUCENT_STATUS** constant. You can also use the **Window.setStatusBarColor()** method with Java code to implement Status Bar animation or translucency fade-in or fade-out.

As a UI design principle, your Android Status Bar object should always have a clear demarcation against an Action Bar, except in cases where you design custom UI images or new media content behind these bars, in which case you should use your darkening gradient, which will ensure that icons are still visible. When you customize both UI navigation (Action Bar) and a Status Bar, you should make them both transparent or only change the transparency for the Status Bar. The navigation bar should remain black in all other cases.

Customizing a Wearable Material Theme: Individual View Themes

Android Styles and Themes can not only be used for customizing a look and feel for your entire wearables application globally, but they can also be used to style and theme *local screens*, which are components of your application.

Why would one want to go to the trouble of developing a style or theme for an *individual* View object or Activity object in Android 5, you might ask?

The answer can be found in the concept of UI design *modularity*, which is a cornerstone of Android wearables app development. Once you develop a Style and Theme using an XML file, you can apply it whenever necessary, which will probably be multiple times during your wearables app development process. In this way, you do the UI design work once (create a module) and apply it many times thereafter. This also ensures that the theme or style is applied in exactly the same way every time. If you need to get into UI design for Android 5 development in greater detail, the book *Pro Android UI* (Apress 2014) covers all of the Android UI design issues in depth.

UI elements (Android *widgets* subclassed from View) in your XML user interface layout container definitions (Android layout containers are subclassed from **ViewGroup**) can reference an **android:theme** attribute or an **android:style** attribute. This allows you to reference your prebuilt style and theme resources in a modular fashion.

Referencing one of the prebuilt style or theme attributes will then apply that UI element as well as any child elements inside that UI design element. This capability can be quite useful for altering theme color palettes in a specific section of your wearables application UI design.

Android Material Design View Widgets: Lists and Cards

The Android 5 API provides two completely new View subclasses (widgets). These can be used for displaying information cards or recyclable lists using the Material Design themes, styles, animation, and special effects.

The **RecyclerView** widget is a plug-and-play enhancement of Android **ListView** class. It supports many layout types and provides performance improvement. The **CardView** widget allows your wearable application to display contextual pieces of information using “cards” that have a consistent look and feel.

Let’s take a closer look at the new UI design tools before I move on to dropshadows, animation, and special effects like Drawable tinting and Color extraction.

Android *RecyclerView* Class: Optimized (Recycled) List Viewing

The Android RecyclerView is a UI design widget that is a more feature-filled version of the Android ListView widget. The RecyclerView is used to display extensive lists of applications data. What is unique about the class is that the data contained in the View can be scrolled extremely efficiently. The way this is done is through the RecyclerView ViewGroup (a layout container) subclass. It holds a limited number of data (View) objects inside its ViewGroup layout container at any single moment in time.

This memory optimization principle is quite similar to how digital video streaming works, keeping only the currently utilized portion of your data list in the system memory, which makes this class faster and more memory efficient. You would want to utilize Android’s RecyclerView widget when you have data collections where the data inside are going to be changed at runtime, based on the actions of your application’s end users or by network transactions.

The `RecyclerView` class accomplishes all this by providing developers a number of software components that wearable developers can implement in their code. These include layout managers, for positioning data View items in the List, animation to add visual effects to data View item operations, and flexibility in designing your own custom layout managers and animation for your wearable application's implementation of this RecyclerView widget.

Android *CardView* Class: The Index Card Organization Paradigm

The Android **CardView** class is a ViewGroup layout container class extending the **FrameLayout** class. The Android FrameLayout class allows you to display a single View UI element (widget) so the CardView would be a collection of FrameLayout individual Views in the paradigm of a stack of 3-by-5 index cards. This class allows you to show any informational data for your wearable application on virtual cards that have a consistent look across the Android (application, wearable, TV, or auto SDK) platforms.

Your CardView widget can also feature shadows and rounded corners for each card in this CardView layout container, although it is the CardView itself that is dropshadowed and rounded, not each individual card. To create a card with a shadow, you need to use a **card_view:cardElevation** attribute.

The CardView class accesses the actual elevation attribute and creates the dynamic shadowing automatically if your user is using Android 5 (API Level 21) or later, and for earlier Android versions, it will create a programmatic shadow implementation based on earlier versions.

If you wanted to enhance the appearance of your CardView widget, you would provide a custom *corner radius* value, say 6dip, which would create rounded corners for each card in your CardView using a **cardCornerRadius** attribute.

If you wanted to show a background image, behind your CardView widget, you would provide a custom *background color* value, like #77BBBBBB, which would create a light gray transparent background color for each card that is in your CardView, using the **cardBackgroundColor** attribute.

If you wanted a dropshadow behind the CardView widget, you would provide a custom *elevation* value, say 5dip, which would create a nice, highly visible dropshadow behind each card in a CardView using a **cardElevation** attribute. Before you use the **cardElevation** attribute, you will need to set a Padding compatibility constant, called **cardUseCompatPadding**, to a value of true in order for the dropshadowing (elevation) effect to be computed by CardView.

To access these CardView attributes, you must import a custom XMLNS (or XML naming schema) for both your LinearLayout parent layout container class as well as for the nested CardView child layout container class. This is done by using the following XML parameter inside each layout container tag:

```
xmlns:card_view=http://schemas.android.com/apk/res-auto
```

Once you have this in place, you can use the attributes specified above by prefacing them with a **card_view:** modifier, so that your **cardCornerRadius** attribute would be then be written as **card_view:cardCornerRadius**, for example.

To implement the three examples I outlined previously in this section, the markup for the LinearLayout parent layout container containing a CardView containing a TextView object would look something like the following XML:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              xmlns:tools="http://schemas.android.com/tools"
              xmlns:card_view="http://schemas.android.com/apk/res-auto" >
    <android.support.v7.widget.CardView xmlns:card_view="http://schemas.android.com/apk/res-auto"
        android:id="@+id/my_card_view"
        android:layout_gravity="center"
        android:layout_width="180dip"
        android:layout_height="180dip"
        card_view:cardCornerRadius="6dip"
        card_view:cardBackgroundColor="#77BBBBBB"
        card_view:cardUseCompatPadding="true"
        card_view:cardElevation="5dip" >
        <TextView android:id="@+id/info_text"
                  android:layout_width="match_parent"
                  android:layout_height="match_parent" />
    </android.support.v7.widget.CardView>
</LinearLayout>
```

Now let's look at special effects applications in Material Design and learn about dropshadowing and animating the View widgets.

Android Material Design Effects: Shadows and Animation

Although Android has always had a shadows and animation feature set, which works with Android View objects (called Android widgets), Material Design takes these effects to a new level by providing more advanced shadows so that everything on a screen has a 3D z axis element to it. For instance View objects other than Text objects can now access dropshadows, and there are now advanced transition animation effects, such as curve interpolation motion and things that simulate 3D effects on the screen such as ripples.

Android Material Design 3D Effects: Automatic View Shadowing

In addition to the x and y properties, Android widgets, which are subclassed from the `View` class in Android, now feature a third, z axis property. This property, which is called an *elevation* property, defines the height of the View object, which in turn determines the characteristics of its shadows.

As many of you who are familiar with 3D know, this takes Android UI design from a 2D place into the exciting 3D realm. This allows photo-realistic i3D UI designs, whereas before, only "flat," 2D UI designs were possible.

This new elevation property was added to represent the elevation or height of a View object, relative to other View objects above and below it in the UI design. The elevation property (or if you prefer, attribute or characteristic) will be used by the Android OS to determine the size of the shadow, so a View object (widget) with a larger elevation value should cast a wider shadow, making the widget appear to be at a higher elevation.

The View widgets in the UI design will also obtain their drawing order via the z values. This is commonly called the *z order* or *z index*, and your UI View objects that have been assigned a higher z value will always appear on top of other View objects (widgets) that have been assigned lower z values.

Android Material Design Animation: Touch Feedback for Your UI

The upgraded Animation API in Android 5 lets you create custom animation for touch feedback for your UI controls (widgets). These allow for triggering these animation effects based on changes in the View widget state, and also allow Activity transitions when the user navigates between Activity screens in your application.

The Material Design–related enhancement to the Animation API allows you to respond to touch screen events on your View widget using new *touch feedback animations*. These implement the new *ripple* element (**RippleDrawable** class), and can be defined as *bounded* (contained within the View) or as *unbounded* (emanating beyond a View bounds). Defining this using XML is fairly straightforward, although it can also be defined using Java 7.

To define a bounded ripple touch feedback animation, reference it inside the **android:background** parameter in your View widget tag, like this:

```
<Button android:id="@+id/my_rippling_muscles_I_mean_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="?android:attr/selectableItemBackground" />
```

To define an unbounded ripple touch feedback animation, again reference it using the **android:background** parameter in your View widget tag, like this:

```
<Button android:id="@+id/my_rippling_abs_I_mean_button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:background="?android:attr/selectableItemBackgroundBorderless" />
```

Touch feedback animations are now integrated inside several standard View widget subclasses, such as the `Button` class. A new API lets you customize these touch feedback animations so you can add them to your custom View.

Users can now hide and show View widgets using a circular reveal animation, which adds another level of wow-factor to the Android operating system. Those of you familiar with 2D and 3D graphics will know that this is being accomplished by applying an animated circle (ShapeDrawable) clipping shape into the rendering pipeline between the two UI widget visuals.

Android Material Design Transitions: Enhanced Activity Transitions

In the area of activity transitional animation, you can now switch between activities with custom activity transition animations. The custom Activity transitions will be applied by the Android OS whenever the user transfers from one Activity to another using an Intent object.

These Activity animations are intended for usage in conjunction with other Material Design animation effects. I suspect that these were added so that the Material Design UX is uniform across the entire Android 5 wearable application. Now, both View and Activity objects offer prebuilt animation! You can also create custom transitional animation, as you could before Android 5.

There are four different types of prebuilt Activity animations in Android 5 Material Design: *enter* and *exit* control full-screen animation effects, and *SharedElementEnter* and *SharedElementExit* control localized UI elements effects for UI elements, which are *shared* (on both Activity screens).

The enter Activity animation will be triggered when a user switches into a new Activity screen from another Activity screen in a wearable application and, thus, this enter transition determines how View widgets in an Activity enter the screen. For example, in an *explode* transition, View widgets enter the screen from the outside of the screen, flying in toward the center of the screen. Be sure to use graphic designs for your Activity that enhance this transitional special effect.

Conversely, an exit type of Activity animation can be triggered when users exit an Activity screen in their wearable application. The exit transition will determine how View widgets in your wearable Activity exit the screen. For example, using the explode transition example, in the exit transition, your wearable application View widgets will exit the screen traveling away from the center, the opposite of an enter explode transition animation.

If there are shared UI elements between two Activity screens, then the *shared element* transitions can be applied for this UI design scenario. As you may have guessed, these are called the SharedElementEnter and SharedElementExit transitions. The SharedElement transition determines how View widgets that are shared between two Activity screens will handle the transition between these Activity screens.

For example, if two Activity screens have the same digital image, but it's in a different position or resolution, the changeImageTransform shared element transition will translate (reposition) or scale (resize) the image smoothly between the two Activity screens.

There are other shared element transitions such as the changeBounds shared element transition, which will animate any changes in the layout bounds of any target Activity View widgets. There is also a changeTransform shared element transition that can animate any changes in the scale (size) and rotation (orientation) of target View widgets between two Activity screens being transitioned. Finally, there is a changeClipBounds shared element transition, which will animate any changes in the clipping path boundary for target Activity View widgets that have clipping paths assigned to them.

Android 5 now supports three primary enter and exit transitions; these can also be utilized in conjunction with a shared element transition. There is the *explode* transition, which moves View widgets in or out from the center of the screen, the *slide* transition, which slides View widgets in from, or away from, the edges of the scene, and the *fade* transition, which adds, or removes, the View widgets from the screen by changing their opacity value.

You're probably wondering if there's any way to create custom transitions. This is done by creating a custom transition subclass using the Visibility class, a subclass of the `Transition` class, created for this exact purpose.

Android Material Design Motion: Enhanced Motion Curves or Paths

Those of you who are experienced with 3D animation software such as Blender3D or digital video editing software such as Lightworks are familiar with the concept of using *motion curves* to control the rate of change in speed for things like video clips or 3D object movement in scenes.

This is called a motion curve because it allows you to precisely control the way in which something moves over time, which is important in film making, character animation, and game design. There is another closely related tool that is called a *motion path*. A motion path defines how an object will move through a 2D or 3D space.

Therefore, a motion curve is a tool that will be used for defining *temporal animation attributes* (changes in animation speed over time), while a motion path is a tool that will be used for defining *spacial animation attributes*.

Animation in Android OS, as well as in the new Android 5 Material Design, relies on these motion curves using the Interpolator class. This class can now be used to define more complex 4D motion curve interpolation (a fourth dimension, or 4D, means change over time). Material Design now adds motion paths to support 2D spatial movement patterns, so now not only the rate of movement can be controlled, but also where that movement occurs.

The **PathInterpolator** class is a new Interpolator subclass based on Bézier, which is a complex type of curve definition mathematics. Bézier curves have been implemented in Android 5 as Path (class) objects. This Interpolator class can specify Bézier motion curves in a 1-by-1 square, with anchor points at 0,0 and 1,1 and with custom *x,y control points*, which developers specify using the **pathInterpolator** class's constructor method parameters. You will usually define a **PathInterpolator** object using an XML resource, like this:

```
<pathInterpolator
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:controlX1="0.5"
    android:controlY1="0"
    android:controlX2="1"
    android:controlY2="1" />
```

This Android 5 operating system provides XML resources for three basic new motion curves in the Material Design specification. These would use markup and would be referenced in your XML markup, or Java code, using the `@interpolator/` path referencing header, like this:

```
@interpolator/linear_out_slow_in.xml
@interpolator/fast_out_linear_in.xml
@interpolator/fast_out_slow_in.xml
```

Android Material Design Animate State Change: *StateListAnimator* Class

Android state changes, such as the signal meters on your phone, can now be animated using the new Android 5 Material Design feature set. This will allow wearable applications developers to add even more detail to the wow-factor elements inside their Android wearables' applications design.

The new Android StateListAnimator class lets developers use ObjectAnimator objects that are triggered (run) by the Android OS whenever the state of a View object changes. The way you would set up your StateListAnimator as an XML resource leverages the **<selector>** tag (selects among different states) and the **<set>** tag (creates selection set), along with **<item>** tags defining the states and the **<objectAnimator>** tags defining the Object Animation. A simple **pressed=true** and **pressed=false** StateListAnimator Selector object is set up in this fashion, by using a four-level (deep) nested XML construct:

```
<selector xmlns:android="http://schemas.android.com/apk/res/android" >
   <item android:state_pressed="true" >
      <set>
         <objectAnimator android:propertyName="translationZ"
             android:duration="120"
             android:valueTo="5dip"
             android:valueType="floatType" />
      </set>
   </item>
   <item android:state_pressed="false"
         android:state_enabled="true"
         android:state_focused="true" >
      <set>
         <objectAnimator android:propertyName="translationZ"
             android:duration="120"
             android:valueTo="0"
             android:valueType="floatType" />
      </set>
   </item>
</selector>
```

To attach custom state change animations to the View, define an ObjectAnimator using the `<selector>` element in an XML resource file as shown above. Next, reference it inside the View object XML tag you want to be effected by it using the **android:stateListAnimator** XML parameter.

To assign this state change in animations StateListAnimator to a View in your Java code, you should utilize an **AnimationInflater.loadStateListAnimator()** method call, and then remember to assign this ObjectAnimator to your View, using the **View.setStateListAnimator()** method call.

It is important to remember that whenever your wearables application theme extends a Theme.Material Material Design theme, UI Button objects have a z animation enabled (setup and activate) by default. To avoid this behavior in your UI Button objects, set this **android:stateListAnimator** attribute to a data value of **@null** in the XML markup, or "**null** in Java code.

Next let's look at what Android 5 added to a plethora of Android Drawable classes already in Android 4.x to be able to provide this greatly enhanced Material Design capability via animated vectors, bitmap graphics, tinting capabilities, color extraction, and new state animation graphics.

Android Material Design Graphics Processing: Drawables

There are also some new Drawable API capabilities for Material Design that make it easier to use Android Drawable objects to help you implement sleek material design wins for your applications. New Android Drawable classes include a VectorDrawable, AnimatedVectorDrawable, RippleDrawable, Palette, and AnimatedStateListDrawable, all of which I'll discuss in this section.

These new Android 5 Drawable classes add Scalable Vector Graphics (SVG) path support, morphing, i3D ripple special effects, palette color extraction, and animated transitions for multistate Drawable objects to Android OS. Android 5 has added some very powerful classes, where 2D vector and bitmap graphics are concerned!

Android 5 Drawable Tinting: *.setTint()* and *.setImageTintMode()*

With Android 5 and above, you can *tint* BitmapDrawable objects as well as NinePatchDrawable objects. You can tint the digital image objects in their entirety or limit this tinting effect to certain pixels. This is done by defining an alpha channel to "mask" the tinting effect.

You can tint these Drawable objects using Android Color class resources or using Android Theme attributes that reference these Color class resources. Usually, you would create these assets once, and then use them across your wearable applications, using the tint capability to tint them as needed to match your theme. As you might imagine, you can use this for optimization, as you would use far less graphic image assets across an entire application.

You can apply a tint to BitmapDrawable or NinePatchDrawable objects in the Java code for your wearable application using a **.setTint()** method. You can also set the tint color and the tint mode in the XML UI layout container definition. This is accomplished by using the **android:tint** and an **android:tintMode** parameters (attributes) inside your View object tags.

Android 5 Vector Drawable Objects: The *VectorDrawable* Class

The new Android 5 `VectorDrawable` class and the object created using this class can be scaled up or down without losing any quality. This is because, unlike the bitmap image, a vector image is made up of code and mathematics (lines, curves, fills, and gradients) and not pixels. Therefore, if scaling occurs, the vector image will actually be rendered to fill the amount of pixels available to display it. A vector image is not being resampled like a pixel-based image is when it is scaled, but rather *rerendered* to fit the new screen resolution, whether it's a 320-by-240 smartwatch or 4096-by-2160 iTV.

For this reason, vector imagery can be used from wearables all the way up to 4K iTVs, with the same exact visual quality. This is not possible using bitmap images. Because vector imagery is text based, it will be at least one order of magnitude more data compact than bitmap imagery, because vectors are code (math and text), not an array of data-heavy pixel values.

As you might imagine, vector imagery would be perfect for single-color app icons. You only need one image asset for a vector image, as opposed to the bitmap image format, where you would need to provide an asset file for each screen density. To create a vector image, you would define SVG data for the shape inside of a `<vector>` XML parent tag. The XML markup to define an SVG vector image of a color filled square would look like the following:

```
<vector xmlns:android="http://schemas.android.com/apk/res/android"
        android:height="320dip"
        android:width="320dip"
        android:viewportWidth="160"
        android:viewportHeight="160" >
        <path android:fillColor="#AACCEE"
              android:pathData="M0,0 L0,100 100,100 100,0 Z" />
</vector>
```

SVG imagery is encapsulated in Android 5 using VectorDrawable objects. For information about SVG Path command syntax, see the SVG Path reference on the W3C web site (http://www.w3.org/TR/SVG/paths.html). I also cover this in depth in the book *Beginning Java 8 Games Development* (Apress 2014), since Java 8 and JavaFX have extensive **SVG Path** support.

You can also simulate the popular multimedia software genre called *warping* and *morphing* by animating the SVG path property of VectorDrawable objects, thanks to another all new Android 5 class called AnimatedVectorDrawable.

Next let's take a closer look at all of the new Android 5 (automatic) color extraction capabilities, which are provided by the Android Palette class.

Android 5 Automated Color Palette Extraction: The *Palette* Class

Android 5 added a new Palette class that facilitates a *colors extraction* algorithm, which allows developers to automatically extract prominent colors from a bitmap image asset in your application. The Android Support Library r21 and above includes the Palette class, which lets you extract prominent colors from an image in Android application versions previous to version 5, such as Android 3.x and Android 4.x applications. Palette will extract the following types of prominent colors from a bitmap image's color spectrum:

- Vibrant
- Vibrant dark
- Vibrant light
- Muted
- Muted dark
- Muted light

The Palette class is a helper class, which helps developers extract six different classifications of colors (listed above). To use this helper class, you would pass the bitmap object you want palettized to the Palette class's **.generate(Bitmap image)** method, using the following method call:

```
Palette.generate(Bitmap imageAssetName);
```

Be sure to do this in a background thread where you load the image assets. If you can't use a background thread, you can also call the Palette class's **.generateAsync()** method, providing a listener instead, like this:

```
public AsyncTask<Bitmap, Void, Palette> generateAsync (Bitmap bmp, Palette.
PaletteAsyncListener pal)
```

You can also retrieve the prominent colors from the image using the *getter* methods in the Palette class, like **.getVibrantColor()** or **.getMutedColor()**. A **.generate()** method will return a 16-color palette. If you need more than that, you can use another (overloaded) **.generate()** method with this format:

```
Palette.generate(Bitmap image, int numColorsInPalette);
```

I looked at the source code for this Palette class and there does not seem to be any maximum number of colors you can ask this class to provide (most palettes max out at 8-bit color, or 256 colors). This allows for some very interesting applications for this class, as it is not tied to 8-bit color. The more colors you ask for in a palette, the longer your processing time. This is why there's an **AsyncTask<>** version of the **.generate()** method call!

To use the Palette class in your wearables application's IntelliJ project, you will need to add the following Gradle dependency to your app's module:

```
dependencies { ... (default Gradle dependencies remain in here)
               compile 'com.android.support:palette-v7:21.0.+'  }
```

Android 5 State Animation: An *AnimatedStateListDrawable* Class

Besides the new Android RippleDrawable class, which creates the effects discussed in the past couple sections, and VectorDrawable class, there's also an all new AnimatedStateListDrawable class that allows you to animate the transition between StateListDrawable objects.

The Android AnimatedStateListDrawable class lets you create an animated state list of drawable objects (hence the class name), which calls animations between state changes for the referenced View widget. Some of these system widgets in Android 5 will use these animations by default. The following example shows how to define an AnimatedStateListDrawable by using an XML resource:

```
<animated-selector xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/pressed"
          android:drawable="@drawable/drawable_pressed"
          android:state_pressed="true" />
```

```
    <item android:id="@+id/focused"
          android:drawable="@drawable/drawable_focused"
          android:state_focused="true" />
    <item android:id="@id/default"
          android:drawable="@drawable/drawable_default" />
    <transition android:fromId="@+id/default"
                android:toId="@+id/pressed" >
          <animation-list>
                <item android:duration="85"
                      android:drawable="@drawable/asset1" />
                <item android:duration="85"
                      android:drawable="@drawable/asset2" />
          </animation-list>
    </transition>
    <transition android:fromId="@+id/pressed"
                android:toId="@+id/default" >
          <animation-list>
                <item android:duration="85"
                      android:drawable="@drawable/asset2" />
                <item android:duration="85"
                      android:drawable="@drawable/asset1" />
          </animation-list>
    </transition>
</animated-selector>
```

The top part of the **<animated-selector>** XML definition defines the states, using **<item>** tags specifying each state, and the bottom part defines your transitions, using (surprise) **<transition>** tags with **<animation-list>** tags nested inside them.

What You Will Learn from This Book

This book will focus on those features of the Android 5 operating system and the IntelliJ IDEA**,** which are used to create Android wearable apps, using Android Studio and the Wear SDK. If you require foundational Android 5 apps development knowledge or want to learn how to create a wearable application for Neptune Pine (or another smartwatch that does not use Wear SDK), take a look at my book *Android Apps for Absolute Beginners* (3rd *e*dition*,* 2014, Apress).

The first part of this book will create the foundation for the rest of the book, including this chapter covering wearable types, concepts, and the new Android 5 Material Design additions to the Android OS. Then you'll set up a development workstation, go over the wearables features of Android, and learn about the new IntelliJ IDEA. You'll also set up the emulators that will be used to test the wearable applications during the book.

The second part of the book will show you how to create wearable apps for smartwatches using the Wear SDK. You will learn about areas of Android technology that are important for wearables application developers to master, such as creating and delivering a wearable application, notifications, data layer, synchronization, and user interface layout design, using cards, and lists.

The third part of the book will explain how to create smartwatch faces using the Android Watch Faces API. You'll learn how to create a Watch Face Service, how to draw your Watch Faces to the screen, how to design Android Watch Faces, how to optimize your Watch Faces for best performance, how to display information (data) inside of your Watch Faces designs, and finally how to create Watch Faces app configuration screens for your Watch Faces.

Summary

In this first chapter, you took a look at wearables' types and concepts, and learned about the many new features that Google added to Android 5 OS. You looked at Bluetooth LE, Material Design, new Drawable types, and advanced 3D such as OpenGL ES 3.1.

In the next chapter, you'll put together your development workstation and all of the open source software that you will be able to use to develop your advanced pro Android wearables application.

Chapter 2

Setting Up an Android 5 Wearables Application Development Workstation

Now that you have some foundational knowledge about wearables and what Android 5 has added to make the wearable applications memorable, this chapter will help you put together another type of foundation. Your development workstation is the most important combination of hardware and software for reaching your goal of Pro Android Wearables application development. Here I will spend some time upfront considering the hardware you'll need and the software infrastructure that you will need to put together a professional, well-rounded, Android software development workstation with a dozen arrows in your software development quiver right off the bat (strange analogy mix isn't it? Robin Hood and baseball). Then you will have everything you need for the rest of the book, no matter what type of wearable app you develop!

We'll also get all of those tedious tasks out of the way regarding putting together a 100% professional **Pro Android Wearables production workstation**.

Because readers of this book will generally want to be developing using an identical Android Wearables Applications Software Development Environment, I will outline all of the steps in this chapter to put together a completely **decked-out** Android Studio Development Workstation. You'll need to do this because everything you will be learning over the course of this book needs to be experienced equally by all of the readers of this book. You'll learn where to download and how to install some of the most impressive open source software packages on the face of this planet!