THANKS FOR READING THE SAMPLE

Thanks for reading this sample of Step-by-step Android Wear Application Development. In order to give you a feeling for the entire book, I chose a few different sections from throughout the book to include (rather than just one chapter). Now you can see how different aspects of Android Wear application development are discussed.

If you enjoy the sample, please purchase the full book.

Thanks!

-Alex, Author

For my son, Brandon:

As promised, I finished this book before your third birthday.

You now have my full, undivided attention.

Sample Chapter

Getting Started

INTRODUCTION

For over 10 years, I have been developing firmware for handhelds in the telecommunication industry. I have always been a fan of wearable technology. As a matter of fact, James Bond and *Star Trek* were my favorite childhood viewings. I would excitedly stare at the screen, wondering how I could get my hands on the remarkable wearable devices the characters used to conquer the villains. Unfortunately, I quickly realized that the only place I could find those devices was in my dreams (or at the toy store!).

Fast-forward 30 years and wearable technologies are now expanding our capabilities to unforeseen levels. I first started experimenting with the Android Wear Developer Preview in March 2014. Back then, Google's development environment, Android Studio, was still in beta testing and the APIs in the SDK were still subject to change. In less than a year (on December 8th), Android Studio became the official integrated development environment (IDE) for Android. Having gone through all of Android's hurdles, I think this is a great time to learn Android Wear programming—the APIs have stabilized and the IDE has improved, thus giving developers a more seamless experience. If you still need more reasons, consider the growing number of major hardware manufacturers realizing the massive growth potential of the wearable market and jumping onto the bandwagon to develop fashionable and functional smartwatches. At the time of this writing, the first five smartwatches powered by Android Wear are: 1) the Samsung Gear Live, 2) the LG G Watch, 3) the LG G Watch R, 3) the Motorola moto 360, 4) the Sony SmartWatch 3, and 5) the Asus ZenWatch. In the upcoming months, two more stunning smartwatches will be released: the LG Watch Urbane and the Huawei Watch.

For your information, here's a summary of several important events worth noting:

- On March 18, 2014, Google officially released Android Wear along with the release of a developer preview to allow developers to start building apps in Android 4.4W1 (API Level 20).
- On June 25, 2014, the full Android Wear SDK was made available to developers.

- On October 21st, Google announced the first major update of Android Wear to version 4.4W2. This new firmware for Android Wear brought in offline music support, as well as some GPS functionality.
- On December 8th, 2014, Android Studio became the official IDE for Android.

WHY LEARN ANDROID WEAR NOW?

Have you ever thought to yourself: "I wish I had learned to develop iPhone apps when the iPhone first came out"? Very few people had the foresight to act on the fact that computing platforms are continuously getting smaller, cheaper, faster, and more intimate. Computers have evolved from fixed room-sized machines to desktops, laptops, and handheld devices. The same disruptive transition is about to take place again. Only this time, instead of taking your smartphone out of your pocket, the computer has become wearable. This is a great time to start learning Android Wear development. Consumers will come to embrace smartwatches because they will be relatively unobtrusive and will allow them to do things that smartphones can't. To begin, Android Wear's developer site at https://developer.android.com/wear/ has the most authoritative documentation on designing, developing, and distributing for Wear. Although you could study Google's online developer's guide to acquire the needed knowledge, you might be overwhelmed by the guide's vast amount of information. In contrast, none of this information compares to having step-by-step tutorials with working code examples on every major wearable development topic. This is exactly the kind of information this book contains—just enough theory to help you grasp the basics. And in the last chapter, you will learn how to use your newfound knowledge to build awesome real-world applications.

THE WORLD OF ANDROID WEAR

Android Wear is a version of Google's Android operating system designed for smartwatches and other wearables. It comes with a new user interface—the result of Google working to understand how we use our phones today and how we can be more in touch with our environment. According to Google, we check our phones for information roughly 150 times per day. And each time that we do that, we are likely to get distracted with other applications before we even focus on what we were looking for on the phone in the first place. This is where Android Wear comes into play. The best line to describe Android Wear is "Information that moves with you." One of the big challenges in the design of a wearable's operating system is establishing context that is relevant to what the user wants. It's obviously not enough for the watch to just tell the time; it has to simplify things by adding convenience. This could be anything from receiving timely traffic updates, stock information, weather forecasts, incoming messages, or phone calls and could even include useful health and fitness applications. Android Wear provides just the right information at the right moment by utilizing the time, your location, and your voice to reflect what is relevant to you right now. This is what Google has been betting on, along with the likes of a plethora of smartwatch OEMs. And so far, the odds are good. In just over a year, Android Wear has grown to be one million strong thanks to great workmanship from smartwatch manufacturers, passionate developers, and devoted users like you.

Having Google Maps integrated in Android Wear means you'll know which street to turn on when walking in an unfamiliar area or which bus stop to get off at based on your route. Accessing Google's voice assistant by simply raising your wrist and saying, "OK, Google," means you'll get an instant response to your query that makes sense in the moment. And with Google Now notifications at a glance, you can access the application that much more quickly and conveniently to get the right information at the right time. You could be at the airport and it'll show you the barcode for your boarding pass. You could be cooking in the kitchen and ask for the score of your favorite sports team. All of this contextual and timely information you're used to having on your phone suddenly becomes much more easily accessible with Android-Wear-powered devices.

"Everywhere you can use it, a single, continuous Android experience will be there." – Google

RESOURCES

With great opportunities come great challenges. The wearable revolution is considered the next frontier for innovation and is rapidly gaining popularity. There are a growing number of communities available where developers can find solutions to problems, learn from each other, share ideas, and exchange experiences. These communities are a great time saver and you will get information for many issues, sometimes even from Google's staff. If you are turning around in circles on a specific issue, chances are somebody has also gone through the same exercise. As time is of the essence, experienced developers will almost always turn to communities to search for help. So subscribe and be active in them!

The following are some developer communities that you should definitely check out regularly:

- The Android Wear Developers Community on Google+ is at https://plus.google.com/communities/113381227473021565406. This is a great starting place to share your thoughts and discuss Android Wear design and development topics.
- Stack Overflow is a forum that has lots of questions and answers from professional programmers concerning common Android Wear development issues. The site is at http://stackoverflow.com/questions/tagged/android-wear.
- The Android Developers Blog at http://android-developers.blogspot.ca has the latest news along with great programming tips and best practices from Google's talented staff, developer advocates, and product managers.

BEFORE YOU GET STARTED

As mentioned previously, Android Wear's developer site at https://developer.android.com/wear/ is your primary reference for designing, developing, and distributing Wear. You may also want to check out the following Android-Wear-related session at Google I/O 2014 entitled, "Android Wear: The developer's perspective":

https://www.google.com/events/io/schedule/session/9bf77f55-afbe-e311-b297-00155d5066d7

I also recommend checking out the following Android Wear DevBytes:

- Receiving Voice Replies
  https://www.youtube.com/watch?v=SEZbZK4jFLY&list=PLWz5rJ2EKKc-kIrPiq098QH9dOle-fLef&index=12
- Page Notifications
  https://www.youtube.com/watch?v=N7aJPyvHPgs&list=PLWz5rJ2EKKc-kIrPiq098QH9dOle-fLef&index=11
- Stackable Notifications
  https://www.youtube.com/watch?v=L4LvKOTkZ7Q&list=PLWz5rJ2EKKc-kIrPiq098QH9dOle-fLef&index=10
- Building Advanced UIs for Android Wear
  https://www.youtube.com/watch?v=uTQ9KKyM9wo&list=PLWz5rJ2EKKc-kIrPiq098QH9dOle-fLef&index=6
- Fullscreen Apps for Android Wear
  https://www.youtube.com/watch?v=naf_WbtFAlY&index=5&list=PLWz5rJ2EKKc-kIrPiq098QH9dOle-fLef
- New Notification Features for Android Wear
  https://www.youtube.com/watch?v=tKoQatxG0_8&index=4&list=PLWz5rJ2EKKc-kIrPiq098QH9dOle-fLef
- Wearable DataLayer API
  https://www.youtube.com/watch?v=2zReOOEUP18&index=3&list=PLWz5rJ2EKKc-kIrPiq098QH9dOle-fLef

ERRATA

My editor and I have made every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error, such as a spelling mistake or a faulty piece of code, we would be very grateful for your feedback. By sending in errata, you may save another reader hours of frustration and at the same time help us provide even higher-quality and more up-to-date information.

How to Reach Me

I can be easily reached at aho @learnandroidwear.com. You should also regularly check out my site at http://www.learnandroidwear.com for errata, additional content, future notifications, and news on future projects. Writing this book has been a real delight. I hope you enjoy this book and I am confident that after reading it, you'll be well versed in, and comfortable with, developing apps for Android Wear.

ANDROID STUDIO IDE

As you know by now, Android Studio is the official, Google-backed development environment for Android. Based off of IntelliJ IDEA, Android Studio is the foundation for the next-generation build system for Android apps. It gives developers top-notch development tools and comes with the following features:

- More Focused/Effective UI and Workspace: Since Android Studio is designed for Android development, it does not carry all the legacy and cross-platform baggage of Eclipse. Moreover, it provides a comprehensive sample code base that developers can easily import into their projects and Android Studio's installer also sets up the most up-to-date SDK.
- Code Editor: The IDE gives smart and relevant suggestions in every context, such as instant code completion, real-time code analysis, easy project navigation, and dependable refactoring tools.
- Dynamic Layout Preview: Provides the ability to visualize apps in multiple screen form factors with UI widgets and drag-and-drop functionality.
- Google Cloud Platform: Android Studio allows you to easily add cloud-based backends to your Android apps. With just a few clicks, it can generate the required code and dependencies to your project to quickly get you up and running.

Installing Android Studio on Windows

1. First, check if you have the Java Development Kit (JDK) version 6.0 or greater already installed (JRE alone is not sufficient).
2. To check if you have the JDK installed, open a terminal and type java -version. If you have JDK version 6.0 or above, continue to step 4.
3. Visit the JAVA install page. Download and install the latest JDK.
4. Set your JAVA_HOME path. Select Computer from your Start Menu, then select System Properties → Advanced system settings → Environment variables.
5. Click New under System Variables. Enter JAVA_HOME in the Variable name text box, and the path to your Java installation directory in the Variable value text box. It should look something like this:

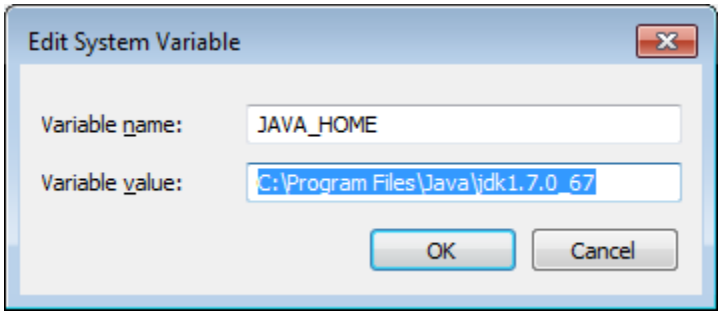


Figure 1-01

6. Visit the Android Studio download page, and download the self-installing EXE file for your platform. Your operating system will automatically be detected. Double-click the downloaded file and follow the self-guided instructions.
7. Once installed, open Android Studio and click on Configure followed by SDK Manager.

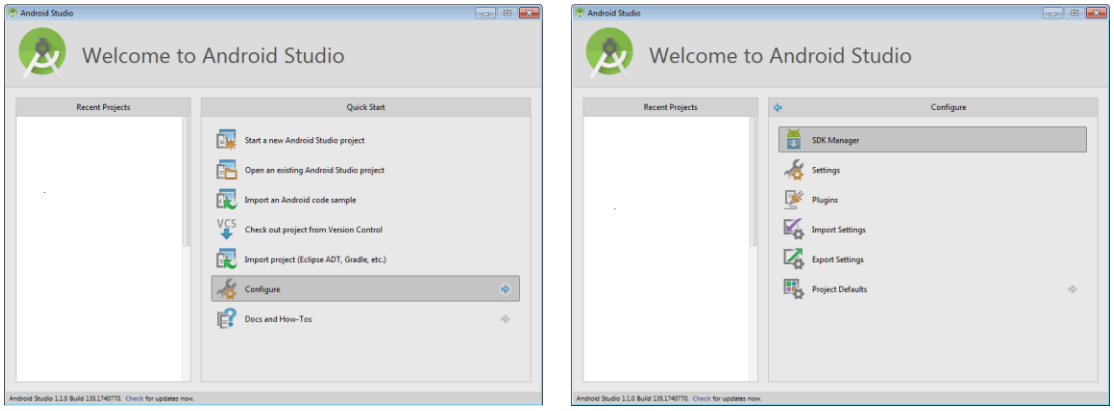


Figure 1-02

8. We are building for Android 5.0.1 (API 21). Check the following packages under the Tools section:

- Android SDK Tools
- Android SDK Platform-Tools
- Android SDK Build-Tools

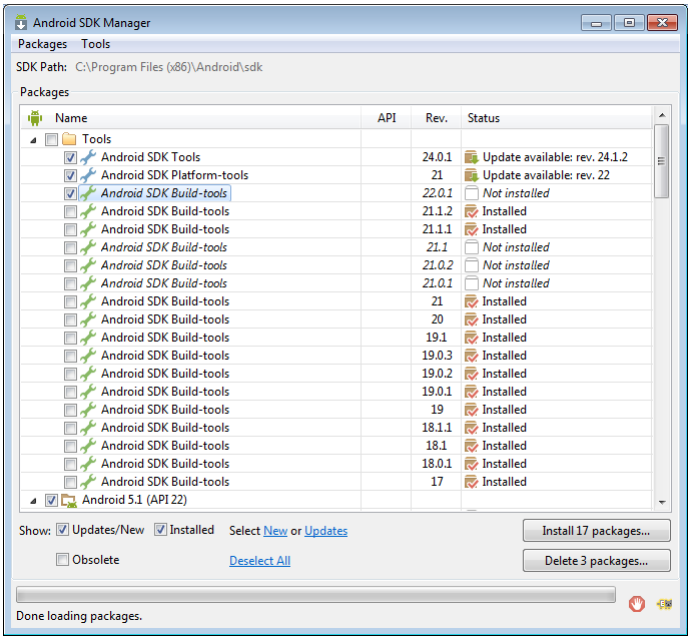


Figure 1-03

9. Check the following packages under the Android 5.0.1 (API 21) section:

- Android Wear Intel x86 Atom System Image
- Google APIs (x86 System Image)

10. Under the Extras section, check the following packages:

- Android Support Repository
- Android Support Library
- Google USB Driver

11. Click Install and accept the licenses that appear to proceed with the installation.

Creating an Android Virtual Device (AVD)

1. Open Android Studio. Click on the AVD Manager in the toolbar. An AVD lets you prototype, test, and run your apps on an emulator.

Figure 1-04

2. In the Android Virtual Device Manager window, select Wear under the Category pane. Android Wear comes in two screen configurations: round and square. Select the form factor of the device you want to create for and click Next. You should set up one emulator for each form factor to ensure that your app looks good and works correctly.

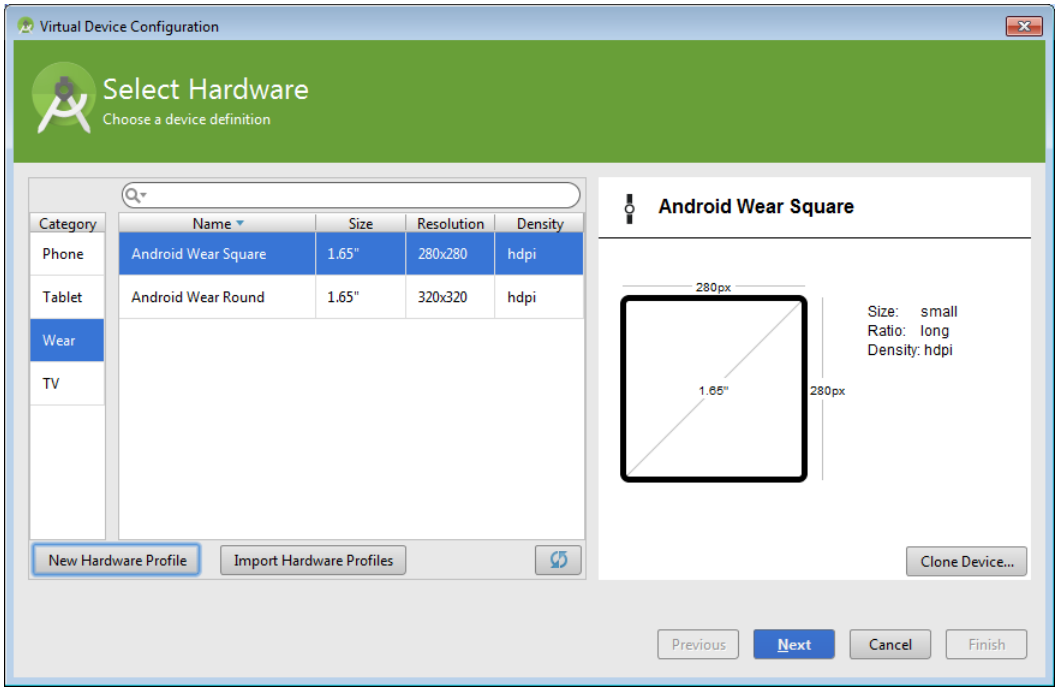


Figure 1-05

3. Select Lollipop (API 21) for the x86 platform, then click Next.

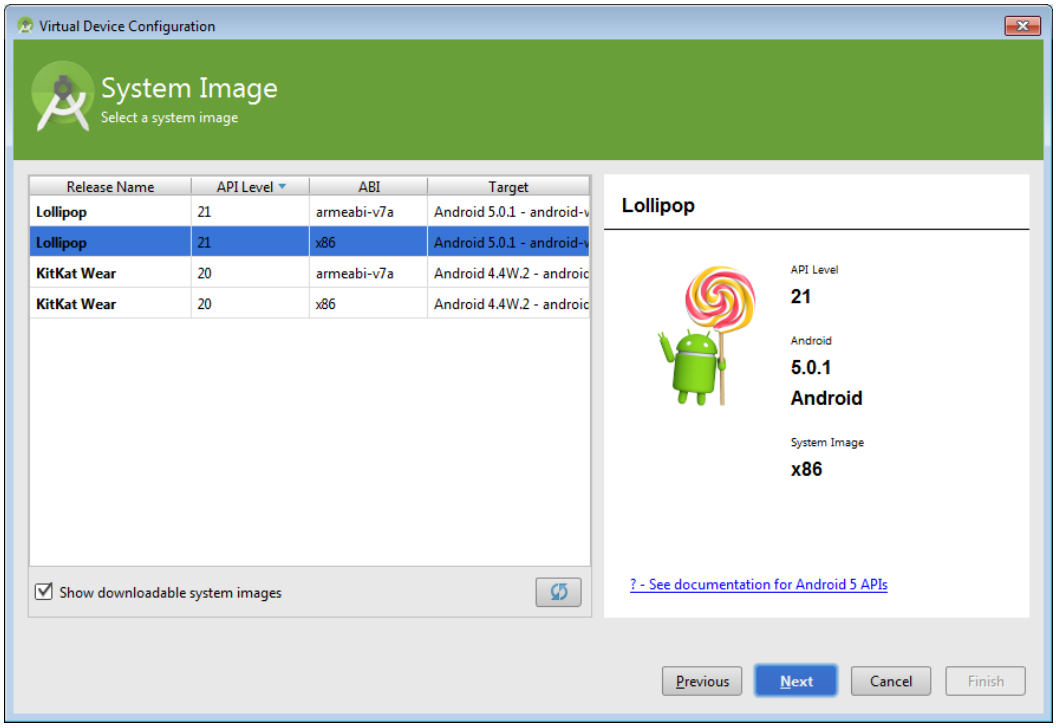


Figure 1-06

4. Key in a meaningful AVD name for your new device, then click Finish.

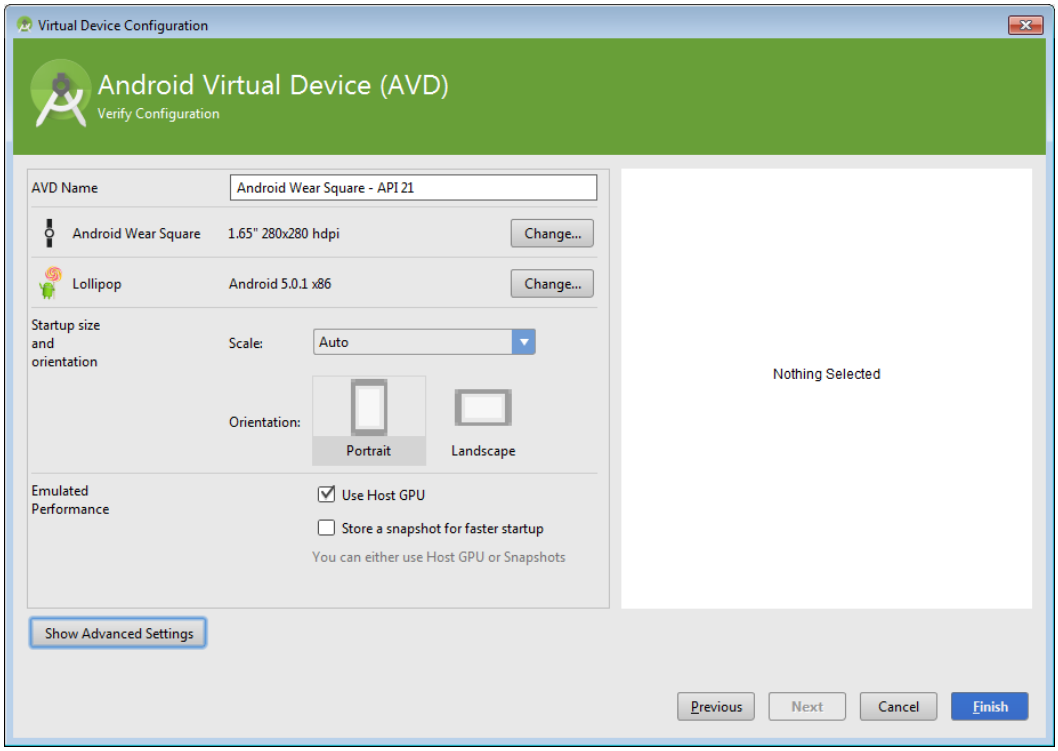


Figure 1-07

5. You should now see the new device you created in the AVD manager, where you can start, modify or delete it.

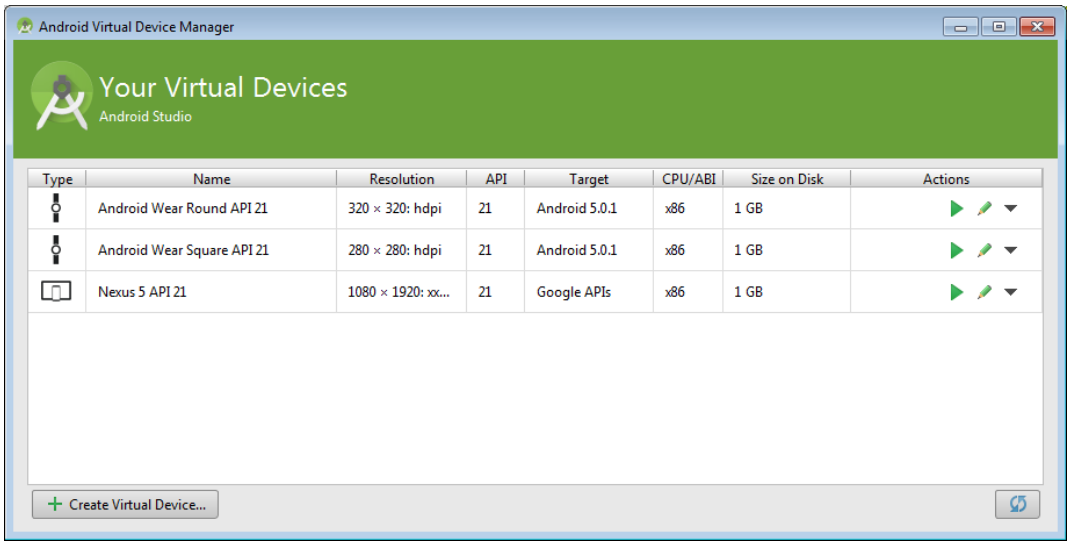


Figure 1-08

YOUR FIRST ANDROID WEAR APPLICATION

With Android Studio downloaded and installed, it is now time to create your first program. As in all programming books, let's give credit to Brian Kernighan and create your first Hello World program. The following sections will give you a detailed overview of the various components that make up an Android Wear project.

STEP BY STEP Creating Your First Hello World Application

1. Open Android Studio and create a new project by clicking on "Start a new Android Studio Project".

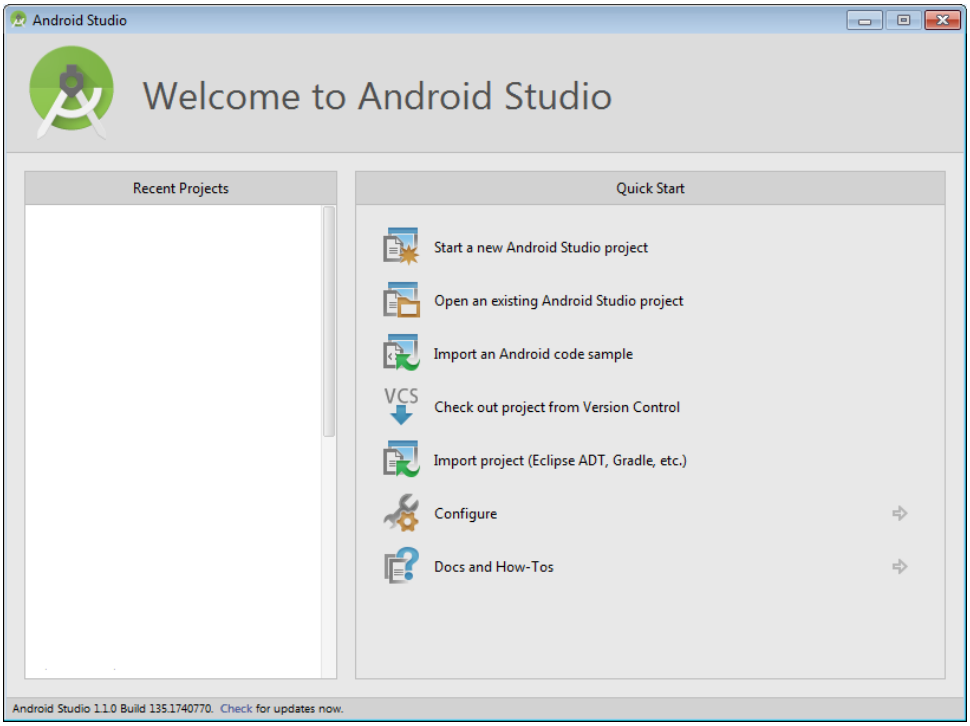


Figure 1-12

2. In the Create New project window, name the application "HelloWorld" and key in a company domain. For simplicity, just use learnandroidwear.com because all examples in this book use this domain name. Click Next.

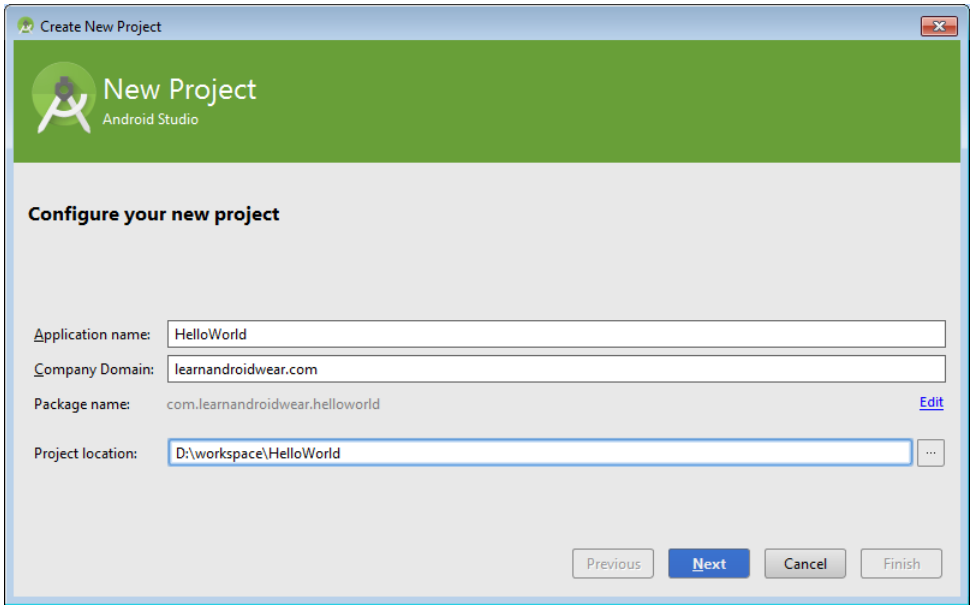


Figure 1-13

NOTE *The recommended convention for the package name is to use your domain name in reverse order, followed by the project name. In my case, my website's domain name is* learnandroidwear.com*; hence, my package name would be* com.learnandroidwear.HelloWorld. *The Java Language Specification establishes package naming conventions to avoid the possibility of two published packages having the same name. The naming conventions describe how to create unique package names, so that packages that are widely distributed will have unique namespaces. This allows packages to be separately, easily, and automatically installed and cataloged.*

3. Click on the Wear platform and select API 21 as the Minimum SDK. Click Next.

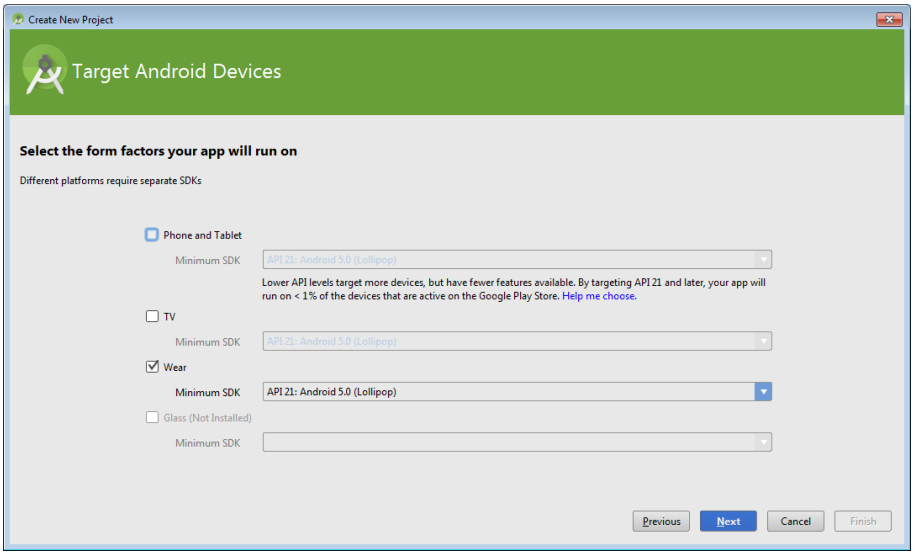


Figure 1-14

4. Select Blank Wear Activity then click Next.

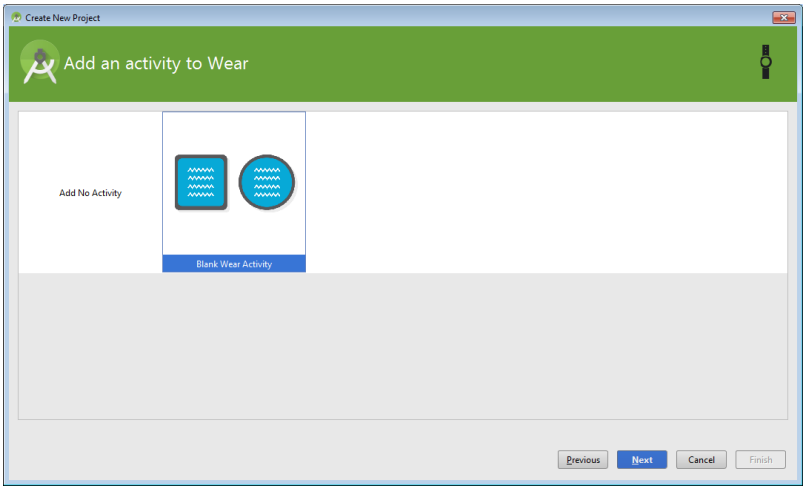


Figure 1-15

5. Fill in the application details for the wearable activity as shown in Figure 1-16. Click Finish.

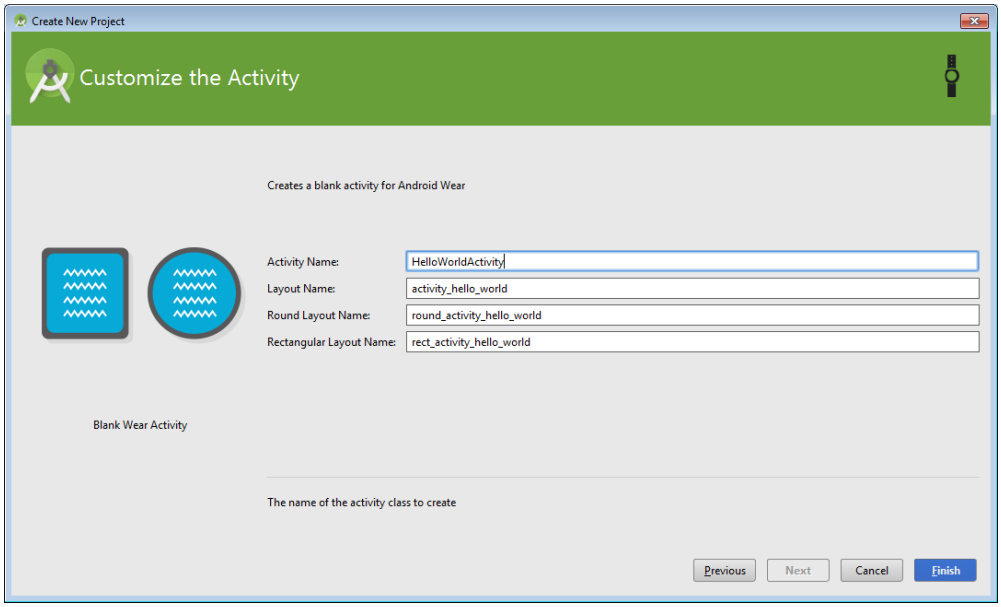


Figure 1-16

6. The Android Studio IDE should now look like Figure 1-17.

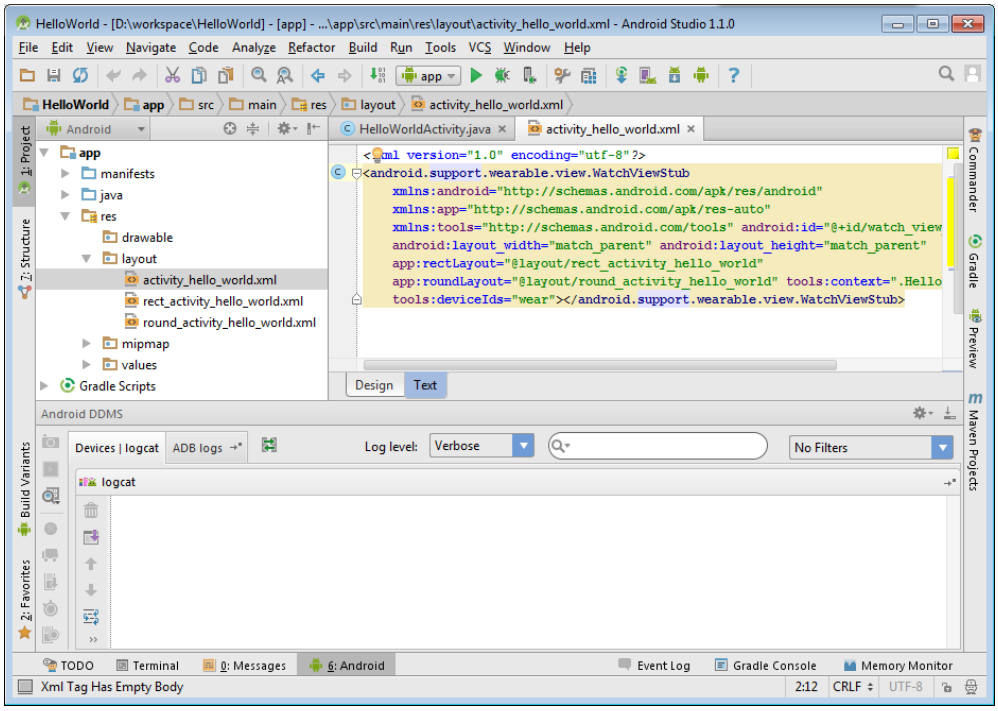


Figure 1-17

7. Launch your Android Wear square or round AVD, then press Shift+F9 to debug the project.

Don’t worry if you don’t grasp some of the concepts or code in the preceding exercise. In the following chapters, you will see a lot of examples about every development topic to get you well versed with Android Wear development. What’s important is that you have successfully installed and configured your development environment, created a working app, and are now well prepared to explore the exciting world of Android Wear.

Sample Chapter

User Interface Essentials

TOPICS COVERED IN THIS CHAPTER

- Using the new Wearable UI Library
- UI Design Guidelines
- How to manage round and square screens
- Working with CardFragments, CardFrames, and CardScrollView
- How to display confirmation activity to provide feedback
- How to use the DelayedConfirmationView
- How to use the GridViewPager and WearableListView

In this chapter, you will learn the details about creating user interfaces in Android Wear. You will learn about the general guidelines for building engaging and dynamic user interfaces that are relevant to what the user wants yet facilitate the quick and lightweight interactions that are essential to wearables. With the introduction of Android Wear, Google added a UI library optimized for wearables, which ensures a consistent user experience across apps.

The following table shows you the major classes that will be covered in this chapter:

Class	Description
BoxInsetLayout	A FrameLayout defining a single layout that works for both square and round screens.
CardFragment	A fragment that presents content within an expandable, vertically scrollable card that has a consistent look and feel across the platform. At a minimum, it contains a title, a description, and an optional icon. If you need a customized layout, override the onCreateContentView() method.
CardScrollView	A special type of FrameLayout designed to host one CardFrame and provide support for scrolling.
CircledImageView	An ImageView surrounded by a circle. You can include a drawable image in the center of the circle or an optional outer border to show progress. This widget is widely use throughout the Android Wear ecosystem. The settings page is a good example of this widget in use.
ConfirmationActivity	An activity that displays confirmation animations to provide feedback about whether an action has succeeded or failed.
CrossFadeDrawable	An extension of the Drawable abstraction layer that contains two drawables and lets you adjust various visual effects between the two.
DelayedConfirmationView	A subclass of the CircledImageView that provides a circular countdown timer to let users cancel an action they have just performed.
DismissOverlayView	A simple view for implementing the long press method to dismiss an app. It is typically used in combination with the setIntroText() method to inform users that they can exit the app using the long press gesture. To display the introduction text, call showIntroIfNecessary() in your onCreate() method.
DotsPageIndicator	Displays an indicator dot for each page in a GridViewPager (from left to right, the dots represent the order in which the views were opened).
FragmentGridPagerAdapter	An adapter class that handles fragments and acts as a bridge between the GridViewPager and the data source that feeds data into it.
GridPagerAdapter	Another adapter class responsible for providing views requested by a GridViewPager (instead of fragments).
GridViewPager	A two-dimensional scrolling grid of items that allows users to scroll vertically or horizontally through pages of data.
WatchViewStub	Used to detect the screen shape at runtime and inflate a square or round layout.
WearableListView	Use this class to display and select vertically scrollable items that are optimized for ease of use on wearable devices. When the user stops scrolling, this class automatically focuses on the nearest item.
WearableListView.Adapter	An adapter for the WearableListView responsible for inflating the correct layout for individual list items and for displaying the data onto the list view. In essence, it acts as a bridge between the XML layout and data.

Table 2-1

UI DESIGN GUIDELINES

Designing and developing applications for wearables involves a different mindset. A common mistake is to port your existing handheld app to a wearable device; this will certainly not yield a very good user experience. Android Wear devices have limited user interaction capabilities. For example, they don't have keyboards that allow users to enter data. Instead, Android wearables are intended for micro-interactions. For example, you tap to select, swipe to dismiss, or speak to reply. There is one action for each interaction. Do not take design lightly, especially design for wearables. With their smaller screens, good design is not just what looks good, it needs to be consistent, fulfill its purpose, and address the needs of users who are on the go.

The Android Wear developer site at https://developer.android.com/design/wear/index.html has great documentation on a number of general design guidelines that you should follow. Here is a summary of these guidelines (with my own variations):

- *Keep things consistent*: Use consistent design patterns that users are already accustomed to. For example, a CardFragment presents information within an expandable, vertically scrollable card that has a consistent look and feel across the platform. When users sees a card, they already know how to use it.
- *To scroll or not to scroll*: Wearable screens are small, which requires developers to use certain tricks to present a lot of information in such a small screen form factor. One method is to use card expansion, so that only part of the information is visible at one time and the rest is available via scrolling up or down. Be mindful of the amount of scrollable content you include by setting a reasonable expansion factor to prevent the user from having to scroll through exceptionally long cards that result from unexpectedly large content (e.g., web search results).
- *A good design is unobtrusive*: Because wearables are worn so close to the body, a good design should show respect toward the user. For example, be sensitive to the frequency with which you send notifications. A user doesn't need to be notified each time his favorite sports team scores. In the same manner, do not constantly demand the user's attention with notifications that cause the device to vibrate unnecessarily. As a rule of thumb, vibrate when notifications are personal and timely (e.g., when receiving a message or when a flight schedule changes), otherwise add non-urgent notifications silently to the ContextStream.
- *Contextual awareness*: A good design is attuned to the user's presence. Avoid showing stale or irrelevant information. Instead, exploit time, location, and orientation to reflect what is relevant to the user right now.
- *Less, but better (or less is more)*: Long messages on cards should be cropped appropriately to focus on essential aspects, ensuring that useful information is conveyed at a glance. In the same manner, if your app creates notifications for received messages and more than one message is received, use stacked notifications to group similar notifications in a card stack.
- *User-oriented design*: A good design is based on its use and is designed to teach and facilitate a given situation for its user. For example, when implementing the long-press-to-dismiss UI pattern using the DismissOverlayView class, you must also inform the users that they can exit your app using a long press. Additionally, when implementing a reply

action in a notification, ensure that the list of predefined responses is adequate and relevant.

- *Design for brief interactions*: Interactions with a Wear app should be brief—users should interact with the device for no more than a few seconds. Otherwise, consider simplifying and making your app more focused.
- *Engaging animations*: Care in the design process shows respect toward the user. Relying on UI patterns that are ready at hand, such as a confirmation animation, can make an app experience more engaging and dynamic for the user. Animating the confirmation icon can provide valuable feedback and help users visualize the results of their actions.

GRIDVIEWPAGER

As the name suggests, GridViewPager gives you a two-dimensional scrolling grid of items to choose from. It is a layout manager that allows users to scroll vertically and horizontally through pages of data. You can combine different types of pages such as a CardFragment, an ImageView, action icons and custom layouts depending on your use cases. The number of rows and columns can be asymmetric; not all rows needs to have the same number of pages. As such, you can implement a one-dimensional layout with only one row or only one column. Furthermore, implementing a dots indicator will give users a visual cue as to the number of pages in the context stream. For added visual appeal, each page may have a background image that will automatically perform parallax effects when scrolling between pages. Otherwise, the GridViewPager works much like any other selection widget — use setAdapter() to provide the data and child views, invoke setOnPageChangeListener() to register a page change listener, and so on.

INFO *In the context of Android Wear, each row represents a single piece of content. When moving between rows, GridViewPager will by default navigate back to the first column. This behavior is seamless and expected. For example, if you are viewing the details of a stock index or the weather forecast, it would be out of context and outright confusing to land on an action button for a different item. However, depending on your use cases, you can change this behavior by overriding the* getCurrentColumnForRow() *method.*

There are two ways to implement a GridViewPager:

- Without fragment, by extending the GridPagerAdapter and providing a custom View object
- By assigning a fragment such as a CardFragment for each page and extending the FragmentGridPagerAdapter

Using the GridViewPager without Fragments

In the following Step-by-Step section, you will learn how to implement a GridViewPager without fragments. That is, you will create a 3 by 3 car gallery using the GridViewPager, GridPagerAdapter, ImageView, and DotsPageIndicator as shown in Figure 2-15.

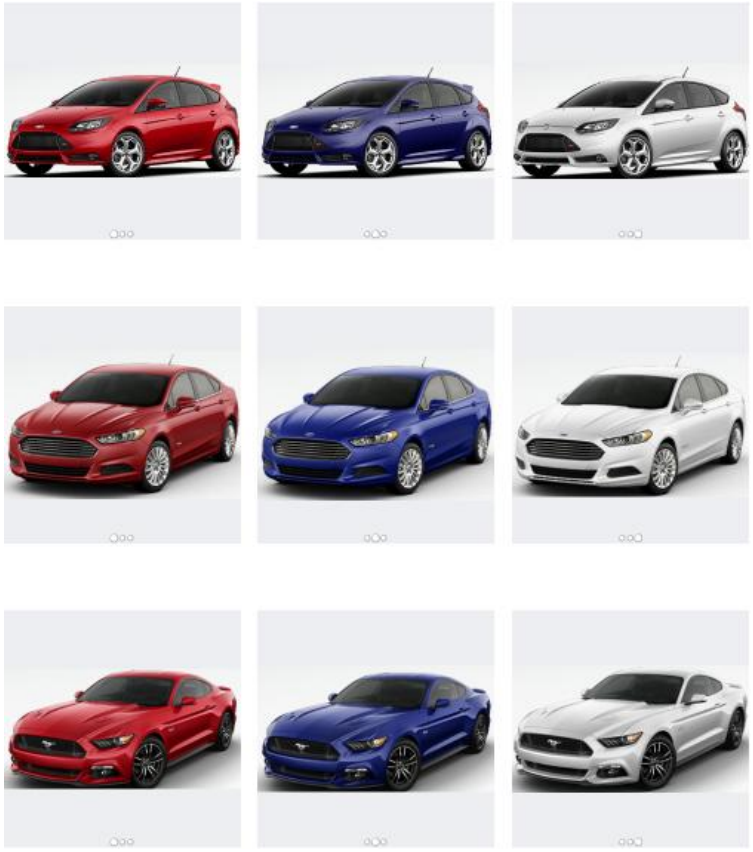


Figure 2-15

STEP BY STEP Using the GridViewPager (without Fragments)

1. Using Android Studio, create a new blank project for the Wear platform and name it SimpleGridView.
2. Select API 21 as the minimum SDK level, then click Next.
3. In the mobile activity dialog, select "Add no Activity", then click Finish.
4. Unzip images.zip into a temporary folder. Go into the CarImages folder and perform a select all (Ctrl-A) followed by a copy (Ctrl-C).
5. In your Android Studio project pane, go to the res folder and select drawable as shown below. Right-click on drawable then choose paste.

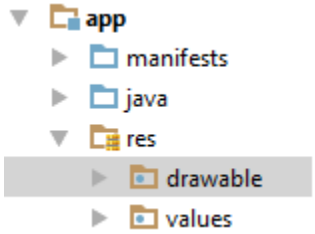


Figure 2-16

6. If the layout folder in the project tree does not exist, right-click on the res folder, then click on New → Directory and name the folder "layout".
7. Add a new file in the res/layout folder and name it activity_main. Key in Framelayout in the Root Tag field. Populate it as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <android.support.wearable.view.GridViewPager
        android:id="@+id/pager"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:keepScreenOn="true" />

    <android.support.wearable.view.DotsPageIndicator
        android:id="@+id/page_indicator"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center_horizontal|bottom">
    </android.support.wearable.view.DotsPageIndicator>

</FrameLayout>
```

8. Add a new Class file and name it SimpleGridActivity. Populate it with the following content:

```
package com.learnandroidwear.simplegridview;

import android.app.Activity;
import android.content.Context;
import android.content.res.Resources;
import android.graphics.Color;
import android.os.Bundle;
import android.support.wearable.view.DotsPageIndicator;
import android.support.wearable.view.GridPagerAdapter;
import android.support.wearable.view.GridViewPager;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;

public class SimpleGridActivity extends Activity {

    // The images to display
    // Here we use a 2D array for row/column indexing
    Integer[][] carImageIDs = {
        {R.drawable.bg_red_focus,
         R.drawable.bg_blue_focus,
         R.drawable.bg_white_focus},
        {R.drawable.bg_red_fusion,
         R.drawable.bg_blue_fusion,
         R.drawable.bg_white_fusion},
        {R.drawable.bg_red_mustang,
         R.drawable.bg_blue_mustang,
         R.drawable.bg_white_mustang}
    };
```

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    final Resources res = getResources();
    final GridViewPager pager = (GridViewPager) findViewById(R.id.pager);

    // Assigns an adapter to provide the content for this pager---
    pager.setAdapter(new ImageAdapter(this));
    DotsPageIndicator dotsPageIndicator = (DotsPageIndicator)
        findViewById(R.id.page_indicator);
    dotsPageIndicator.setPager(pager);
}

public class ImageAdapter extends GridPagerAdapter {
    final Context mContext;

    public ImageAdapter(final Context context) {
        mContext = context;
    }

    @Override
    public int getRowCount() {
        return 3;
    }

    @Override
    public int getColumnCount(int i) {
        return 3;
    }

    // Go to current column when scrolling up or down
    // (instead of default column 0)
    @Override
    public int getCurrentColumnForRow(int row, int currentColumn) {
        return currentColumn;
    }

    // Return our car image based on the provided row and column
    @Override
    protected Object instantiateItem(ViewGroup viewGroup, int row, int col) {
        ImageView imageView;
        imageView = new ImageView(mContext);
        imageView.setImageResource(carImageIDs[row][col]);
        imageView.setBackgroundColor(Color.rgb(236, 238, 242));
        viewGroup.addView(imageView);
        return imageView;
    }

    @Override
    protected void destroyItem(ViewGroup viewGroup, int i, int i2, Object o) {
        viewGroup.removeView((View) o);
    }

    @Override
    public boolean isViewFromObject(View view, Object o) {
        return view.equals(o);
    }
```

```
    }
}
```

9. In the AndroidManifest.xml, add the following entries in bold:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.learnandroidwear.simplegridview">
    <uses-feature android:name="android.hardware.type.watch" />
    <application android:allowBackup="true"
        android:label="@string/app_name"
        android:icon="@drawable/ic_launcher"
        android:theme="@android:style/Theme.DeviceDefault">

        <activity
            android:name="com.learnandroidwear.simplegridview.SimpleGridActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

    </application>

</manifest>
```

10. Press Shift+F9 to debug the project on a wearable device or emulator.

Understanding the Code

You first added the GridViewPager and DotsPageIndicator views to activity_main.xml.

```
<android.support.wearable.view.GridViewPager
        android:id="@+id/pager"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:keepScreenOn="true" />

<android.support.wearable.view.DotsPageIndicator
    android:id="@+id/page_indicator"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal|bottom">
```

As mentioned earlier, the GridViewPager is used to display content data in a horizontal and/or vertical scrolling list. The DotsPageIndicator is used to display a visual cue regarding the number of pages in the context stream.

The list of images to be displayed is stored in the carImageIDs 2D array:

```
//---The images to display---
Integer[][] carImageIDs =
    {
        {R.drawable.bg_red_focus,
         R.drawable.bg_blue_focus,
         R.drawable.bg_white_focus},
```

```
            {R.drawable.bg_red_fusion,
             R.drawable.bg_blue_fusion,
             R.drawable.bg_white_fusion},
            {R.drawable.bg_red_mustang,
             R.drawable.bg_blue_mustang,
             R.drawable.bg_white_mustang}
        };
```

You create the ImageAdapter class (which extends the GridPagerAdapter class) so that it can bind to the GridViewPager view with a series of ImageView views. The GridPagerAdapter class acts as a bridge between the GridViewPager and the data source that feeds data into it.

For the ImageAdapter class, you implemented the following methods:

```
public class ImageAdapter extends GridPagerAdapter {
    public ImageAdapter(final Context context) { ... }

    //Returns the number of rows in our car gallery
    public int getRowCount() { ... }

    //Returns the number of columns in our car gallery
    public int getColumnCount(int i) { ... }

    //Go to current column when scrolling up or down
    // (instead of defaulting to column 0)---
    public int getCurrentColumnForRow(int row, int currentColumn) { ... }

    protected Object instantiateItem(ViewGroup viewGroup, int row, int col) { ... }
}
```

In particular, the instantiateItem() method returns an ImageView object by referencing the carImageIDs array by its row and column.

In our example, we are using a GridViewPager with 9 pages (3 columns and 3 rows) therefore getRowCount() and getColumnCount() both return 3 items, respectively.

Finally, the getCurrentColumnForRow() method is an interesting method. Recall earlier that the default behavior of the GridViewPager is to go back to the first column whenever you scroll up or down from a page. However, in our example we want to remain in the same column when moving between rows. This makes sense in the context of our use case because if the user is looking at a blue Ford Focus, scrolling down will display a blue Ford Fusion and so forth. Overriding the getCurrentColumnForRow() method and simply returning to the currentColumn allows you to achieve this.

CREATING LISTS

One of the most commonly used views in Android Wear is the list view. A list view is used to display a set of choices from which the user can select by tapping on the item. In Android Wear, the Wearable UI library includes the WearableListView class, which lets users scroll up and down and select through a vertical list of items that are optimized for ease of use on wearable devices. For example, tapping on an off-centered item will snap it on the center of the screen and a single

tap selects it. For added convenience on small screens, this class automatically snaps the nearest item in the center of the screen when the user stops scrolling. Finally, the WearableListView can be used to display a minimal, text-only scrolling list of items or can be extensively customized and adapted to your needs by adding a header, icons, by using colors, and various animated and scaling effects while scrolling through the list.

HINT *You should not use the traditional* ListView *with Android Wear since it will not yield a very good user experience — it makes it difficult for users to select individual list items on small screens. Instead, use the WearableListView.*

Custom Wearable List View

The best way to understand how to use the WearableListView is to obtain a reference of it from the XML layout, wire it to an instance of the adapter to populate the list, and setting a click listener to trigger an action when the user selects a particular list item.

In the following step-by-step section, you will learn how to create a custom wearable list by implementing the WearableListView.OnCenterProximityListener interface to display a title, circular icons, and to animate list items and text fade-ins as the user scrolls through the list.

Figure 2-16

STEP BY STEP Custom Wearable List View

1. Using Android Studio, create a new Android project for the Wear platform and name it CustomList.

2. Add a new XML file in the res/layout folder in the Mobile branch and name it activity_main. Key in BoxInsetLayout in the Root Tag field. Populate it as follows:

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<android.support.wearable.view.BoxInsetLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_height="match_parent"
    android:layout_width="match_parent">

    <RelativeLayout
        android:id="@+id/frame_layout"
        android:layout_height="match_parent"
        android:layout_width="match_parent"
        app:layout_box="left|bottom|right">

        <android.support.wearable.view.WearableListView
            android:id="@+id/wearable_List"
            android:layout_height="match_parent"
            android:layout_width="match_parent"
            android:background="#434343"
            android:scrollbars="none"
            android:dividerHeight="0dp">
        </android.support.wearable.view.WearableListView>

        <TextView
            android:id="@+id/header"
            android:layout_width="wrap_content"
            android:layout_height="60dp"
            android:layout_centerHorizontal="true"
            android:gravity="bottom"
            android:textSize="20sp"
            android:fontFamily="sans-serif-light"
            android:text="Settings"/>

    </RelativeLayout>
</android.support.wearable.view.BoxInsetLayout>
```

3. Add a new XML file in res/layout folder and name it list_item. Populate it as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<com.learnandroidwear.customwearablelistview.WearableListItemLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:gravity="center_vertical"
    android:layout_width="match_parent"
    android:layout_height="80dip">

    <android.support.wearable.view.CircledImageView
        android:id="@+id/circle"
        app:circle_color="#c1c1c1"
        android:layout_gravity="center_vertical"
        android:layout_marginStart="32dp"
        android:layout_marginEnd="32dp"
        android:layout_width="20dip"
        android:layout_height="20dip"
        app:circle_border_color="#FFFFFFFF"
        app:circle_border_width="2dp" />

    <TextView
        android:id="@+id/name"
        android:alpha="0.5"
```

```
        android:gravity="center_vertical"
        android:fontFamily="sans-serif-condensed-light"
        android:textColor="@color/white"
        android:textSize="16sp"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"/>

</com.learnandroidwear.customwearablelistview.WearableListItemLayout>
```

4. Add a new XML file in res/values folder and name it dimen. Key in merge in the Root Tag field. Populate it as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <dimen name="small_circle_radius">18dp</dimen>
    <dimen name="big_circle_radius">24dp</dimen>
</resources>
```

5. Add a new Class file and name it WearableListItemLayout. Populate it with the following content:

```
package com.learnandroidwear.customwearablelistview;

import android.content.Context;
import android.graphics.Color;
import android.support.wearable.view.CircledImageView;
import android.support.wearable.view.WearableListView;
import android.util.AttributeSet;
import android.widget.LinearLayout;
import android.widget.TextView;

public class WearableListItemLayout extends LinearLayout
        implements WearableListView.OnCenterProximityListener {

    private CircledImageView mCircle;
    private TextView mName;
    private final float mFadedTextAlpha;
    private final int mUnselectedCircleColor, mSelectedCircleColor;
    private float mBigCircleRadius;
    private float mSmallCircleRadius;

    public WearableListItemLayout(Context context) {
        this(context, null);
    }

    public WearableListItemLayout(Context context, AttributeSet attrs) {
        this(context, attrs, 0);
    }

    public WearableListItemLayout(Context context, AttributeSet attrs,
                                  int defStyle) {
        super(context, attrs, defStyle);

        mFadedTextAlpha = 40 / 100f;
        mUnselectedCircleColor = Color.parseColor("#434343");
        mSelectedCircleColor = Color.parseColor("#434343");
        mSmallCircleRadius = getResources().
```

```
                getDimensionPixelSize(R.dimen.small_circle_radius);
        mBigCircleRadius = getResources().
                getDimensionPixelSize(R.dimen.big_circle_radius);

        // When expanded, the circle may extend beyond the bounds of the view
        setClipChildren(false);
    }

    @Override
    protected void onFinishInflate() {
        super.onFinishInflate();
        mCircle = (CircledImageView) findViewById(R.id.circle);
        mName = (TextView) findViewById(R.id.name);
    }

    @Override
    public void onCenterPosition(boolean animate) {
        mName.setAlpha(1f);
        mCircle.setAlpha(1f);
        mCircle.setCircleColor(mSelectedCircleColor);
        mCircle.setCircleRadius(mBigCircleRadius);
    }

    @Override
    public void onNonCenterPosition(boolean animate) {
        mName.setAlpha(mFadedTextAlpha);
        mCircle.setAlpha(mFadedTextAlpha);
        mCircle.setCircleColor(mUnselectedCircleColor);
        mCircle.setCircleRadius(mSmallCircleRadius);
    }
}
```

6. Add a new Class file and name it WearableAdapter. Populate it with the following content:

```
package com.learnandroidwear.customwearablelistview;

import android.content.Context;
import android.support.wearable.view.CircledImageView;
import android.support.wearable.view.WearableListView;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.TextView;

import java.util.ArrayList;

public class WearableAdapter extends WearableListView.Adapter {
    private ArrayList<Integer> mItems;
    private final LayoutInflater mInflater;

    public WearableAdapter(Context context, ArrayList<Integer> items) {
        mInflater = LayoutInflater.from(context);
        mItems = items;
    }

    @Override
    public WearableListView.ViewHolder onCreateViewHolder(
            ViewGroup viewGroup, int i) {
        return new ItemViewHolder(mInflater.inflate(R.layout.list_item, null));
```

```
    }

    @Override
    public void onBindViewHolder(WearableListView.ViewHolder viewHolder,
                                 int position) {
        ItemViewHolder itemViewHolder = (ItemViewHolder) viewHolder;
        TextView textView = itemViewHolder.mItemTextView;
        textView.setText(String.format("Item %d", position + 1));
        CircledImageView circledView = itemViewHolder.mCircledImageView;
        circledView.setImageResource(mItems.get(position));
    }

    @Override
    public int getItemCount() {
        return mItems.size();
    }

    private static class ItemViewHolder extends WearableListView.ViewHolder {
        private CircledImageView mCircledImageView;
        private TextView mItemTextView;

        public ItemViewHolder(View itemView) {
            super(itemView);
            mItemTextView = (TextView) itemView.findViewById(R.id.name);
            mCircledImageView = (CircledImageView)
                    itemView.findViewById(R.id.circle);
        }
    }
}
```

7. Add a new Class file and name it CustomWearableList. Populate it with the following content:

```
package com.learnandroidwear.customwearablelistview;

import android.app.Activity;
import android.os.Bundle;
import android.support.wearable.view.WearableListView;
import android.widget.TextView;
import android.widget.Toast;
import java.util.ArrayList;

public class CustomWearableList extends Activity {
    private static ArrayList<Integer> mIcons;
    private TextView mHeader;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        // Sample image set for the list
        mIcons = new ArrayList<Integer>();
        mIcons.add(R.drawable.ic_action_attach);
        mIcons.add(R.drawable.ic_action_call);
        mIcons.add(R.drawable.ic_action_locate);
        mIcons.add(R.drawable.ic_action_mail);
        mIcons.add(R.drawable.ic_action_microphone);
        mIcons.add(R.drawable.ic_action_photo);
        mIcons.add(R.drawable.ic_action_star);
```

```
        mIcons.add(R.drawable.ic_action_user);
        mIcons.add(R.drawable.ic_action_video);

        // This is our list header
        mHeader = (TextView) findViewById(R.id.header);

        WearableListView wearableListView =
                (WearableListView) findViewById(R.id.wearable_List);
        wearableListView.setAdapter(new WearableAdapter(this, mIcons));
        wearableListView.setClickListener(mClickListener);
        wearableListView.addOnScrollListener(mOnScrollListener);
    }

    // Handle our Wearable List's click events
    private WearableListView.ClickListener mClickListener =
            new WearableListView.ClickListener() {
        @Override
        public void onClick(WearableListView.ViewHolder viewHolder) {
            Toast.makeText(CustomWearableList.this,
                    String.format("You selected item #%s",
                    viewHolder.getLayoutPosition()+1),
                    Toast.LENGTH_SHORT).show();
        }

        @Override
        public void onTopEmptyRegionClick() {
            Toast.makeText(CustomWearableList.this,
                    "Top empty area tapped", Toast.LENGTH_SHORT).show();
        }
    };

    // The following code ensures that the title scrolls as the user scrolls up
    // or down the list
    private WearableListView.OnScrollListener mOnScrollListener =
            new WearableListView.OnScrollListener() {
        @Override
        public void onAbsoluteScrollChange(int i) {
            // Only scroll the title up from its original base position
            // and not down.
            if (i > 0) {
                mHeader.setY(-i);
            }
        }

        @Override
        public void onScroll(int i) {
            // Placeholder
        }

        @Override
        public void onScrollStateChanged(int i) {
            // Placeholder
        }

        @Override
        public void onCentralPositionChanged(int i) {
            // Placeholder
        }
    };
```

```
}
```

8. In the AndroidManifest.xml, add the following entries in bold:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.learnandroidwear.simplelist">
    <uses-feature android:name="android.hardware.type.watch" />

    <application android:allowBackup="true" android:label="@string/app_name"
        android:icon="@drawable/ic_launcher"
        android:theme="@android:style/Theme.DeviceDefault">

        <activity
           <strong> android:name=".CustomWearableList"
            android:label="CustomWearableList" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter></strong>
        </activity>

    </application>

</manifest>
```

9. Press Shift+F9 to debug the project on a wearable device or emulator

Understanding the Code

Let's start by examining the activity_main.xml layout file. So far, all the views are relatively straightforward. The BoxInsetLayout element ensures that our list displays properly on both square and round devices. The WearableListView element allows you to create a list in your Android Wear app. To display a header on the top center, you added a TextView element within the RelativeLayout, and set the layout_alignParentTop and layout_centerHorizontal attributes to "true".

```
<android.support.wearable.view.BoxInsetLayout
    ...

    <RelativeLayout
        ...

        <android.support.wearable.view.WearableListView
            ...
        </android.support.wearable.view.WearableListView>

        <TextView
            android:id="@+id/header"
            android:layout_width="wrap_content"
            android:layout_height="60dp"
            android:layout_centerHorizontal="true"
            android:layout_alignParentTop="true"
            android:gravity="bottom"
            android:textSize="20sp"
            android:fontFamily="sans-serif-light"
            android:text="Settings"/>
```

```
    </RelativeLayout>
</android.support.wearable.view.BoxInsetLayout>
```

Next, you created another XML file named item_layout that specifies the layout definition of a single item in the list view. You added a CircledImageView to display a circle with an icon in the center. In this case, the CircledImageView is useful for displaying a visual cue by showing a large circle for the central item and a small circle for the other items. All of the other properties are self-explanatory.

```
<?xml version="1.0" encoding="utf-8"?>
<com.learnandroidwear.customwearablelistview.WearableListItemLayout
    ...

    <android.support.wearable.view.CircledImageView
        android:id="@+id/circle"
        app:circle_color="#c1c1c1"
        android:layout_gravity="center_vertical"
        android:layout_marginStart="16dp"
        android:layout_marginEnd="32dp"
        android:layout_width="52dip"
        android:layout_height="52dip"
        app:circle_border_color="#FFFFFFFF"
        app:circle_border_width="2dp" />

    <TextView
        android:id="@+id/name"
        android:alpha="0.5"
        android:gravity="center_vertical"
        android:fontFamily="sans-serif-condensed-light"
        android:textColor="@color/white"
        android:textSize="16sp"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"/>

</com.learnandroidwear.customwearablelistview.WearableListItemLayout>
```

NOTE *After you implement a custom layout for list items, you must provide a layout definition file that specifies the layout parameters of each of the components inside a list item. To ensure that the layout manager can find the correct layout to inflate, the root tag field must be in the following format: "<your_package_name.WearableListItemLayout>".*

Similar to the previous exercise, in your activity's onCreate() method, you obtained a reference to the WearableListView from your layout, assigned an adapter to it to bind the WearableListView and the items it is showing, and set a click listener to perform an action when the user selects a particular list item. Note one of the main difference is the addition of a scroll listener: It will be used to scroll the header up or down from its original base position. This will be explained in the next section.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
```

```
        setContentView(R.layout.activity_main);

        // Sample icons for the list
        ...

        // This is our list header
        mHeader = (TextView) findViewById(R.id.header);

        WearableListView wearableListView =
                (WearableListView) findViewById(R.id.wearable_List);
        wearableListView.setAdapter(new WearableAdapter(this, mIcons));
        wearableListView.setClickListener(mClickListener);
        wearableListView.addOnScrollListener(mOnScrollListener);
    }
```

Consider the following code snippet:

```
    // The following code ensures that the title scrolls as the user scrolls up
    // or down the list
    private WearableListView.OnScrollListener mOnScrollListener =
            new WearableListView.OnScrollListener() {

        @Override
        public void onAbsoluteScrollChange(int i) {
            // Only scroll the title up from its original base position
            // and not down.
            if (i > 0) {
                mHeader.setY(-i);
            }
        }

        @Override
        public void onScroll(int i) {
            // Placeholder
        }

        @Override
        public void onScrollStateChanged(int i) {
            // Placeholder
        }

        @Override
        public void onCentralPositionChanged(int i) {
            // Placeholder
        }
    };
```

In this case, you have a TextView above the WearableListView. When the user scrolls up or down the list, the header (i.e. the TextView with the id = "id/header") also needs to scroll up or down at the same time. An OnScrollListener can be set to receive messages when a scrolling event has occurred on the WearableListView. When the onAbsoluteScrollChange() method is called, you simply scroll the header out of sight when scrolling up, otherwise you scroll it back down to its base position .

The WearableAdapter class is responsible for inflating the correct layout for every row and for providing data for the WearableListView. This class is very similar to the adapter of the previous example.

Consider the following code snippet:

```
private static class ItemViewHolder extends WearableListView.ViewHolder {
    private CircledImageView mCircledImageView;
    private TextView mItemTextView;

    public ItemViewHolder(View itemView) {
        super(itemView);
        mCircledImageView = (CircledImageView)
                itemView.findViewById(R.id.circle);
        mItemTextView = (TextView) itemView.findViewById(R.id.name);
    }
}
```

Since we are dealing with a CircledImageView and a TextView, this class stores a reference of these two views into a ViewHolder to avoid unnecessary calls to findViewById().

When the WearableListView needs a new view, it calls this method and stores the views in a ViewHolder (that is, an instance of ItemViewHolder) that will be used to display the list items of the adapter using the onBindViewHolder() method.

```
@Override
public WearableListView.ViewHolder onCreateViewHolder(
        ViewGroup viewGroup, int i) {
    return new ItemViewHolder(mInflater.inflate(R.layout.list_item, null));
}
```

The onBindViewHolder() method is called by the WearableListView to display the data at the specified position in the list. In this case, we obtain a reference of aTextView and a CircledImageView from the ViewHolder, and respectively set the appropriate list icon and text.

```
@Override
public void onBindViewHolder(WearableListView.ViewHolder viewHolder,
                             int position) {
    ItemViewHolder itemViewHolder = (ItemViewHolder) viewHolder;
    CircledImageView circledView = itemViewHolder.mCircledImageView;
    circledView.setImageResource(mItems.get(position));
    TextView textView = itemViewHolder.mItemTextView;
    textView.setText(String.format("Item %d", position + 1));
}
```

To determine when items in a WearableListView becomes or cease to be the central item, you implemented the WearableListView.OnCenterProximityListener interface within the WeararbleListItemLayout class. Here, you implemented three methods:

```
@Override
protected void onFinishInflate() { ... }

@Override
public void onCenterPosition(boolean animate) { ... }

@Override
public void onNonCenterPosition(boolean animate) { ... }
```

The onFinishInflate() method is called after a layout and all of its children has been inflated from XML, so make sure you get a reference of your layout's child views in this method.

As their names suggests, the onCenterPosition() method is called when the row becomes the central item and the onNonCenterPosition() method is called when the row ceases to be the central item. In this case, you set the appropriate circle color and radius depending on the method being called. Moreover, if the animate parameter is set to true, you used the view animation system to perform some horizontal transitions on the view for added visual appeal.

More on the "app:layout_box" attribute

You may be asking yourself why you should care about the app:layout_box attribute when you can just set it to "all"? There are cases where omitting one or more edges can make your UI view more visually appealing. For example, to increase the display area for a list view while scrolling, it is recommended to set the app:layout_box attribute to "left|bottom|right". This will ensure that the list items will scroll past the top edge of the center square on a round screen as shown by the dotted lines in Figure 2-17:

Figure 2-17

To define the layout in Figure 2-17 with three edges (left, bottom and right), simply set the layout_box attributes to "left|bottom|right" as shown below:

```
<android.support.wearable.view.BoxInsetLayout>
    ...
    app:layout_box="left|bottom|right">
    ...
</android.support.wearable.view.BoxInsetLayout>
```

DESIGNING FOR ROUND AND SQUARE SCREENS

When designing layouts for Android Wear apps, you need to account for devices with a square screen (e.g., the Samsung Gear Live or the LG G) and a round screen (e.g., the Motorola moto

360 or the LG G Watch R). In this section, we'll take an in-depth look at how to handle multiple screen shapes.

There are two ways to design great wearable apps that look good on both form factors:

- Using WatchViewStub to detect the screen shape at runtime and inflate a square or round layout.
- Using BoxInsetLayout to define a single layout that works for both square and round screens

Figure 2-18. Left: Samsung Gear Live, Right: LG G Watch R

Using the Defaults

Let's suppose you start off by totally ignoring the issue of screen form factors. What happens?

If your application runs on a square screen, then everything will look fine and will run on the default device definition of 320 x 320 pixels. On the other hand, any content placed near the corners of the screen may be cropped on round devices, so layouts designed for square screens do not work well on round devices as shown in Figure 2-18:

Figure 2-19

WatchViewStub

If you take a look at the Hello World sample application generated during the project creation wizard, you will notice that it makes use of the WatchViewStub. To save you from writing all the boilerplate code to detect screen shapes, the Wearable Support Library provides a view called WatchViewStub that allows you to inflate either a square or round layout at runtime as shown in Figure 2-20. You typically use this approach when you want your app to look different depending on the shape of the device screen in terms of custom graphics or splash screens.

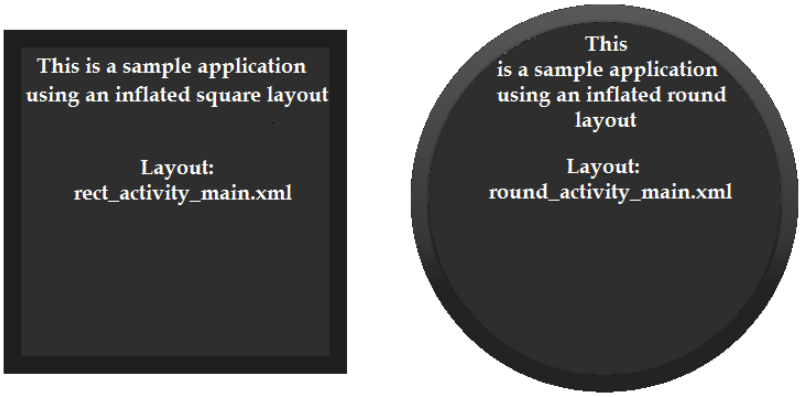


Figure 2-20

Consider the following XML layout. In order for the compiler to know which layout to inflate, we need to create a WatchViewStub layout and specify our square and round layouts as shown in bold in the activity_main.xml file:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.wearable.view.WatchViewStub
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/watch_view_stub"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:rectLayout="@layout/rect_activity_main"
    app:roundLayout="@layout/round_activity_main">
</android.support.wearable.view.WatchViewStub>
```

Next, you need to create and fill-out your UI views for both the square and round layouts. The XML layout names should match the values defined in the app:rectLayout and app:roundLayout of your activity_main.xml. Based on the above example, they should be named rect_activity_main.xml and round_activity_main.xml, respectively.

HINT *The most common mistake when implementing WatchViewStub is failing to fill out either the square or circular layout. In such a case, the developer might inadvertently add the UI views to only the round layout. Since the square layout was not defined, calls to* findViewById() *will throw a* NullPointerException. *It is therefore important to test your app on both form factors.*

In your main activity, you need to get a reference to the WatchViewStub layout as follows:

```
final WatchViewStub stub = (WatchViewStub) findViewById(R.id.watch_view_stub);
```

Finally, it is important to note that you cannot access your child views until inflation has been completed. Instead, you should implement the OnLayoutInflatedListener interface to detect when layout inflation is finished. When the appropriate views are inflated, WatchViewStub will invoke the onLayoutInflated method, thus allowing you to get the required references to your child views using findViewById.

```
stub.setOnLayoutInflatedListener(new WatchViewStub.OnLayoutInflatedListener() {
        @Override
        public void onLayoutInflated(WatchViewStub stub) {
            mTextView = (TextView) stub.findViewById(R.id.text);
        }
    });
```

NOTE *When the* inflate() *method is invoked inside the* WatchViewStub *class, the corresponding square (*rect_activity_main.xml*) or round (*round_activity_main.xml*) layout resource is inflated. The WatchViewStub* stub *variable then replaces itself in its parent with the inflated view, thereby making it visible in the view hierarchy of your application.*

BoxInsetLayout

The BoxInsetLayout is a placeholder on screen that can box its children in the center square of any round display as shown in the picture below. Essentially, it allows you to use the same layout on both square and round screens. You typically use this layout if you want to display standard UI views such as text, buttons, lists or cards.

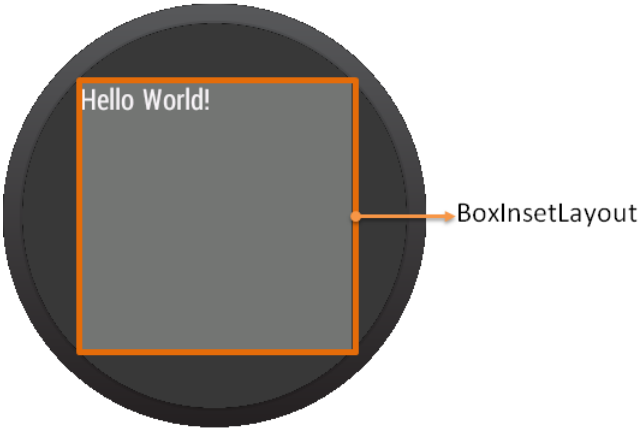


Figure 2-21

In the following example, we have a background image, a text view at the top, an OK button at the bottom left and a Cancel button at the bottom right. If the device has a square screen, the

BoxInsetLayout does nothing and everything is displayed correctly. However, on a round screen, we want all the UI views to be placed inside of the dashed center square as shown below:

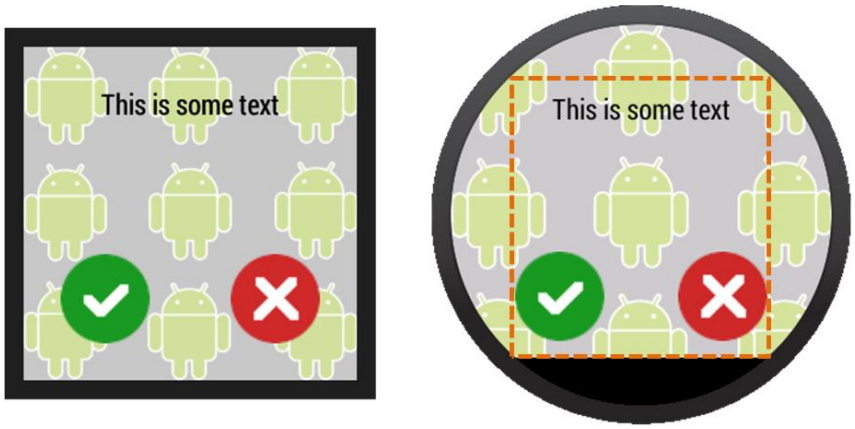


Figure 2-22

The following XML shows you how to create a layout that works on both square and round screens:

```
<android.support.wearable.view.BoxInsetLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:background="@drawable/robot_background"
    android:layout_height="match_parent"
    android:layout_width="match_parent">

    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        app:layout_box="all">
        <TextView
            android:gravity="center"
            android:layout_height="wrap_content"
            android:layout_width="match_parent"
            android:text="@string/sometext"
            android:textColor="@color/black" />

        <ImageButton
            android:layout_gravity="bottom|left"
            android:layout_height="50dp"
            android:layout_width="50dp"
            android:src="@drawable/ok" />

        <ImageButton
            android:layout_gravity="bottom|right"
            android:layout_height="50dp"
            android:layout_width="50dp"
            android:src="@drawable/cancel" />
    </FrameLayout>
</android.support.wearable.view.BoxInsetLayout>
```

Pay close attention to the following elements:

- android:background="@drawable/robot_background"
- app:layout_box="all"

As you can see, the BoxInsetLayout makes it easy to have our background image display over the entire circular area while containing the other UI views inside of the square area. The app:layout_box is an important attribute because it lets the FrameLayout element and its children know the set of edges that they need to be aligned with. This attribute can be set to left, top, right, bottom, or all. A combination of values can also be used, for example: "left|bottom|right".

Sample Chapter

Notification Framework

TOPICS COVERED IN THIS CHAPTER

- What is the Notification Framework?
- How to create a basic notification
- How to use the big view style, big picture style and stacked notifications
- How to add extra pages to a notification
- Creating an advanced custom notification
- Receiving voice input from a notification
- How to create an ongoing notification

Notifications allow users to keep informed about relevant and timely events such as incoming mail, the arrival of a new chat message, or a calendar event. Notifications make up a very important part of any Android App, but they are even more relevant on wearables. And since they are worn so close to your body, wearable notifications aren't just immediate, they're intimate and impactful. Your understanding of Android Wear is not rigorous until you master the architecture of Android's notification framework.

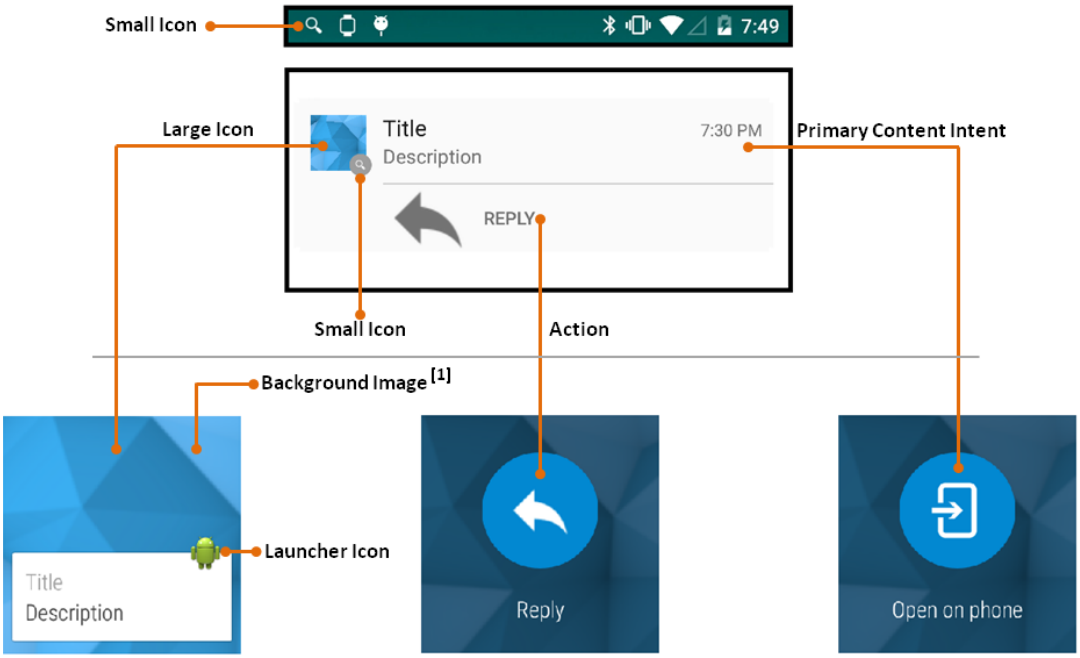
One of the most exciting features of Android Wear is how it seamlessly extends handheld notifications onto your wearable device right out of the box. Interacting with notifications on an Android Watch is as easy and intuitive as using them on your phone because Google has done an outstanding job optimizing them for wearables. And with a little bit of additional development, Android Wear gives you the flexibility to craft rich notification experiences that are more versatile and useful than ever. For example, you can add a reply action in your app to allow users to speak a reply or choose from a list of canned responses. You can implement additional pages with a notification to provide more information without requiring users to open your app on their phones. The five-day weather card is a good example of an additional

page. Furthermore, you can group all similar notifications together in a stack, allowing users to expand each card to view the details or to perform an action. Finally, you can provide a high-resolution background image to your notification to convey additional relevant information. For example, a suitable background image can be chosen to reflect the severity of the current weather or traffic conditions.

Our goal in this chapter is to unwrap the architecture of the notification system with a focus on customizing it for wearables. Throughout this chapter, we will draw a parallel between the notification framework of both handhelds and wearables. This will better your understanding of the different look and feel on wearables and will help you craft rich notifications that will look and work great on all devices.

ANATOMY OF A NOTIFICATION

To better understand the look and feel of Android Wear notifications, let's start by examining the basic parts of a notification and how they appear on different devices. At a minimum, all notifications require an icon, a title, and a description. Android supports optional actions that are displayed at the bottom of a handheld notification and as a separate action icon on wearables. Developers should attempt to perform actions on the wearable device whenever possible. With actions, a single tap should allow users to handle the most common tasks for a particular notification. In cases where the phone must be used, Android makes it possible to open the corresponding Android app by tapping on the handheld notification or the "Open on phone" icon on the wearable.



[1] Can be used when creating notifications with wearable extensions. This is the preferred method to display a nice background instead of a blurred icon.

Figure 3-01

IMPLEMENTATION GUIDELINES

The good news is that implementing notifications on both handhelds and wearables is very similar if one of the standard notification styles suits your needs. On the other hand, chances are you'll want to display a custom notification with wearable-specific options that better support your brand. In Android Wear, notifications can be exploited in four ways:

- If you currently have a handheld app that is using NotificationCompat to send notifications, your app's notifications will "just work" on Android Wear. That is, if one of the standard notification styles suits your needs, then there is no further development required. Android's notification framework will automatically share notifications between connected handhelds and wearables.

- If you want to share notifications between devices but the standard notification styles are not suitable, you can develop a separate wearable app to display custom notifications. For example, you can define a custom notification layout, which defines the notification's appearance in an XML layout file. Your handheld app will then need to send a data event (refer to the Data Layer API chapter) to your wearable app with the information needed to display your notification.

- If you want the actions available on the wearable to be different from those of the handheld, then use the WearableExtender.addAction() method. Once you add an action with this method, the wearable will ignore all handheld actions added via NotificationCompat.Builder.addAction(). That is, only the actions added with WearableExtender.addAction() will appear on the wearable.

- If you are developing a standalone Android Wear application and need to inform the user about important and timely events, then development of your own wearable notifications will be required.

BASIC NOTIFICATIONS

The figure below shows you the layout of a basic notification. At the bare minimum, it requires a notification icon, a title and a description.

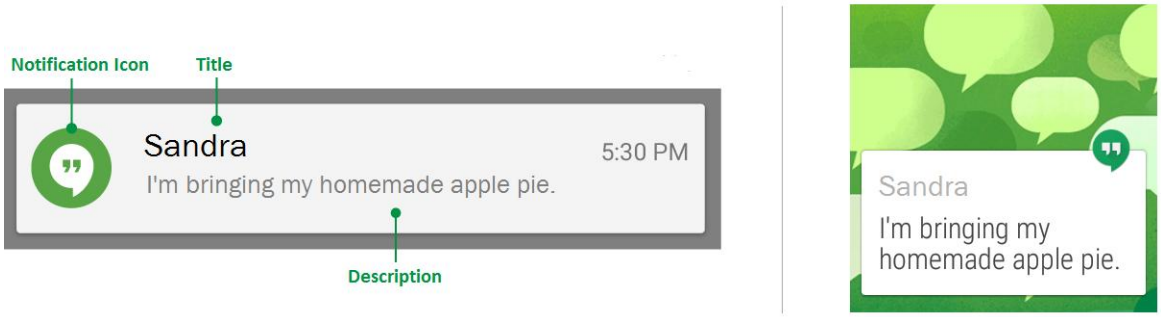


Figure 3-02

Creating a Basic Actionable Notification

Now that you have a better understanding of the anatomy of a notification and its use cases, the following Step-by-Step section demonstrates how to create a basic notification with a primary content intent that, when tapped, returns to the main activity and two pending intents (actions) that will invoke a separate activity.

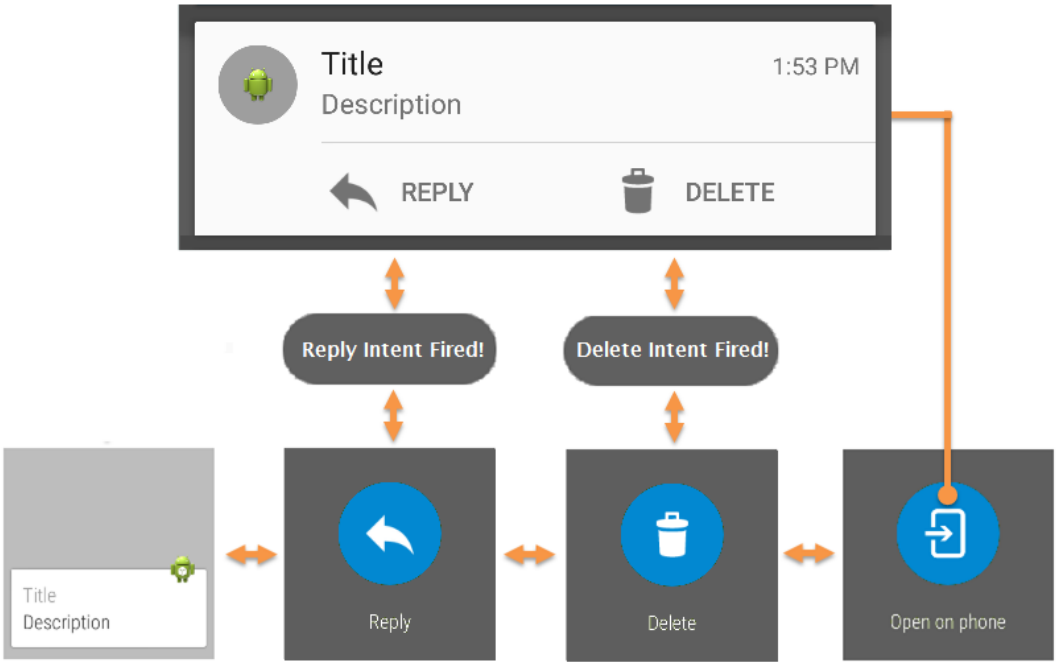


Figure 3-03

STEP BY STEP Creating a Basic Actionable Notification

1. Using Android Studio, create a new blank project for the Phone and Tablet platform and name it BasicNotification.
2. Select API 21 as the minimum SDK level, then click Next.
3. In the mobile activity dialog, select "Add no Activity", then click Finish.
4. In the build.gradle file for the main app, add the following statement in bold:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile 'com.google.android.support:wearable:1.1.0'
}
```

You will need to re-sync your project with the new gradle settings. Click on Tools → Android → Sync Project with Gradle Files as shown in Figure 3-04. Alternatively, if Android Studio detects that your gradle file has been modified, you can click the Sync Now link located on the top right-hand side.

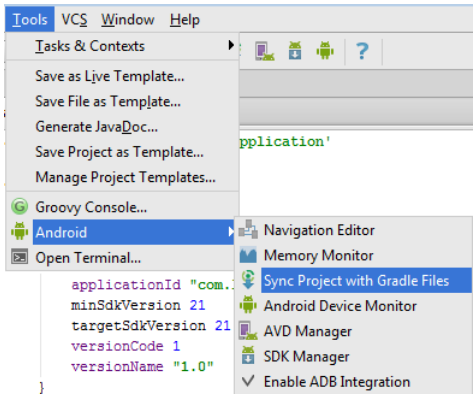


Figure 3-04

5. Unzip CoreIcons01.zip into a temporary folder. Go into the mdpi folder and select ic_action_delete.png and ic_action_reply.png followed by a copy (Ctrl-C).

6. In your Android Studio project pane, go to the res folder and select drawable as shown below. Right-click on drawable then choose paste.

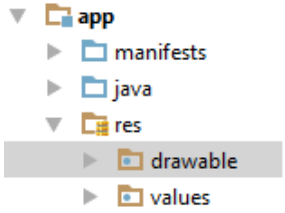


Figure 3-05

7. When the following dialog is displayed, choose "\res\drawable-mdpi" folder and click OK. Click OK again when the Copy dialog appears.

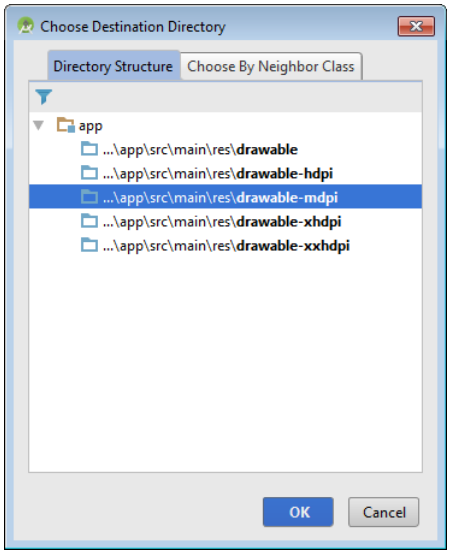


Figure 3-06

NOTE *For brevity, this example uses medium DPI images. For a real-life project, you need to ensure that each drawable folder has a set of images with different resolutions.*

8. If the layout folder in the project tree does not exist, right-click on the res folder, then click on New →Directory and name the folder "layout".

9. Add a new file in the res/layout folder and name it notification. Key in LinearLayout in the Root Tag field. Populate it as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:gravity="center"
    android:layout_height="fill_parent" >

    <TextView
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:gravity="center"
    android:text="Notification Action Detail" />
</LinearLayout>
```

10. Add a new Class file and name it NotificationAction. Populate it with the following content:

```
package com.learnandroidwear.basicnotification;

import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.support.v4.app.NotificationManagerCompat;
```

```java
import android.widget.Toast;

public class NotificationAction extends Activity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.notification);
    }

    @Override
    protected void onResume() {
        super.onResume();

        int notifID;
        String action;

        // Retrieve our Intent extra parameters
        notifID = getIntent().getIntExtra(NotificationActivity.ACTION_NOTIF_ID, 0);
        action = getIntent().getStringExtra(NotificationActivity.ACTION_MESSAGE);

        // Indicate tapped action via a Toast
        Toast.makeText(this, action, Toast.LENGTH_LONG).show();

        // Cancel current notification by specifying its ID
        NotificationManagerCompat.from(this).cancel(notifID);
    }

    @Override
    protected void onNewIntent(Intent intent) {
        super.onNewIntent(intent);
        setIntent(intent);
    }
}
```

11. Add a new XML file in the res/layout folder and name it activity_main. Key in LinearLayout in the Root Tag field. Populate it as follows:

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <Button
        android:id="@+id/btn_displaynotif"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Display Notification"
        android:onClick="onClick"/>

 </LinearLayout>
```

12. Add a new Class file and name it NotificationActivity. Populate it with the following content:

```java
package com.learnandroidwear.basicnotification;
```

```java
import android.app.Activity;
import android.app.PendingIntent;
import android.content.Intent;
import android.os.Bundle;
import android.support.v4.app.NotificationCompat;
import android.support.v4.app.NotificationManagerCompat;
import android.view.View;

public class NotificationActivity extends Activity {

    // A unique value that identifies the notification that we'll be sending.
    public static final int NOTIFICATION_ID = 1;

    public static final String ACTION_NOTIF_ID = "com.learnandroidwear.ACTION_NOTIF_ID";
    public static final String ACTION_MESSAGE = "com.learnandroidwear.ACTION_MESSAGE";

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }

    private void displayNotification() {

        // Cancel all previous notifications
        NotificationManagerCompat.from(this).cancelAll();

        // Use NotificationCompat.Builder to set up our notification.
        NotificationCompat.Builder builder = new NotificationCompat.Builder(this);

        // Set a small icon that will appear in the upper right corner of the
        // notification card
        builder.setSmallIcon(R.mipmap.ic_launcher);

        // Content title and description
        builder.setContentTitle("Title");
        builder.setContentText("Description");

        // Set a content intent to return to this example.
        builder.setContentIntent(PendingIntent.getActivity(this, 0,
                new Intent(this, NotificationActivity.class), 0));

        /* Create two intents that will be fired when the user taps on the
         *  corresponding notification action. The intent needs to be packaged
         *  into a pending intent so that the notification service can fire it on
         *  our behalf.
         *
         *  Note: Since we are using multiple distinct PendingIntent objects
         *  (i.e. NotificationActivity.class) in this example, we need to specify a
         *  unique RequestCode integer in getActivity(). In this case, we will
         *  use 0 and 1 respectively.
         */
        Intent intent1 = new Intent(this, NotificationAction.class);
        intent1.putExtra(ACTION_NOTIF_ID, NOTIFICATION_ID);
        intent1.putExtra(ACTION_MESSAGE, "Reply Intent Fired!");
        PendingIntent pendingIntent1 = PendingIntent.getActivity(this, 0, intent1, 0);

        Intent intent2 = new Intent(this, NotificationAction.class);
        intent2.putExtra(ACTION_NOTIF_ID, NOTIFICATION_ID);
```

```
        intent2.putExtra(ACTION_MESSAGE, "Delete Intent Fired!");
        PendingIntent pendingIntent2 = PendingIntent.getActivity(this, 1, intent2, 0);

        // In addition to the primary content intent (setContentIntent), add another
        // intent by passing a PendingIntent to the addAction() method.
        builder.addAction(R.drawable.ic_action_reply, "Reply", pendingIntent1);
        builder.addAction(R.drawable.ic_action_delete, "Delete", pendingIntent2);

        // Get an instance of the NotificationManagerCompat service
        NotificationManagerCompat notificationManager =
                NotificationManagerCompat.from(this);

        // Build the notification and notify it using notification manager.
        notificationManager.notify(NOTIFICATION_ID, builder.build());
    }

    public void onClick(View view) {
        displayNotification();
    }
}
```

13. In the AndroidManifest.xml, add the following entries in bold:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.learnandroidwear.basicnotification">
    <uses-feature android:name="android.hardware.type.watch" />
    <application android:allowBackup="true" android:label="@string/app_name"
        android:icon="@drawable/ic_launcher"
        android:theme="@android:style/Theme.DeviceDefault">

        <activity
            android:name=".NotificationActivity"
            android:label="@string/app_name" >

            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <activity
            android:name=".NotificationAction"
            android:exported="true"
            android:allowEmbedded="true"
            android:label="@string/app_name"
            android:launchMode="singleTop"
            android:taskAffinity="">
        </activity>

    </application>
</manifest>
```

14. Press Shift+F9 to debug the project on a handheld device. If your configuration is correct, the handheld notification will be extended to the wearable device.

15. On the handheld, drag the notification drawer down to reveal the notification. Tapping on the notification will launch the NotificationActivity. Tapping on the REPLY or DELETE action will launch the NotificationAction and display a Toast. On the wearable, swiping up from the bottom edge of your device will reveal the notification card. Keep swiping left to reveal the two intent actions. Notice how Android seamlessly extends handheld notification actions onto the wearable. As a result, tapping on an action will display the same Toast in the NotificationAction activity. Swipe left one more time to reveal the "Open on phone" primary content intent. Tapping on it will launch the NotificationActivity on your handheld.

Understanding the Code

While this application is not visually exciting, it does show you how to create a basic notification. And more importantly, it illustrates how you can create actionable notifications along with some important ways to pass and retrieve data between activities.

You first created an XML file named activity_main.xml to display a button to launch your notification:

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <Button
        android:id="@+id/btn_displaynotif"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:text="Display Notification"
        android:onClick="onClick"/>

 </LinearLayout>
```

In the preceding snippet, the android:onClick="onClick" attribute specifies the method that will be invoked when the button is pressed in the current activity.

To display a notification, you created an instance of Notification.Builder:

```
NotificationCompat.Builder builder = new NotificationCompat.Builder(this);
```

Next, you set the details of the notification as follows:

```
// Set a small icon that will appear in the upper right corner of the
 // notification card
 builder.setSmallIcon(R.mipmap.ic_launcher);

 // Content title and description
 builder.setContentTitle("Title");
 builder.setContentText("Description");
```

When the notification appears, you can swipe left three times to reveal the "Open on phone" icon. Tapping on it will launch the NotificationActivity activity. This is carried out by invoking a primary content intent specified in the setContentIntent() method that is used to set the PendingIntent that

will be invoked when the user taps on the notification. A PendingIntent is used to grant another application the right to perform an action on your application's behalf, often at a later time, regardless of whether your application is running or not. In this case, you passed a PendingIntent object to the setContentIntent() method as follows:

```
builder.setContentIntent(PendingIntent.getActivity(this, 0,
                        new Intent(this, NotificationActivity.class), 0));
```

In addition to the primary content intent defined by setContentIntent(), you can add additional actions by passing a PendingIntent to the addAction() method. You started by creating two intents that are used to launch another activity when the user taps on the corresponding action button, that is, Reply and Delete. You added a name/value pair to each intent object using putExtra() to pass the notification ID and a string value to the target activity, NotificationAction.class. The ID will be used to dismiss the notification later and the string value will be used to display the selected action via a Toast.

```
Intent intent1 = new Intent(this, NotificationAction.class);
intent1.putExtra(ACTION_NOTIF_ID, NOTIFICATION_ID);
intent1.putExtra(ACTION_MESSAGE, "Reply Intent Fired!");
...
Intent intent2 = new Intent(this, NotificationAction.class);
intent2.putExtra(ACTION_NOTIF_ID, NOTIFICATION_ID);
intent2.putExtra(ACTION_MESSAGE, "Delete Intent Fired!");
```

Next, you created two PendingIntent objects for each of your intents above. The getActivity() method will retrieve a PendingIntent object and you set it using the following arguments:

- context: The context in which the PendingIntent should start the activity
- request code: Private request code for the sender
- intent: The intent for the activity to be launched
- flags: The flags in which the activity is to be launched. You may also want to check out http://developer.android.com/reference/android/app/PendingIntent.html to learn more about the use cases for each flag.

As you can see, the NotificationAction.class component object is used for both intents. To enable two distinct component objects as active at the same time, you specified a unique request code for each getActivity() method:

```
PendingIntent pendingIntent1 = PendingIntent.getActivity(this, 0, intent1, 0);
...
PendingIntent pendingIntent2 = PendingIntent.getActivity(this, 1, intent2, 0);
```

NOTE *If you require multiple distinct PendingIntent objects to be active at the same time, then you will need to specify a unique* request code *ID. This allows the receiver to properly handle and retrieve the extra data from the correct activity instance of the Intent. Otherwise, if you only need one PendingIntent active at a time for any of the Intents you will use, then you can alternatively use the flags* FLAG_CANCEL_CURRENT *or* FLAG_UPDATE_CURRENT *to cancel or modify the PendingIntent behavior associated with the Intent you are supplying.*

In order to create two actions for your notification, you passed each PendingIntent object to separate addAction() methods. Actions are first displayed by the notification system by swiping left when the notification appears.

```
builder.addAction(R.drawable.ic_action_reply, "Reply", pendingIntent1);
builder.addAction(R.drawable.ic_action_delete, "Delete", pendingIntent2);
```

To display the notification, you created a NotificationManagerCompat object and used the notify() method:

```
// Get an instance of the NotificationManagerCompat service
NotificationManagerCompat notificationManager =
        NotificationManagerCompat.from(this);

// Build the notification and notify it using notification manager.
notificationManager.notify(NOTIFICATION_ID, builder.build());
```

When the user taps on the handheld notification or the "Open on phone" action on the wearable, the NotificationAction activity is launched. Here, you retrieve the Intent's extra parameters:

```
// Retrieve our Intent extra parameters
notifID = getIntent().getIntExtra(NotificationActivity.ACTION_NOTIF_ID, 0);
action = getIntent().getStringExtra(NotificationActivity.ACTION_MESSAGE);
```

Next, you display a Toast message indicating the selected action:

```
// Indicate tapped action via a Toast
Toast.makeText(this, action, Toast.LENGTH_LONG).show();
```

Finally, you dismiss the notification by using the cancel() method of the NotificationManagerCompat object and passing it the ID of the notification.

```
// Cancel current notification by specifying its ID
NotificationManagerCompat.from(this).cancel(notifID);
```

Consider the following attribute in bold in the AndroidManifest.xml file:

```
<activity
     android:name=".NotificationAction"
     android:exported="true"
     android:allowEmbedded="true"
     android:label="@string/app_name"
     android:launchMode="singleTop"
     android:taskAffinity="">
</activity>
```

As you are already aware, tapping on the Reply action will launch the NotificationAction activity. If you tap on the Delete action after that, it would start the same activity again using the same intent object (retrieved via getIntent()). As a result, an incorrect Toast message would be displayed ("Reply Intent Fired!" instead of "Delete Intent Fired!"). To address this issue, you need to add the launchMode="singleTop" attribute to your AndroidManifest.xml file. If an instance of the NotificationAction already exists, the system will route the new intent through a call to the onNewIntent() method rather than creating a new instance of the activity, as shown in the following code snippet:

```java
@Override
protected void onNewIntent(Intent intent) {
    super.onNewIntent(intent);
    setIntent(intent);
}
```

Sample Chapter

The Wearable Data Layer API

TOPICS COVERED IN THIS CHAPTER

- What are the Wearable Data Layer APIs?
- How to connect to Google Play Services
- How to send one-way messages using the Message API
- How to synchronize data using the Data Layer API

In this chapter, you will learn how to use the Wearable Data Layer API to create native wearable apps that can communicate securely with a paired smartphone. The Wearable Data Layer API is a part of the Google Play services and has an important role. For example, to conserve battery and reduce wait time, you can offload computationally intensive tasks to the handheld and then send the results back to the wearable. The navigation app for Android Wear is a good example of this, whereby the handheld takes care of GPS post-processing and the wearable receives location data. If you want to share notifications between devices but want to show customized notifications on the wearable to better fit your brand, your handheld app can send a one-way message to your wearable app with the information needed to display the notification. Finally, a fitness application is another good use case whereby the wearable can send periodic heart rate data to the handheld to display progress and show graphs.

In Android Wear, there are three basic APIs that are callable on both sides and facilitate the sending and syncing of data in addition to discovering nodes on an Android network.

Message API

The Message API provides a low-latency, one-way communication mechanism for sending byte arrays. There is no acknowledgement for sent data but you can easily implement code in such a way that the receiver can send a confirmation back through the same mechanism. This API is good for remote procedure calls (RPCs) such as controlling a handheld's media player from the wearable or starting an intent on the wearable from the handheld. If both devices are connected,

the system queues the message for delivery and returns a successful result code. Othewise, an error is returned. A successful result code is not a guarantee that the message was delivered successfully, because the devices may disconnect after receiving the result code.

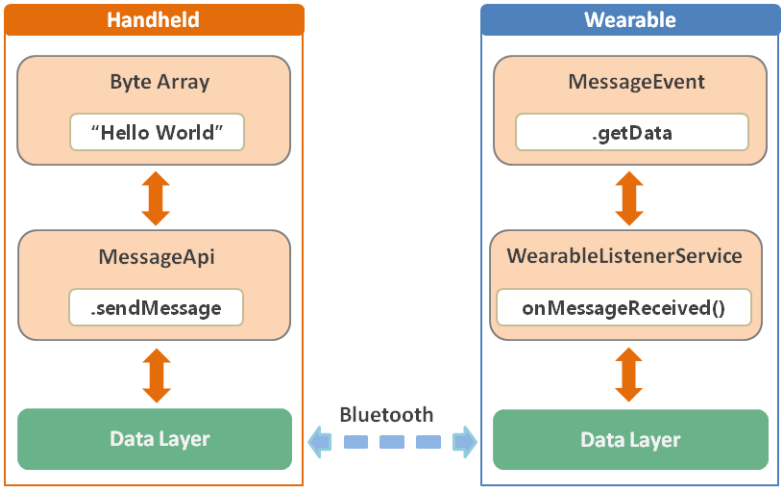


Figure 4-01

Data Layer API

This Data Layer API allows the syncing of data between a handheld and a wearable. Data pushed using this API is shared between devices and is available to both of them. You can think of the Data Layer API as a kind of shared data container with the ability to put data in, take data out, and be notified when this data changes on either side. You do not need to worry about devices disconnecting intermittently, because the system supports the caching of data and will synchronize it whenever the connection is re-established.

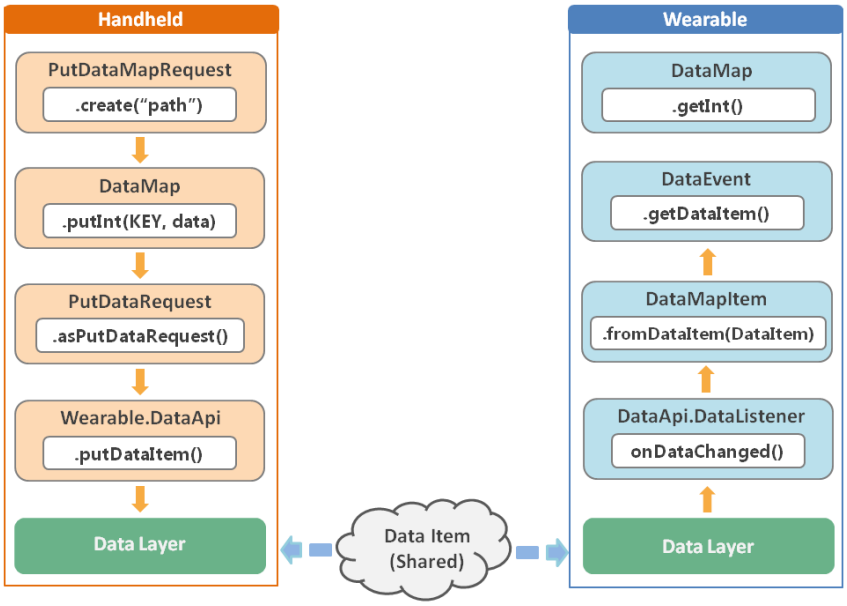


Figure 4-02

Node API

The Node API is used to handle events related to local and connected device nodes. It keeps track of device connections via the NodeListener interface method and is therefore useful for discovering nodes that compose your Android network. This API also provides a method for retrieving a list of all the connected nodes. For example, you can use this API to determine when a device becomes present and then toggle features on or off or have it display an icon indicating that the device is connected.

Wearable Data Objects

The Wearable Data Layer API consists of a set of data objects that the system can send and synchronize between devices and listeners in order to notify your apps of important events with the data layer.

The following table shows you the major data objects that will be covered in this chapter:

Object	Description
Data Item	A replicated data object with automatic syncing between the handheld and wearable. It is identified by its Uri, which contains an ID and a message path. The maximum size of a Data Item is 100 KB. To send larger amounts of data, use Assets.
Asset	A binary data blob for sending large data such as images. You attach assets to data items and the system automatically takes care of the transfer for you. The system will cache large assets to conserve Bluetooth bandwidth and will avoid retransmitting duplicate data. Although assets can be as large as required, care must be taken to minimize long wait times, which might affect the user experience. A common practice is to shrink an image in the handheld app before transferring it to the wearable as an asset.
DataLayerListenerService	Used as a background service, extending this class lets you listen for important data layer events, such as data changes, messages, and connectivity events.
DataListener	Implementing this class in an activity lets you receive important data layer events in a service such as changed or deleted data items. Use this class when you want to listen for changes while the user is actively using your app.

NOTE *The Wearable Data Layer APIs are designed for communication between handhelds and wearables. They provide all the methods and interfaces necessary to establish a connection and handle failures, discovering when a device comes into range and then sending and syncing data. As such, you should avoid using other methods, such as opening low-level sockets, to establish a communication channel.*

Accessing the Wearable API

The Wearable Data Layer API (and a host of other APIs such as Google+, Games, or Drive) is a part of the Google Play services library. It provides a common entry point to all the services, manages the network connection, handles failures, and can perform synchronous and asynchronous API calls to any of the other services.

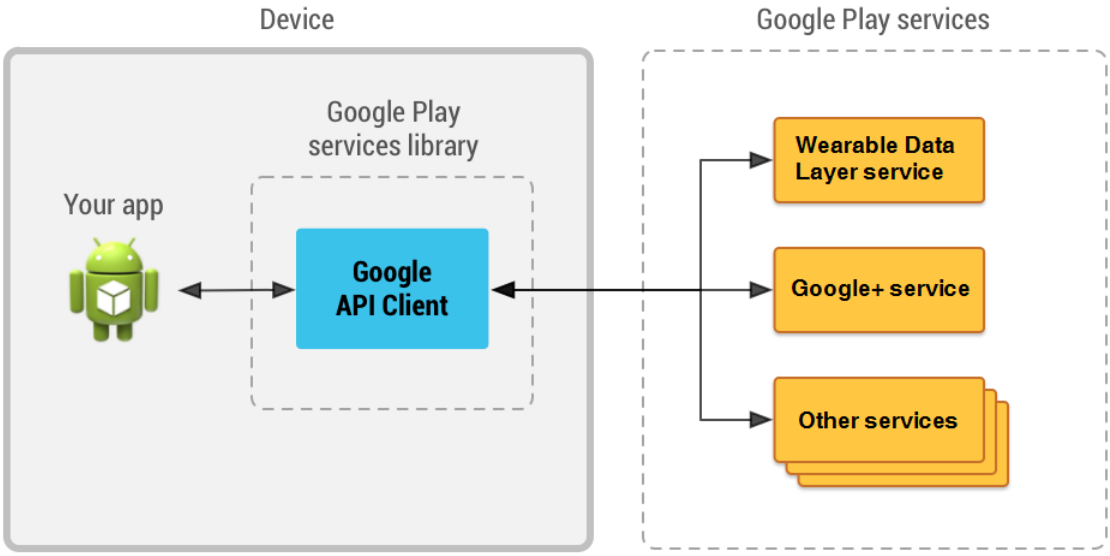


Figure 4-03

Since Google Play Services is outside the scope of this book, for further reading, you should definitely check out https://developer.android.com/google/play-services/index.html to get a general idea of how the Google APIs work. However, what is important to note is that on handheld devices that do not have the Android Wear app installed, connection requests that include the Wearable API will fail with the API_UNAVAILABLE error code. To work around this, if your app uses the Wearable API in addition to other Google APIs, use a separate GoogleApiClient instance to access only the Wearable API. This way, you can access other Google APIs on devices that are not paired with a wearable device.

When you use a separate GoogleApiClient instance to access only the Wearable API, the following code snippet lets you determine whether the Android Wear app is installed on the device:

```
// Connection failed listener method for a client that only
// requests access to the Wearable API
@Override
public void onConnectionFailed(ConnectionResult result) {
    if (result.getErrorCode() == ConnectionResult.API_UNAVAILABLE) {
        // The Android Wear app is not installed
    }
    ...
}
```

Sample Chapter

Projects

PROJECTS DEMONSTRATED IN THIS CHAPTER

- Creating a car catalog to see pictures, browse and voice annotate
- Adding actionable features to a custom notification
- Creating a remote controlled animated dice game

We talked about a lot of stuff in this book, and at this point, you should hopefully be comfortable with Android Wear development. This time, you will be using the techniques discussed in the previous chapters to create the following projects:

- A car catalog app to allow users to browse pictures, display specifications, voice annotate and schedule simulated test drives. The content browsing experience for this app will be simple and intuitive, as well as visually pleasing. To do so, you will be using the GridViewPager, extending the FragmentGridPagerAdapter class and each page will be assigned a CardFragment.
- Expand the custom notification fitness exercise from chapter 3 with actionable features such as the ability to start, pause, resume and reset the countdown timer.
- Explore the exciting world of touch and gestural interactions. Build an animated dice game using the handheld device as a remote control or the wearable's accelerometer sensor to detect shake gestures in order to roll a dice.

1) BUILDING A CAR CATALOG APP

In the following Step by Step, you will learn how to create a car catalog app to browse, take notes and schedule test drives.

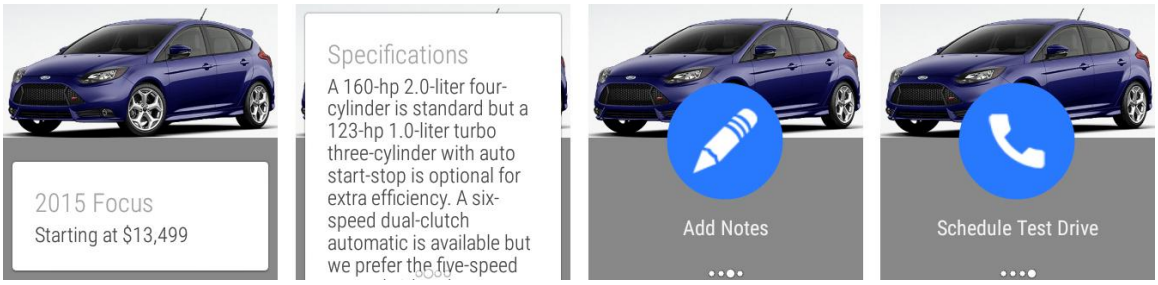


Figure 5-1

2) THE FITNESS APP: ADDING ACTIONABLE FEATURES

In the following Step by Step, you will expand the fitness exercise from chapter 3 and implement actionable features as shown in figure 5-3. The finished application is a fully functional workout exercise with the ability to start, pause, resume and reset the countdown timer. After that, feel free to take it to the next level.

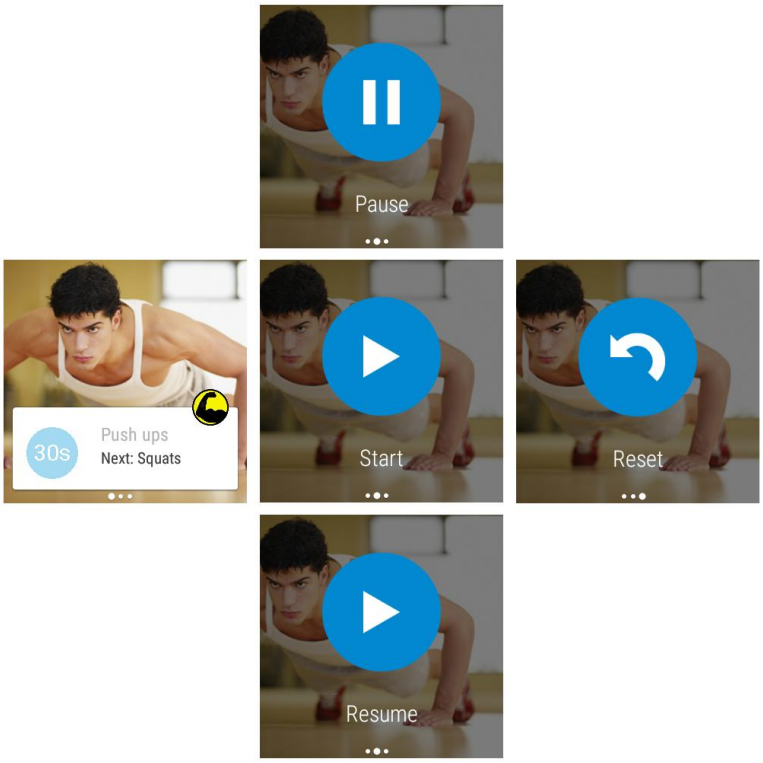


Figure 5-2

3) REMOTE CONTROLLED ANIMATED DICE GAME

In the following Step by Step, you will learn how to build an animated dice game using the mobile device as a remote control or the wearable's accelerometer sensor to detect shake gestures in order to roll a dice. You will also learn how to animate shapes, establish a data connection between the mobile and Wear devices and display a history of dice roll outcomes on the handheld device.

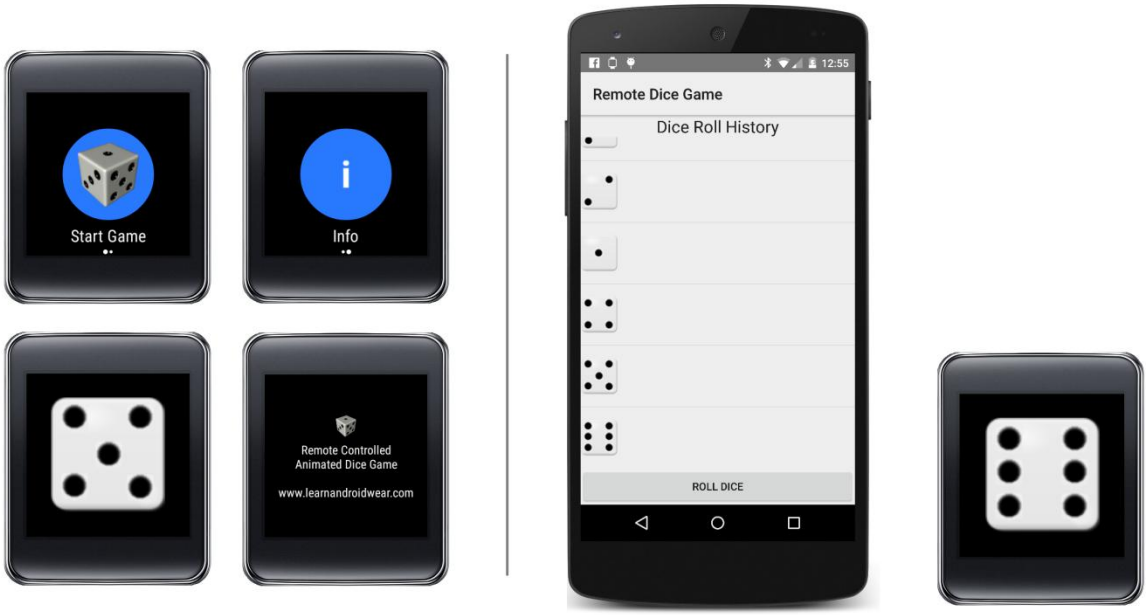


Figure 5-3

ENJOYED THE SAMPLE?

Questions? You can email me at aho@learnandroidwear.com.

Learn more about the book at www.learnandroidwear.com.