69. Printing with the Android Printing Framework

With the introduction of the Android 4.4 KitKat release, it became possible to print content from within Android applications. While subsequent chapters will explore in more detail the options for adding printing support to your own applications, this chapter will focus on the various printing options now available in Android and the steps involved in enabling those options. Having covered these initial topics, the chapter will then provide an overview of the various printing features that are available to Android developers in terms of building printing support into applications.

69.1 The Android Printing Architecture

Printing in Android 4.4 and later is provided by the Printing framework. In basic terms, this framework consists of a Print Manager and a number of print service plugins. It is the responsibility of the Print Manager to handle the print requests from applications on the device and to interact with the print service plugins that are installed on the device, thereby ensuring that print requests are fulfilled. By default, many Android devices have print service plugins installed to enable printing using the Google Cloud Print and Google Drive services. Print Services Plugin for other printer types, if not already installed, may also be obtained from the Google Play store. Print Service Plugins are currently available for HP, Epson, Samsung and Canon printers and plugins from other printer manufactures will most likely be released in the near future though the Google Cloud Print service plugin can also be used to print from an Android device to just about any printer type and model. For the purposes of this book, we will use the HP Print Services Plugin as a reference example.

69.2 The Print Service Plugins

The purpose of the Print Service plugins is to enable applications to print to compatible printers that are visible to the Android device via a local area wireless network or Bluetooth. Print Service plugins are currently available for a wide range of printer brands including HP, Samsung, Brother, Canon, Lexmark and Xerox.

The presence of the Print Service Plugin on an Android device can be verified by loading the Google Play app and performing a search for “Print Services Plugin”. Once the plugin is listed in the Play Store, and in the event that the plugin is not already installed, it can be installed by selecting the *Install* button. Figure 69-1, for example, shows the HP Print Service plugin within Google Play:

Figure 69-1

The Print Services plugins will automatically detect compatible HP printers on the network to which the Android device is currently connected and list them as options when printing from an application.

69.3 Google Cloud Print

Google Cloud Print is a service provided by Google that enables you to print content onto your own printer over the web from anywhere with internet connectivity. Google Cloud Print supports a wide range of devices and printer models in the form of both *Cloud Ready* and *Classic* printers. A Cloud Ready printer has technology built-in that enables printing via the web. Manufacturers that provide cloud ready printers include Brother, Canon, Dell, Epson, HP, Kodak and Samsung. To identify if your printer is both cloud ready and supported by Google Cloud Print, review the list of printers at the following URL:

https://www.google.com/cloudprint/learn/printers.html

In the case of classic, non-Cloud Ready printers, Google Cloud Print provides support for cloud printing through the installation of software on the computer system to which the classic printer is connected (either directly or over a home or office network).

To set up Google Cloud Print, visit the following web page and login using the same Google account ID that you use when logging in to your Android devices:

https://www.google.com/cloudprint/learn/index.html

Once printers have been added to your Google Cloud Print account, they will be listed as printer destination options when you print from within Android applications on your devices.

69.4 Printing to Google Drive

In addition to supporting physical printers, it is also possible to save printed output to your Google Drive account. When printing from a device, select the *Save to Google Drive* option in the printing panel. The content to be printed will then be converted to a PDF file and saved to the Google Drive cloud-based storage associated with the currently active Google Account ID on the device.

69.5 Save as PDF

The final printing option provided by Android allows the printed content to be saved locally as a PDF file on the Android device. Once selected, this option will request a name for the PDF file and a location on the device into which the document is to be saved.

Both the Save as PDF and Google Drive options can be invaluable in terms of saving paper when testing the printing functionality of your own Android applications.

69.6 Printing from Android Devices

Google recommends that applications which provide the ability to print content do so by placing the print option in the Overflow menu (a topic covered in some detail in the chapter entitled *Creating and Managing Overflow Menus on Android*). A number of applications bundled with Android now include “Print…” menu options. Figure 69-2, for example, shows the Print option in the Overflow menu of the Chrome browser application:

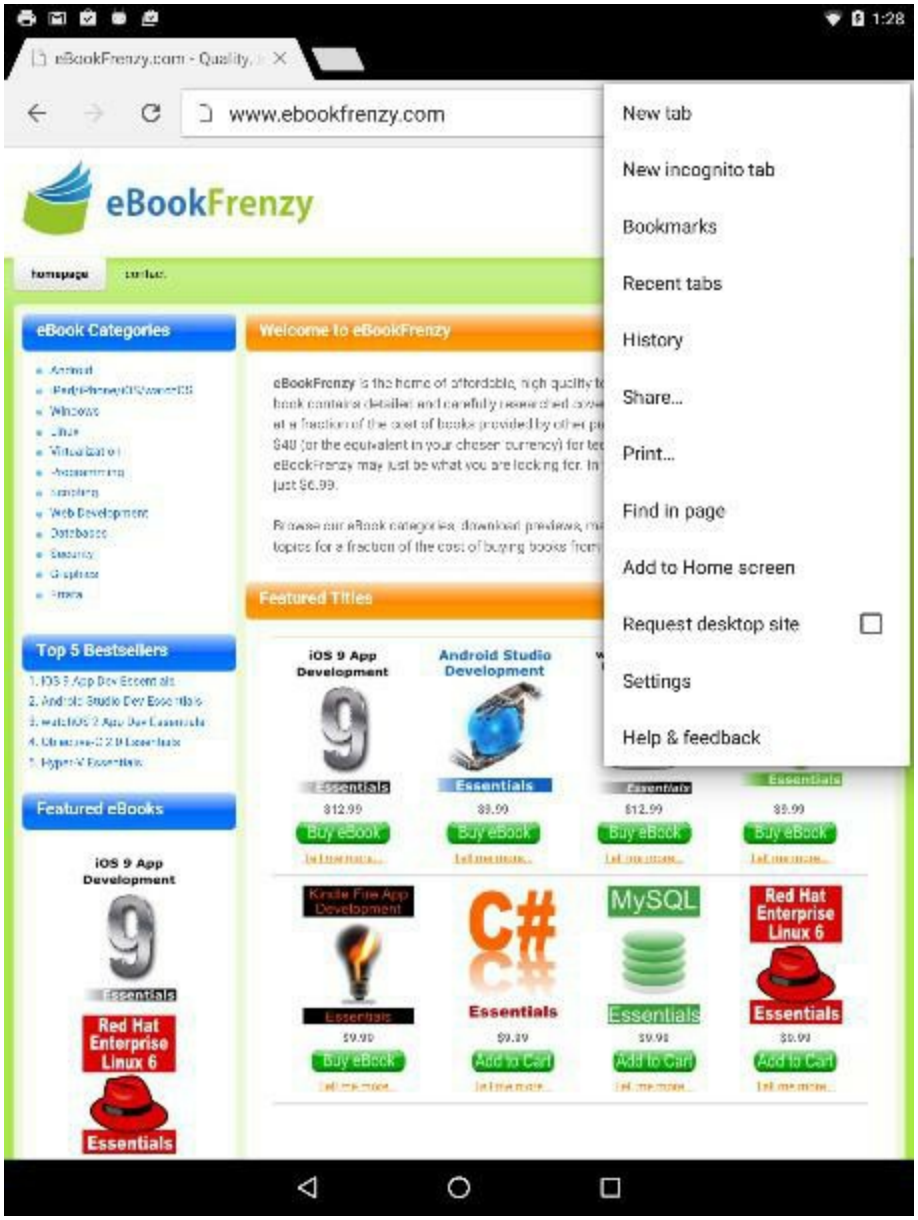

Figure 69-2

Once the print option has been selected from within an application, the standard Android print screen will appear showing a preview of the content to be printed as illustrated in Figure 69-3:

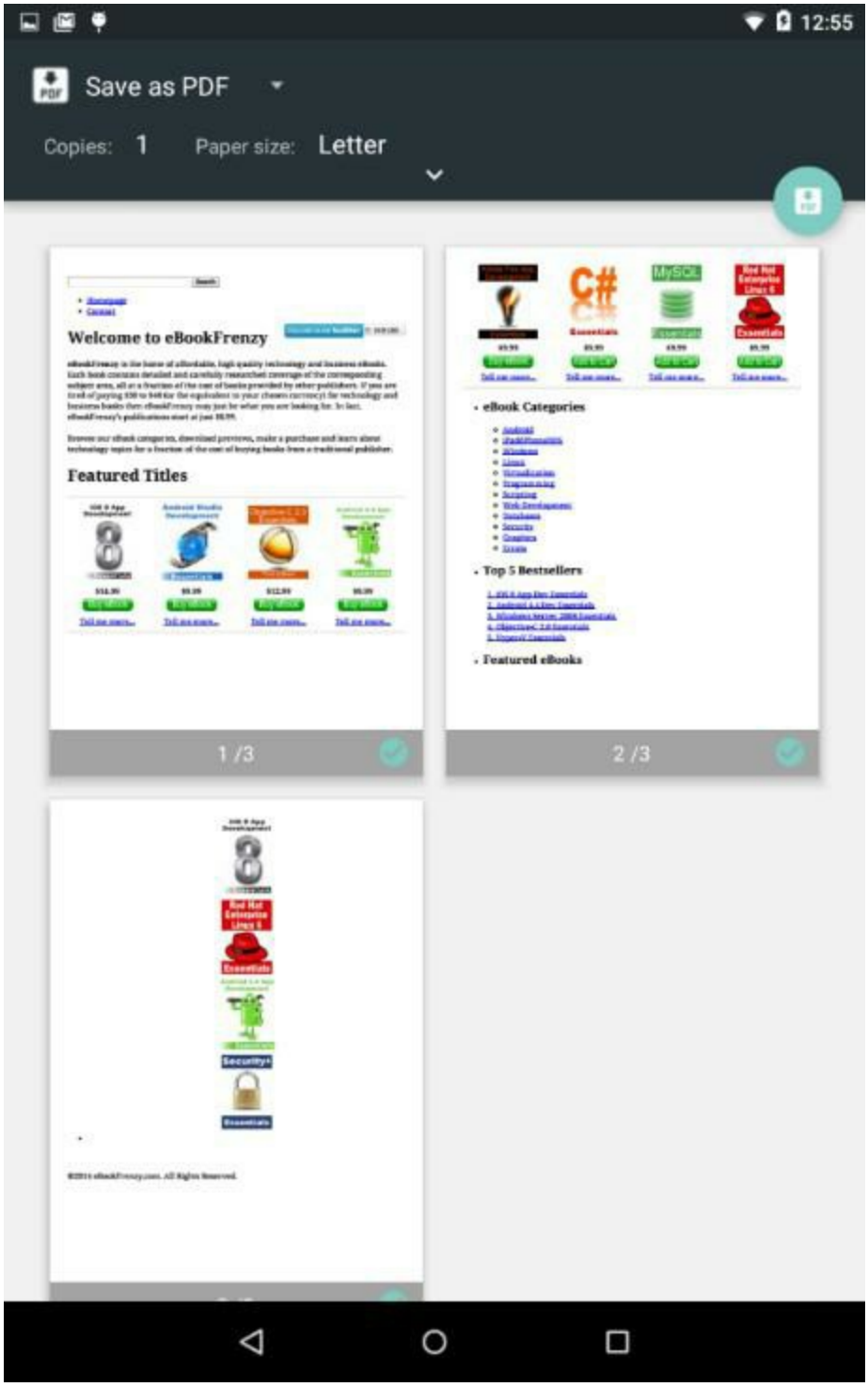

Figure 69-3

Tapping the panel along the top of the screen will display the full range of printing options:

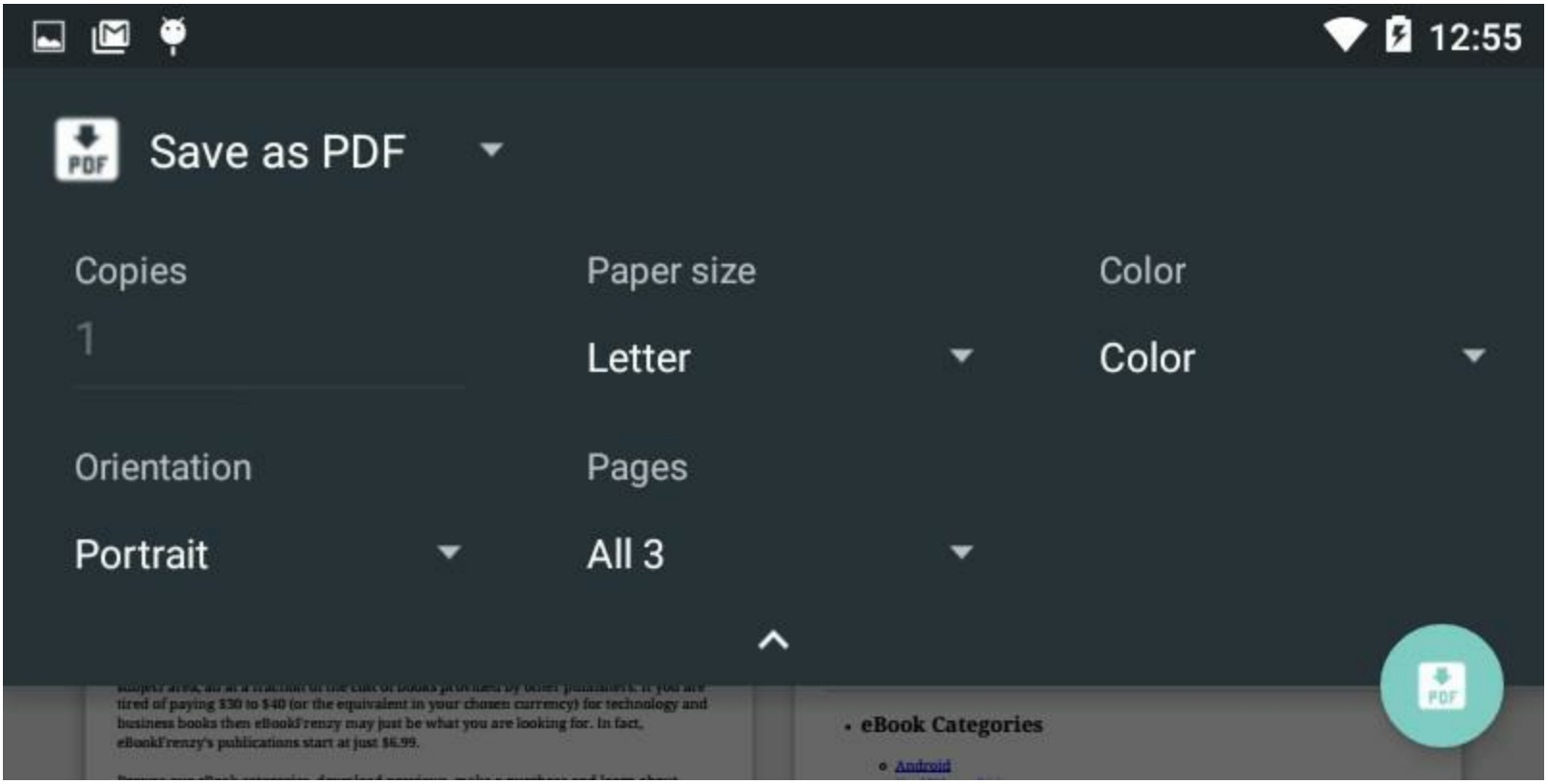

Figure 69-4

The Android print panel provides the usual printing options such as paper size, color, orientation and number of copies. Other print destination options may be accessed by tapping on the current printer or PDF output selection.

69.7 Options for Building Print Support into Android Apps

The Printing framework introduced into the Android 4.4 SDK provides a number of options for incorporating print support into Android applications. These options can be categorized as follows:

69.7.1 Image Printing

As the name suggests, this option allows image printing to be incorporated into Android applications. When adding this feature to an application, the first step is to create a new instance of the PrintHelper class:

```
PrintHelper imagePrinter = new PrintHelper(context);
```

Next, the scale mode for the printed image may be specified via a call to the *setScaleMode()* method of the PrintHelper instance. Options are as follows:

- **SCALE_MODE_FIT** – The image will be scaled to fit within the paper size without any cropping or changes to aspect ratio. This will typically result in white space appearing in one dimension.
- **SCALE_MODE_FILL** – The image will be scaled to fill the paper size with cropping performed where necessary to avoid the appearance of white space in the printed output.

In the absence of a scale mode setting, the system will default to SCALE_MODE_FILL. The following code, for example, sets scale to fit mode on the previously declared PrintHelper instance:

```
imagePrinter.setScaleMode(PrintHelper.SCALE_MODE_FIT);
```

Similarly, the color mode may also be configured to indicate whether the print output is to be in color

or black and white. This is achieved by passing one of the following options through to the *setColorMode()* method of the PrintHelper instance:

- **COLOR_MODE_COLOR** – Indicates that the image is to be printed in color.
- **COLOR_MODE_MONOCHROME** – Indicates that the image is to be printed in black and white.

The printing framework will default to color printing unless the monochrome option is specified as follows:

```
imagePrinter.setColorMode(PrintHelper.COLOR_MODE_MONOCHROME);
```

All that is required to complete the printing operation is an image to be printed and a call to the *printBitmap()* method of the PrintHelper instance, passing through a string representing the name to be assigned to the print job and a reference to the image (in the form of either a Bitmap object or a Uri reference to the image):

```
Bitmap bitmap = BitmapFactory.decodeResource(getResources(),
            R.drawable.oceanscene);
imagePrinter.printBitmap("My Test Print Job", bitmap);
```

Once the print job has been started, the Printing framework will display the print dialog and handle both the subsequent interaction with the user and the printing of the image on the user-selected print destination.

69.7.2 Creating and Printing HTML Content

The Android Printing framework also provides an easy way to print HTML based content from within an application. This content can either be in the form of HTML content referenced by the URL of a page hosted on a web site, or HTML content that is dynamically created within the application.

To enable HTML printing, the WebView class has been extended in Android 4.4 to include support for printing with minimal coding requirements.

When dynamically creating HTML content (as opposed to loading and printing an existing web page) the process involves the creation of a WebView object and associating with it a WebViewClient instance. The web view client is then configured to start a print job when the HTML has finished being loaded into the WebView. With the web view client configured, the HTML is then loaded into the WebView, at which point the print process is triggered.

Consider, for example, the following code:

```
private WebView myWebView;

public void printContent(View view)
{
       WebView webView = new WebView(this);
       webView.setWebViewClient(new WebViewClient() {

            public boolean shouldOverrideUrlLoading(WebView view,
                      String url)
            {
                  return false;
            }
```

```
            @Override
                public void onPageFinished(WebView view, String url) {
                       createWebPrintJob(view);
                       myWebView = null;
                   }
               });

               String htmlDocument =
                    "<html><body><h1>Android Print Test</h1><p>"
                   + "This is some sample content.</p></body></html>";



               webView.loadDataWithBaseURL(null, htmlDocument,
                         "text/HTML", "UTF-8", null);

               myWebView = webView;
    }
```

The code in this method begins by declaring a variable named *myWebView* in which will be stored a reference to the WebView instance created in the method. Within the *printContent()* method, an instance of the WebView class is created to which a WebViewClient instance is then assigned.

The WebViewClient assigned to the web view object is configured to indicate that loading of the HTML content is to be handled by the WebView instance (by returning *false* from the *shouldOverrideUrlLoading()*) method. More importantly, an *onPageFinished()* handler method is declared and implemented to call a method named *createWebPrintJob()*. The *onPageFinished()* callback method will be called automatically when all of the HTML content has been loaded into the web view. This ensures that the print job is not started until the content is ready, thereby ensuring that all of the content is printed.

Next, a string is created containing some HTML to serve as the content. This is then loaded into the web view. Once the HTML is loaded, the *onPageFinished()* method will trigger. Finally, the method stores a reference to the web view object. Without this vital step, there is a significant risk that the Java runtime system will assume that the application no longer needs the web view object and will discard it to free up memory (a concept referred to in Java terminology as *garbage collection*) resulting in the print job terminating prior to completion.

All that remains in this example is to implement the *createWebPrintJob()* method as follows:

```
    private void createWebPrintJob(WebView webView) {

        PrintManager printManager = (PrintManager) this
                    .getSystemService(Context.PRINT_SERVICE);

        PrintDocumentAdapter printAdapter =
                     webView.createPrintDocumentAdapter("MyDocument");
        String jobName = getString(R.string.app_name) + " Document";

        PrintJob printJob = printManager.print(jobName, printAdapter,
                    new PrintAttributes.Builder().build());
    }
```

This method simply obtains a reference to the PrintManager service and instructs the web view instance to create a print adapter. A new string is created to store the name of the print job (which in

this is case based on the name of the application and the word "Document").

Finally, the print job is started by calling the *print()* method of the print manager, passing through the job name, print adapter and a set of default print attributes. If required, the print attributes could be customized to specify resolution (dots per inch), margin and color options.

69.7.3 Printing a Web Page

The steps involved in printing a web page are similar to those outlined above, with the exception that the web view is passed the URL of the web page to be printed in place of the dynamically created HTML, for example:

```
webView.loadUrl("http://developer.android.com/google/index.html");
```

It is also important to note that the WebViewClient configuration is only necessary if a web page is to automatically print as soon as it has loaded. If the printing is to be initiated by the user selecting a menu option after the page has loaded, only the code in the *createWebPrintJob()* method outlined above need be included in the application code. The next chapter, entitled *An Android HTML and Web Content Printing Example*, will demonstrate just such a scenario.

69.7.4 Printing a Custom Document

While the HTML and web printing features introduced by the Printing framework provide an easy path to printing content from within an Android application, it is clear that these options will be overly simplistic for more advanced printing requirements. For more complex printing tasks, the Printing framework also provides custom document printing support. This allows content in the form of text and graphics to be drawn onto a canvas and then printed.

Unlike HTML and image printing, which can be implemented with relative ease, custom document printing is a more complex, multi-stage process which will be outlined in the *A Guide to Android Custom Document Printing* chapter of this book. These steps can be summarized as follows:

- Connect to the Android Print Manager
- Create a Custom Print Adapter sub-classed from the PrintDocumentAdapter class
- Create a PdfDocument instance to represent the document pages
- Obtain a reference to the pages of the PdfDocument instance, each of which has associated with it a Canvas instance
- Draw the content on the page canvases
- Notify the print framework that the document is ready to print

The custom print adapter outlined in the above steps needs to implement a number of methods which will be called upon by the Android system to perform specific tasks during the printing process. The most important of these are the *onLayout()* method which is responsible for re-arranging the document layout in response to the user changing settings such as paper size or page orientation, and the *onWrite()* method which is responsible for rendering the pages to be printed. This topic will be covered in detail in the chapter entitled *A Guide to Android Custom Document Printing*.

69.8 Summary

The Android 4.4 KitKat release introduced the ability to print content from Android devices. Print output can be directed to suitably configured printers, a local PDF file or to the cloud via Google Drive. From the perspective of the Android application developer, these capabilities are available

for use in applications by making use of the Printing framework. By far the easiest printing options to implement are those involving content in the form of images and HTML. More advanced printing may, however, be implemented using the custom document printing features of the framework.

70. An Android HTML and Web Content Printing Example

As outlined in the previous chapter, entitled *Printing with the Android Printing Framework*, the Android Printing framework can be used to print both web pages and dynamically created HTML content. While there is much similarity in these two approaches to printing, there are also some subtle differences that need to be taken into consideration. This chapter will work through the creation of two example applications in order to bring some clarity to these two printing options.

70.1 Creating the HTML Printing Example Application

Begin this example by launching the Android Studio environment and creating a new project, entering *HTMLPrint* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 21: Android 5.0 (Lollipop). Continue to proceed through the screens, requesting the creation of an Empty Activity named *HTMLPrintActivity* with a corresponding layout named *activity_html_print*.

70.2 Printing Dynamic HTML Content

The first stage of this tutorial is to add code to the project to create some HTML content and send it to the Printing framework in the form of a print job.

Begin by locating the *HTMLPrintActivity.java* file (located in the Project tool window under *app -> java -> com.ebookfrenzy.htmlprint*) and loading it into the editing panel. Once loaded, modify the code so that it reads as outlined in the following listing:

```
package com.ebookfrenzy.htmlprint;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import android.print.PrintAttributes;
import android.print.PrintDocumentAdapter;
import android.print.PrintManager;
import android.content.Context;

public class HTMLPrintActivity extends AppCompatActivity {

    private WebView myWebView;

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

        WebView webView = new WebView(this);
        webView.setWebViewClient(new WebViewClient() {
```

```
            public boolean shouldOverrideUrlLoading(WebView view,
                                        WebResourceRequest request)
            {
                return false;
            }

            @Override
            public void onPageFinished(WebView view, String url)
            {
                createWebPrintJob(view);
                myWebView = null;
            }
        });

        String htmlDocument =
           "<html><body><h1>Android Print Test</h1><p>"
           + "This is some sample content.</p></body></html>";

        webView.loadDataWithBaseURL(null, htmlDocument,
                "text/HTML", "UTF-8", null);

        myWebView = webView;
    }
}
```

The code changes begin by declaring a variable named *myWebView* in which will be stored a reference to the WebView instance used for the printing operation. Within the *onCreate()* method, an instance of the WebView class is created to which a WebViewClient instance is then assigned.

The WebViewClient assigned to the web view object is configured to indicate that loading of the HTML content is to be handled by the WebView instance (by returning *false* from the *shouldOverrideUrlLoading()* method). More importantly, an *onPageFinished()* handler method is declared and implemented to call a method named *createWebPrintJob()*. The *onPageFinished()* method will be called automatically when all of the HTML content has been loaded into the web view. As outlined in the previous chapter, this step is necessary when printing dynamically created HTML content to ensure that the print job is not started until the content has fully loaded into the WebView.

Next, a String object is created containing some HTML to serve as the content and subsequently loaded into the web view. Once the HTML is loaded, the *onPageFinished()* callback method will trigger. Finally, the method stores a reference to the web view object in the previously declared *myWebView* variable. Without this vital step, there is a significant risk that the Java runtime system will assume that the application no longer needs the web view object and will discard it to free up memory resulting in the print job terminating before completion.

All that remains in this example is to implement the *createWebPrintJob()* method which is currently configured to be called by the *onPageFinished()* callback method. Remaining within the *HTMLPrintActivity.java* file, therefore, implement this method so that it reads as follows:

```
private void createWebPrintJob(WebView webView) {

    PrintManager printManager = (PrintManager) this
            .getSystemService(Context.PRINT_SERVICE);
```

```
        PrintDocumentAdapter printAdapter =
                webView.createPrintDocumentAdapter("MyDocument");

        String jobName = getString(R.string.app_name) + " Print Test";

        printManager.print(jobName, printAdapter,
                  new PrintAttributes.Builder().build());
    }
```

This method obtains a reference to the PrintManager service and instructs the web view instance to create a print adapter. A new string is created to store the name of the print job (in this case based on the name of the application and the word “Print Test”).

Finally, the print job is started by calling the *print()* method of the print manager, passing through the job name, print adapter and a set of default print attributes.

Compile and run the application on a device or emulator running Android 5.0 or later. Once launched, the standard Android printing page should appear as illustrated in Figure 70-1.

Figure 70-1

Print to a physical printer if you have one configured, save to Google Drive or, alternatively, select the option to save to a PDF file. Once the print job has been initiated, check the generated output on your chosen destination. Note that when using the Save to PDF option, the system will request a name and location for the PDF file. The *Downloads* folder makes a good option, the contents of which can be viewed by selecting the *Downloads* icon located amongst the other app icons on the device. Figure 70-2, for example, shows the PDF output generated by the Save to PDF option viewed on an Android device:

Figure 70-2

70.3 Creating the Web Page Printing Example

The second example application to be created in this chapter will provide the user with an Overflow menu option to print the web page currently displayed within a WebView instance. Create a new project in Android Studio, entering *WebPrint* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 21: Android 5.0 (Lollipop). Continue to proceed through the screens, requesting the creation of a Basic Activity (since we will be making use of the context menu provided by the Basic Activity template) named *WebPrintActivity* with the remaining properties set to the default values.

70.4 Removing the Floating Action Button

Selecting the Basic Activity template provided a context menu and a floating action button. Since the floating action button is not required by the app it can be removed before proceeding. Load the *activity_web_print.xml* layout file into the Layout Editor, select the floating action button and tap the keyboard *Delete* key to remove the object from the layout. Edit the *WebPrintActivity.java* file and remove the floating action button code from the onCreate method as follows:

```
@Override
```

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_web_print);
    Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
    setSupportActionBar(toolbar);

    FloatingActionButton fab =
            (FloatingActionButton) findViewById(R.id.fab);
    fab.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            Snackbar.make(view, "Replace with your own action",
                    Snackbar.LENGTH_LONG)
                    .setAction("Action", null).show();
        }
    });
}
```

70.5 **Designing the User Interface Layout**

Load the *content_web_print.xml* layout resource file into the Layout Editor tool if it has not already been loaded and, in Design mode, select and delete the “Hello World!” TextView object. From the *Widgets* section of the palette, drag and drop a WebView object onto the center of the device screen layout. Using the Properties tool window, change the layout_width and layout_height properties of the WebView to *match_parent* so that it fills the entire layout canvas as outlined in Figure 70-3:

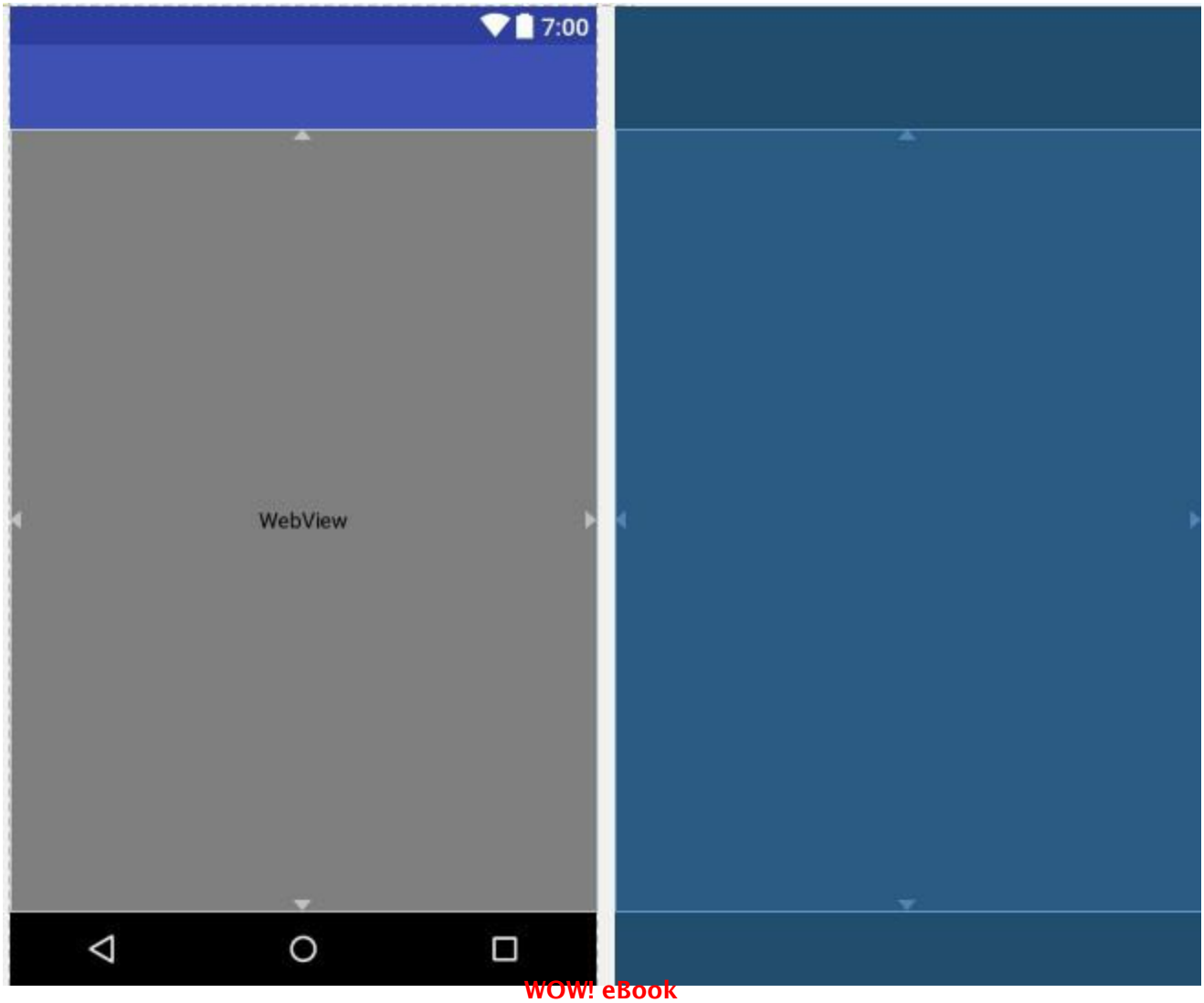

Figure 70-3

Select the newly added WebView instance and change the ID of the view to *myWebView*.

Before proceeding to the next step of this tutorial, an additional permission needs to be added to the project to enable the WebView object to access the internet and download a web page for printing. Add this permission by locating the *AndroidManifest.xml* file in the Project tool window and double-clicking on it to load it into the editing panel. Once loaded, edit the XML content to add the appropriate permission line as shown in the following listing:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.ebookfrenzy.webprint" >

    <uses-permission android:name="android.permission.INTERNET" />

    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme" >
        <activity
            android:name=".WebPrintActivity"
            android:label="@string/app_name"
            android:theme="@style/AppTheme.NoActionBar" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

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

</manifest>
```

70.6 Loading the Web Page into the WebView

Before the web page can be printed, it needs to be loaded into the WebView instance. For the purposes of this tutorial, this will be performed by a call to the *loadUrl()* method of the WebView instance, which will be placed in the *onCreate()* method of the WebPrintActivity class. Edit the *WebPrintActivity.java* file, therefore, and modify it as follows:

```
package com.ebookfrenzy.webprint;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.Menu;
import android.view.MenuItem;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import android.webkit.WebResourceRequest;

public class WebPrintActivity extends AppCompatActivity {
```

```
    private WebView myWebView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_web_print);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        myWebView = (WebView) findViewById(R.id.myWebView);
        myWebView.setWebViewClient(new WebViewClient(){
            @Override
            public boolean shouldOverrideUrlLoading(
                    WebView view, WebResourceRequest request) {
                return super.shouldOverrideUrlLoading(
                        view, request);
            }
        });
        myWebView.getSettings().setJavaScriptEnabled(true);
        myWebView.loadUrl(
                "https://developer.android.com/google/index.html");
    }
.
.
}
```

70.7 Adding the Print Menu Option

The option to print the web page will now be added to the Overflow menu using the techniques outlined in the chapter entitled *Creating and Managing Overflow Menus on Android*.

The first requirement is a string resource with which to label the menu option. Within the Project tool window, locate the *app -> res -> values -> strings.xml* file, double-click on it to load it into the editor and modify it to add a new string resource:

```
<resources>
    <string name="app_name">WebPrint</string>
    <string name="action_settings">Settings</string>
    <string name="print_string">Print</string>
</resources>
```

Next, load the *app -> res -> menu -> menu_web_print.xml* file into the menu editor, switch to Text mode and replace the *Settings* menu option with the print option:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    tools:context="com.ebookfrenzy.webprint.WebPrintActivity" >
    <item android:id="@+id/action_settings"
        android:title="@string/action_settings"
        android:orderInCategory="100"
        app:showAsAction="never" />

    <item
        android:id="@+id/action_print"
        android:orderInCategory="100"
        app:showAsAction="never"
        android:title="@string/print_string"/>
```

```
</menu>
```

All that remains in terms of configuring the menu option is to modify the *onOptionsItemSelected()* handler method within the *WebPrintActivity.java* file:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.action_print) {
        createWebPrintJob(myWebView);
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

With the *onOptionsItemSelected()* method implemented, the activity will call a method named *createWebPrintJob()* when the print menu option is selected from the overflow menu. The implementation of this method is identical to that used in the previous HTMLPrint project and may now be added to the *WebPrintActivity.java* file such that it reads as follows:

```
package com.ebookfrenzy.webprint;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.support.v7.widget.Toolbar;
import android.view.Menu;
import android.view.MenuItem;
import android.webkit.WebView;
import android.webkit.WebViewClient;
import android.webkit.WebResourceRequest;
import android.print.PrintAttributes;
import android.print.PrintDocumentAdapter;
import android.print.PrintManager;
import android.content.Context;

public class WebPrintActivity extends AppCompatActivity {

    private WebView myWebView;

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

        myWebView = (WebView) findViewById(R.id.myWebView);

        myWebView.loadUrl(
                "http://www.cnn.com");
    }

    private void createWebPrintJob(WebView webView) {

        PrintManager printManager = (PrintManager) this
                .getSystemService(Context.PRINT_SERVICE);
```

```
            PrintDocumentAdapter printAdapter =
                    webView.createPrintDocumentAdapter("MyDocument");

            String jobName = getString(R.string.app_name) +
                    " Print Test";

            printManager.print(jobName, printAdapter,
                    new PrintAttributes.Builder().build());
        }
    .
    .
    }
```

With the code changes complete, run the application on a physical Android device or emulator running Android version 5.0 or later. Once successfully launched, the WebView should be visible with the designated web page loaded. Once the page has loaded, select the Print option from the Overflow menu (Figure 70-4) and use the resulting print panel to print the web page to a suitable destination.

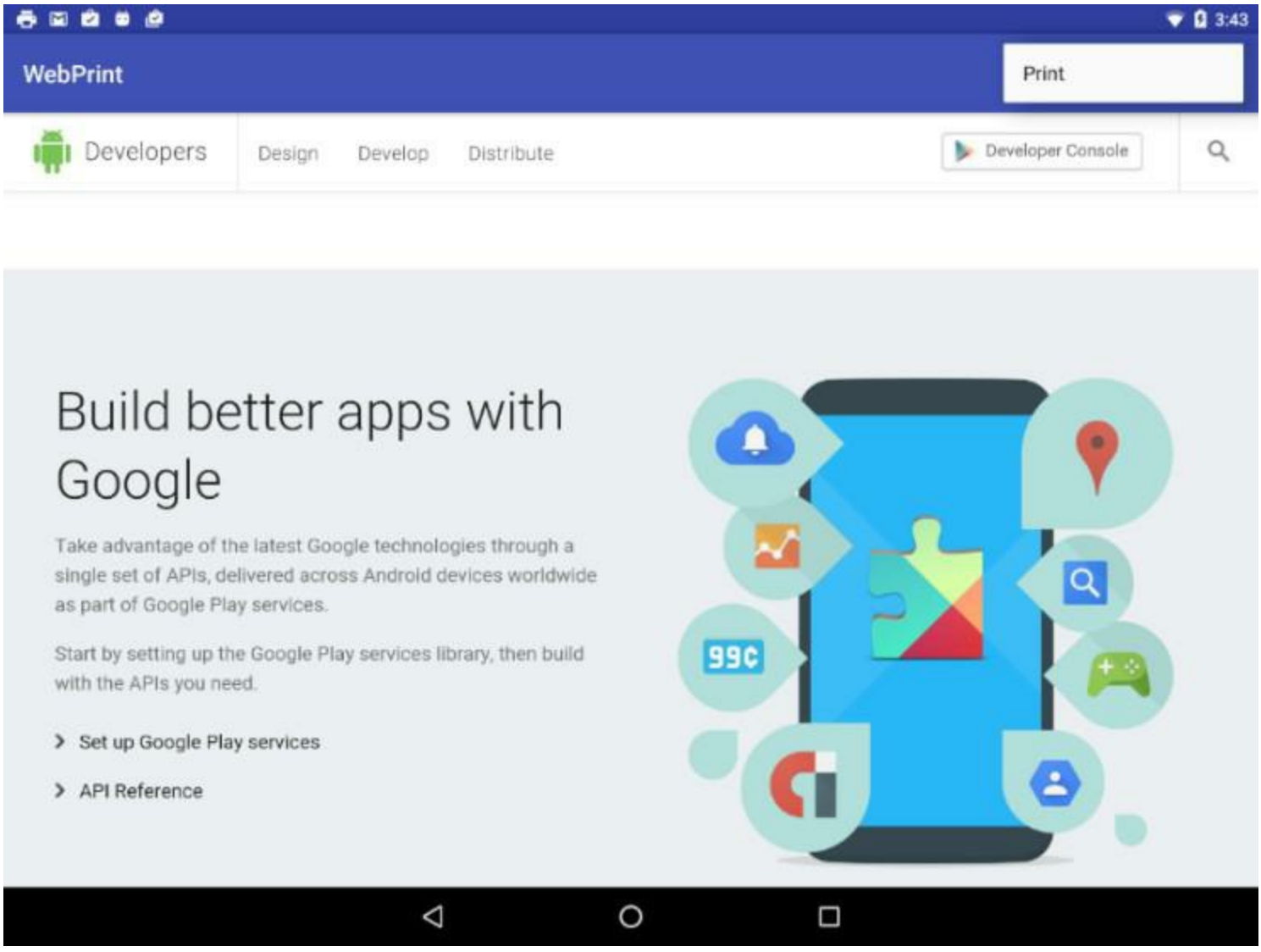

Figure 70-4

70.8 **Summary**

The Android Printing framework includes extensions to the WebView class that make it possible to print HTML based content from within an Android application. This content can be in the form of HTML created dynamically within the application at runtime, or a pre-existing web page loaded into

a WebView instance. In the case of dynamically created HTML, it is important to use a WebViewClient instance to ensure that printing does not start until the HTML has been fully loaded into the WebView.

71. A Guide to Android Custom Document Printing

As we have seen in the preceding chapters, the Android Printing framework makes it relatively easy to build printing support into applications as long as the content is in the form of an image or HTML markup. More advanced printing requirements can be met by making use of the custom document printing feature of the Printing framework.

71.1 An Overview of Android Custom Document Printing

In simplistic terms, custom document printing uses canvases to represent the pages of the document to be printed. The application draws the content to be printed onto these canvases in the form of shapes, colors, text and images. In actual fact, the canvases are represented by instances of the Android Canvas class, thereby providing access to a rich selection of drawing options. Once all the pages have been drawn, the document is then printed.

While this sounds simple enough, there are actually a number of steps that need to be performed to make this happen, which can be summarized as follows:

- Implement a custom print adapter sub-classed from the PrintDocumentAdapter class
- Obtain a reference to the Print Manager Service
- Create an instance of the PdfDocument class in which to store the document pages
- Add pages to the PdfDocument in the form of PdfDocument.Page instances
- Obtain references to the Canvas objects associated with the document pages
- Draw content onto the canvases
- Write the PDF document to a destination output stream provided by the Printing framework
- Notify the Printing framework that the document is ready to print

In this chapter, an overview of these steps will be provided, followed by a detailed tutorial designed to demonstrate the implementation of custom document printing within Android applications.

71.1.1 Custom Print Adapters

The role of the print adapter is to provide the Printing framework with the content to be printed, and to ensure that it is formatted correctly for the user's chosen preferences (taking into consideration factors such as paper size and page orientation).

When printing HTML and images, much of this work is performed by the print adapters provided as part of the Android Printing framework and designed for these specific printing tasks. When printing a web page, for example, a print adapter is created for us when a call is made to the *createPrintDocumentAdapter()* method of an instance of the WebView class.

In the case of custom document printing, however, it is the responsibility of the application developer to design the print adapter and implement the code to draw and format the content in preparation for printing.

Custom print adapters are created by sub-classing the PrintDocumentAdapter class and overriding a set of callback methods within that class which will be called by the Printing framework at various stages in the print process. These callback methods can be summarized as follows:

- **onStart()** – This method is called when the printing process begins and is provided so that the application code has an opportunity to perform any necessary tasks in preparation for creating the print job. Implementation of this method within the PrintDocumentAdapter sub-class is optional.
- **onLayout()** – This callback method is called after the call to the *onStart()* method and then again each time the user makes changes to the print settings (such as changing the orientation, paper size or color settings). This method should adapt the content and layout where necessary to accommodate these changes. Once these changes are completed, the method must return the number of pages to be printed. Implementation of the *onLayout()* method within the PrintDocumentAdapter sub-class is mandatory.
- **onWrite()** – This method is called after each call to *onLayout()* and is responsible for rendering the content on the canvases of the pages to be printed. Amongst other arguments, this method is passed a file descriptor to which the resulting PDF document must be written once rendering is complete. A call is then made to the *onWriteFinished()* callback method passing through an argument containing information about the page ranges to be printed. Implementation of the *onWrite()* method within the PrintDocumentAdapter sub-class is mandatory.
- **onFinish()** – An optional method which, if implemented, is called once by the Printing framework when the printing process is completed, thereby providing the application the opportunity to perform any clean-up operations that may be necessary.

71.2 Preparing the Custom Document Printing Project

Launch the Android Studio environment and create a new project, entering *CustomPrint* into the Application name field and *ebookfrenzy.com* as the Company Domain setting before clicking on the *Next* button.

On the form factors screen, enable the *Phone and Tablet* option and set the minimum SDK setting to API 21: Android 5.0 (Lollipop). Continue to proceed through the screens, requesting the creation of an Empty Activity named *CustomPrintActivity* with a corresponding layout resource file named *activity_custom_print*.

Load the *activity_custom_print.xml* layout file into the Layout Editor tool and, in Design mode, select and delete the “Hello World!” TextView object. Drag and drop a Button view from the Form Widgets section of the palette and position it in the center of the layout view. With the Button view selected, change the text property to “Print Document”, extract the string to a new string resource and change the layout_width property to *wrap_content*. On completion, the user interface layout should match that shown in Figure 71-1:

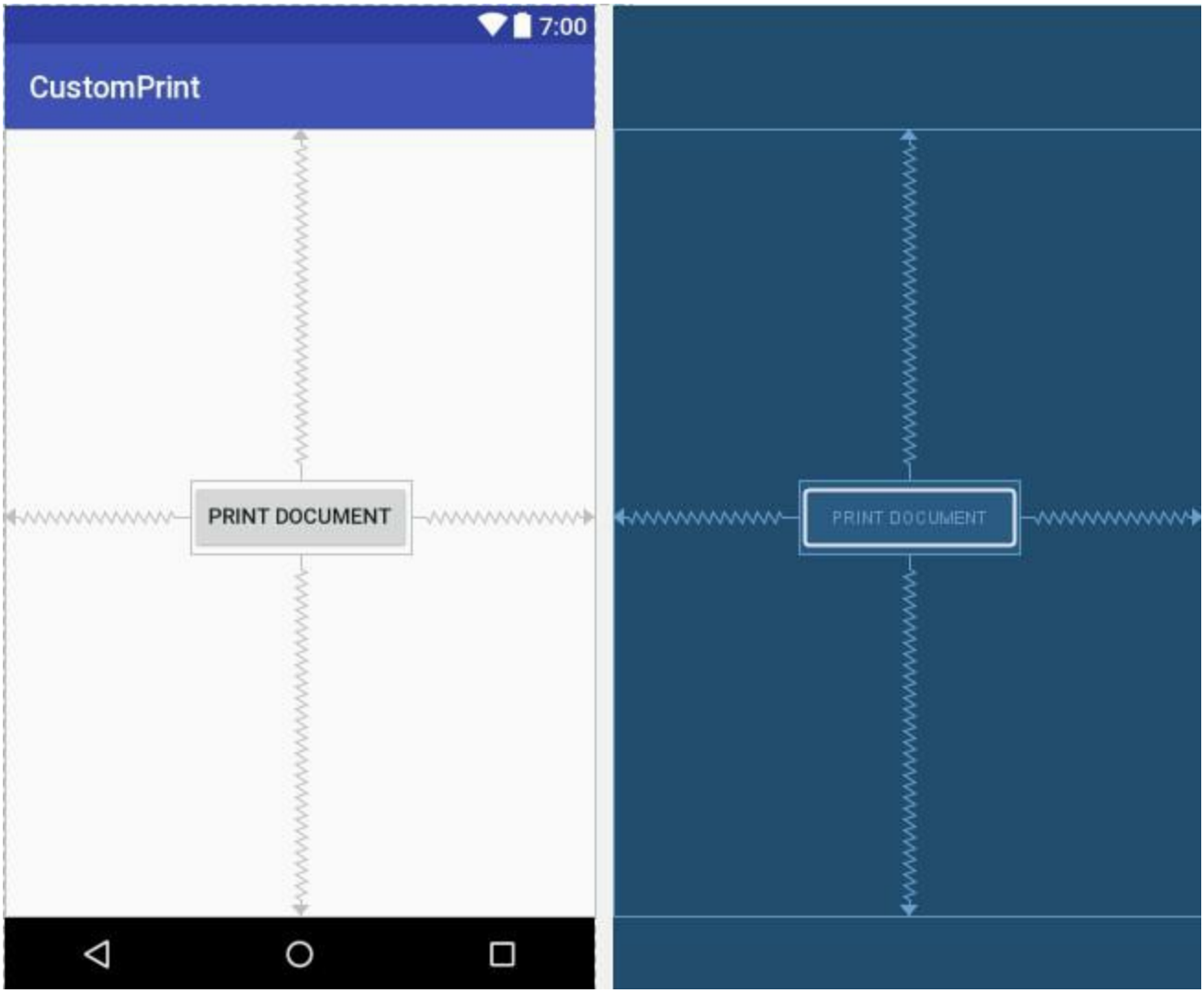

Figure 71-1

When the button is selected within the application it will be required to call a method to initiate the document printing process. Remaining within the Properties tool window, set the *onClick* property to call a method named *printDocument*.

71.3 Creating the Custom Print Adapter

Most of the work involved in printing a custom document from within an Android application involves the implementation of the custom print adapter. This example will require a print adapter with the *onLayout()* and *onWrite()* callback methods implemented. Within the *CustomPrintActivity.java* file, add the template for this new class so that it reads as follows:

```
package com.ebookfrenzy.customprint;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.os.CancellationSignal;
import android.os.ParcelFileDescriptor;
import android.print.PageRange;
import android.print.PrintAttributes;
import android.print.PrintDocumentAdapter;
import android.content.Context;

public class CustomPrintActivity extends AppCompatActivity {
```

```
    public class MyPrintDocumentAdapter extends PrintDocumentAdapter
    {
        Context context;

        public MyPrintDocumentAdapter(Context context)
        {
            this.context = context;
        }

        @Override
        public void onLayout(PrintAttributes oldAttributes,
                             PrintAttributes newAttributes,
                             CancellationSignal cancellationSignal,
                             LayoutResultCallback callback,
                             Bundle metadata) {
        }

        @Override
        public void onWrite(final PageRange[] pageRanges,
                            final ParcelFileDescriptor destination,
                            final CancellationSignal
                                    cancellationSignal,
                            final WriteResultCallback callback) {
        }

    }
.
.
}
```

As the new class currently stands, it contains a constructor method which will be called when a new instance of the class is created. The constructor takes as an argument the context of the calling activity which is then stored so that it can be referenced later in the two callback methods.

With the outline of the class established, the next step is to begin implementing the two callback methods, beginning with *onLayout()*.

71.4 Implementing the onLayout() Callback Method

Remaining within the *CustomPrintActivity.java* file, begin by adding some import directives that will be required by the code in the *onLayout()* method:

```
package com.ebookfrenzy.customprint;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.os.CancellationSignal;
import android.os.ParcelFileDescriptor;
import android.print.PageRange;
import android.print.PrintAttributes;
import android.print.PrintDocumentAdapter;
import android.content.Context;
import android.print.PrintDocumentInfo;
import android.print.pdf.PrintedPdfDocument;
import android.graphics.pdf.PdfDocument;

public class CustomPrintActivity extends AppCompatActivity {
```

```
.
.
.
}
```

Next, modify the MyPrintDocumentAdapter class to declare variables to be used within the *onLayout()* method:

```
public class MyPrintDocumentAdapter extends PrintDocumentAdapter
{
        Context context;
        private int pageHeight;
        private int pageWidth;
        public PdfDocument myPdfDocument;
        public int totalpages = 4;
.
.
}
```

Note that for the purposes of this example, a four page document is going to be printed. In more complex situations, the application will most likely need to dynamically calculate the number of pages to be printed based on the quantity and layout of the content in relation to the user's paper size and page orientation selections.

With the variables declared, implement the *onLayout()* method as outlined in the following code listing:

```
@Override
public void onLayout(PrintAttributes oldAttributes,
                     PrintAttributes newAttributes,
                     CancellationSignal cancellationSignal,
                     LayoutResultCallback callback,
                     Bundle metadata) {

        myPdfDocument = new PrintedPdfDocument(context, newAttributes);

        pageHeight =
               newAttributes.getMediaSize().getHeightMils()/1000 * 72;
        pageWidth =
               newAttributes.getMediaSize().getWidthMils()/1000 * 72;

        if (cancellationSignal.isCanceled() ) {
               callback.onLayoutCancelled();
               return;
        }

        if (totalpages > 0) {
           PrintDocumentInfo.Builder builder = new PrintDocumentInfo
                  .Builder("print_output.pdf").setContentType(
                        PrintDocumentInfo.CONTENT_TYPE_DOCUMENT)
                  .setPageCount(totalpages);

           PrintDocumentInfo info = builder.build();
           callback.onLayoutFinished(info, true);
        } else {
           callback.onLayoutFailed("Page count is zero.");
```

```
        }
    }
```

Clearly this method is performing quite a few tasks, each of which requires some detailed explanation.

To begin with, a new PDF document is created in the form of a PdfDocument class instance. One of the arguments passed into the *onLayout()* method when it is called by the Printing framework is an object of type PrintAttributes containing details about the paper size, resolution and color settings selected by the user for the print output. These settings are used when creating the PDF document, along with the context of the activity previously stored for us by our constructor method:

```
myPdfDocument = new PrintedPdfDocument(context, newAttributes);
```

The method then uses the PrintAttributes object to extract the height and width values for the document pages. These dimensions are stored in the object in the form of thousandths of an inch. Since the methods that will use these values later in this example work in units of 1/72 of an inch these numbers are converted before they are stored:

```
pageHeight = newAttributes.getMediaSize().getHeightMils()/1000 * 72;
pageWidth = newAttributes.getMediaSize().getWidthMils()/1000 * 72;
```

Although this example does not make use of the user's color selection, this property can be obtained via a call to the *getColorMode()* method of the PrintAttributes object which will return a value of either COLOR_MODE_COLOR or COLOR_MODE_MONOCHROME.

When the *onLayout()* method is called, it is passed an object of type *LayoutResultCallback*. This object provides a way for the method to communicate status information back to the Printing framework via a set of methods. The *onLayout()* method, for example, will be called in the event that the user cancels the print process. The fact that the process has been cancelled is indicated via a setting within the CancellationSignal argument. In the event that a cancellation is detected, the *onLayout()* method must call the *onLayoutCancelled()* method of the *LayoutResultCallback* object to notify the Print framework that the cancellation request was received and that the layout task has been cancelled:

```
if (cancellationSignal.isCanceled() ) {
        callback.onLayoutCancelled();
        return;
}
```

When the layout work is complete, the method is required to call the *onLayoutFinished()* method of the *LayoutResultCallback* object, passing through two arguments. The first argument takes the form of a PrintDocumentInfo object containing information about the document to be printed. This information consists of the name to be used for the PDF document, the type of content (in this case a document rather than an image) and the page count. The second argument is a Boolean value indicating whether or not the layout has changed since the last call made to the *onLayout()* method:

```
if (totalpages > 0) {
        PrintDocumentInfo.Builder builder = new PrintDocumentInfo
                .Builder("print_output.pdf")
                .setContentType(
                    PrintDocumentInfo.CONTENT_TYPE_DOCUMENT)
                .setPageCount(totalpages);

        PrintDocumentInfo info = builder.build();
```

```
        callback.onLayoutFinished(info, true);
    } else {
        callback.onLayoutFailed("Page count is zero.");
    }
```

In the event that the page count is zero, the code reports this failure to the Printing framework via a call to the *onLayoutFailed()* method of the *LayoutResultCallback* object.

The call to the *onLayoutFinished()* method notifies the Printing framework that the layout work is complete, thereby triggering a call to the *onWrite()* method.

71.5 Implementing the onWrite() Callback Method

The *onWrite()* callback method is responsible for rendering the pages of the document and then notifying the Printing framework that the document is ready to be printed. When completed, the *onWrite()* method reads as follows:

```
package com.ebookfrenzy.customprint;

import java.io.FileOutputStream;
import java.io.IOException;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.os.CancellationSignal;
import android.os.ParcelFileDescriptor;
import android.print.PageRange;
import android.print.PrintAttributes;
import android.print.PrintDocumentAdapter;
import android.content.Context;
import android.print.PrintDocumentInfo;
import android.print.pdf.PrintedPdfDocument;
import android.graphics.pdf.PdfDocument;
import android.graphics.pdf.PdfDocument.PageInfo;
.
.
.
.
@Override
public void onWrite(final PageRange[] pageRanges,
                    final ParcelFileDescriptor destination,
                    final CancellationSignal cancellationSignal,
                    final WriteResultCallback callback) {

        for (int i = 0; i < totalpages; i++) {
               if (pageInRange(pageRanges, i))
                {
                    PageInfo newPage = new PageInfo.Builder(pageWidth,
                          pageHeight, i).create();

                    PdfDocument.Page page =
                           myPdfDocument.startPage(newPage);

                    if (cancellationSignal.isCanceled()) {
                         callback.onWriteCancelled();
                         myPdfDocument.close();
```

```
                        myPdfDocument = null;
                        return;
                    }
                    drawPage(page, i);
                    myPdfDocument.finishPage(page);
                }
            }

            try {
                myPdfDocument.writeTo(new FileOutputStream(
                            destination.getFileDescriptor()));
            } catch (IOException e) {
                callback.onWriteFailed(e.toString());
                return;
            } finally {
                myPdfDocument.close();
                myPdfDocument = null;
            }

            callback.onWriteFinished(pageRanges);
    }
```

The *onWrite()* method starts by looping through each of the pages in the document. It is important to take into consideration, however, that the user may not have requested that all of the pages that make up the document be printed. In actual fact, the Printing framework user interface panel provides the option to specify that specific pages, or ranges of pages be printed. Figure 71-2, for example, shows the print panel configured to print pages 1-4, pages 8 and 9 and pages 11-13 of a document.

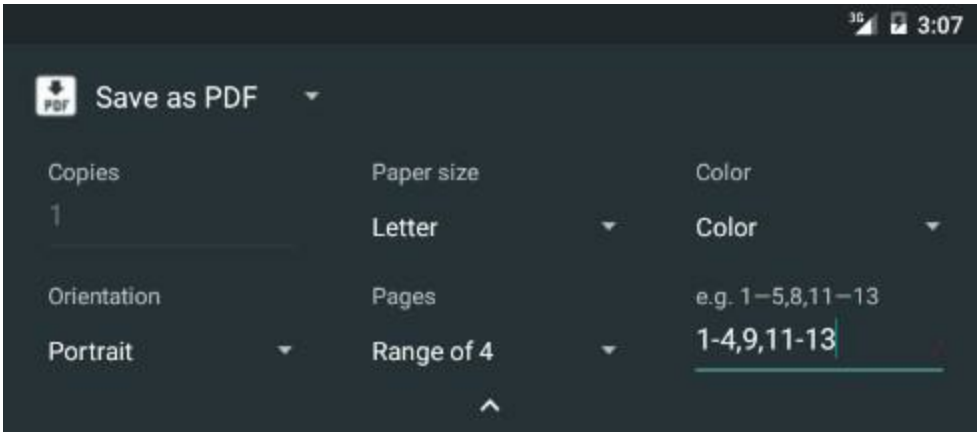

Figure 71-2

When writing the pages to the PDF document, the *onWrite()* method must take steps to ensure that only those pages specified by the user are printed. To make this possible, the Printing framework passes through as an argument an array of PageRange objects indicating the ranges of pages to be printed. In the above *onWrite()* implementation, a method named *pagesInRange()* is called for each page to verify that the page is within the specified ranges. The code for the *pagesInRange()* method will be implemented later in this chapter.

```
    for (int i = 0; i < totalpages; i++) {
            if (pageInRange(pageRanges, i))
            {
```

For each page that is within any specified ranges, a new PdfDocument.Page object is created. When creating a new page, the height and width values previously stored by the *onLayout()* method are passed through as arguments so that the page size matches the print options selected by the user:

```
PageInfo newPage = new PageInfo.Builder(pageWidth,
                                pageHeight, i).create();

PdfDocument.Page page = myPdfDocument.startPage(newPage);
```

As with the *onLayout()* method, the *onWrite()* method is required to respond to cancellation requests. In this case, the code notifies the Printing framework that the cancellation has been performed, before closing and de-referencing the myPdfDocument variable:

```
if (cancellationSignal.isCanceled()) {
        callback.onWriteCancelled();
        myPdfDocument.close();
        myPdfDocument = null;
        return;
}
```

As long as the print process has not been cancelled, the method then calls a method to draw the content on the current page before calling the *finishedPage()* method on the myPdfDocument object.

```
drawPage(page, i);
myPdfDocument.finishPage(page);
```

The *drawPage()* method is responsible for drawing the content onto the page and will be implemented once the *onWrite()* method is complete.

When the required number of pages have been added to the PDF document, the document is then written to the *destination* stream using the file descriptor which was passed through as an argument to the *onWrite()* method. If, for any reason, the write operation fails, the method notifies the framework by calling the *onWriteFailed()* method of the *WriteResultCallback* object (also passed as an argument to the *onWrite()* method).

```
try {
        myPdfDocument.writeTo(new FileOutputStream(
                destination.getFileDescriptor()));
} catch (IOException e) {
                callback.onWriteFailed(e.toString());
                return;
} finally {
                myPdfDocument.close();
                myPdfDocument = null;
}
```

Finally, the *onWriteFinish()* method of the *WriteResultsCallback* object is called to notify the Printing framework that the document is ready to be printed.

71.6 Checking a Page is in Range

As previously outlined, when the *onWrite()* method is called it is passed an array of PageRange objects indicating the ranges of pages within the document that are to be printed. The PageRange class is designed to store the start and end pages of a page range which, in turn, may be accessed via the *getStart()* and *getEnd()* methods of the class.

When the *onWrite()* method was implemented in the previous section, a call was made to a method named *pageInRange()*, which takes as arguments an array of PageRange objects and a page number. The role of the *pageInRange()* method is to identify whether the specified page number is within the ranges specified and may be implemented within the MyPrintDocumentAdapter class in the

CustomPrintActivity.java class as follows:

```
public class MyPrintDocumentAdapter extends PrintDocumentAdapter
{
.
.
.
        private boolean pageInRange(PageRange[] pageRanges, int page)
        {
                for (int i = 0; i<pageRanges.length; i++)
                {
                        if ((page >= pageRanges[i].getStart()) &&
                                        (page <= pageRanges[i].getEnd()))
                                return true;
                }
                return false;
        }
.
.
}
```

71.7 Drawing the Content on the Page Canvas

We have now reached the point where some code needs to be written to draw the content on the pages so that they are ready for printing. The content that gets drawn is completely application specific and limited only by what can be achieved using the Android Canvas class. For the purposes of this example, however, some simple text and graphics will be drawn on the canvas.

The *onWrite()* method has been designed to call a method named *drawPage()* which takes as arguments the PdfDocument.Page object representing the current page and an integer representing the page number. Within the *CustomPrintActivity.java* file this method should now be implemented as follows:

```
package com.ebookfrenzy.customprint;

import java.io.FileOutputStream;
import java.io.IOException;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.os.CancellationSignal;
import android.os.ParcelFileDescriptor;
import android.print.PageRange;
import android.print.PrintAttributes;
import android.print.PrintDocumentAdapter;
import android.content.Context;
import android.print.PrintDocumentInfo;
import android.print.pdf.PrintedPdfDocument;
import android.graphics.pdf.PdfDocument;
import android.graphics.pdf.PdfDocument.PageInfo;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;

public class CustomPrintActivity extends AppCompatActivity {
.
```

```
.
        public class MyPrintDocumentAdapter extends
                                        PrintDocumentAdapter
        {
.
.
                private void drawPage(PdfDocument.Page page,
                                      int pagenumber) {
                    Canvas canvas = page.getCanvas();

                    pagenumber++; // Make sure page numbers start at 1

                    int titleBaseLine = 72;
                    int leftMargin = 54;

                    Paint paint = new Paint();
                    paint.setColor(Color.BLACK);
                    paint.setTextSize(40);
                    canvas.drawText(
                       "Test Print Document Page " + pagenumber,
                                                    leftMargin,
                                                    titleBaseLine,
                                                    paint);

                    paint.setTextSize(14);
                    canvas.drawText("This is some test content to verify that
custom document printing works", leftMargin, titleBaseLine + 35, paint);

                    if (pagenumber % 2 == 0)
                                paint.setColor(Color.RED);
                    else
                        paint.setColor(Color.GREEN);

                    PageInfo pageInfo = page.getInfo();


                    canvas.drawCircle(pageInfo.getPageWidth()/2,
                                      pageInfo.getPageHeight()/2,
                                      150,
                                      paint);
                }
.
.
}
```

Page numbering within the code starts at 0. Since documents traditionally start at page 1, the method begins by incrementing the stored page number. A reference to the Canvas object associated with the page is then obtained and some margin and baseline values declared:

```
Canvas canvas = page.getCanvas();

pagenumber++;

int titleBaseLine = 72;
int leftMargin = 54;
```

Next, the code creates Paint and Color objects to be used for drawing, sets a text size and draws the

page title text, including the current page number:

```
Paint paint = new Paint();

paint.setColor(Color.BLACK);
paint.setTextSize(40);

canvas.drawText("Test Print Document Page " + pagenumber,
                                             leftMargin,
                                             titleBaseLine,
                                             paint);
```

The text size is then reduced and some body text drawn beneath the title:

```
paint.setTextSize(14);

canvas.drawText("This is some test content to verify that custom document
printing works", leftMargin, titleBaseLine + 35, paint);
```

The last task performed by this method involves drawing a circle (red on even numbered pages and green on odd). Having ascertained whether the page is odd or even, the method obtains the height and width of the page before using this information to position the circle in the center of the page:

```
if (pagenumber % 2 == 0)
       paint.setColor(Color.RED);
else
       paint.setColor(Color.GREEN);

PageInfo pageInfo = page.getInfo();

canvas.drawCircle(pageInfo.getPageWidth()/2,
                  pageInfo.getPageHeight()/2,
                  150, paint);
```

Having drawn on the canvas, the method returns control to the *onWrite()* method.

With the completion of the *drawPage()* method, the MyPrintDocumentAdapter class is now finished.

71.8 Starting the Print Job

When the "Print Document" button is touched by the user, the *printDocument()* onClick event handler method will be called. All that now remains before testing can commence, therefore, is to add this method to the *CustomPrintActivity.java* file, taking particular care to ensure that it is placed outside of the MyPrintDocumentAdapter class:

```
package com.ebookfrenzy.customprint;

import java.io.FileOutputStream;
import java.io.IOException;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.os.CancellationSignal;
import android.os.ParcelFileDescriptor;
import android.print.PageRange;
import android.print.PrintAttributes;
import android.print.PrintDocumentAdapter;
```

```
import android.content.Context;
import android.print.PrintDocumentInfo;
import android.print.pdf.PrintedPdfDocument;
import android.graphics.pdf.PdfDocument;
import android.graphics.pdf.PdfDocument.PageInfo;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.print.PrintManager;
import android.view.View;

public class CustomPrintActivity extends AppCompatActivity {

        public void printDocument(View view)
        {
            PrintManager printManager = (PrintManager) this
                    .getSystemService(Context.PRINT_SERVICE);

            String jobName = this.getString(R.string.app_name) +
                        " Document";

            printManager.print(jobName, new
                     MyPrintDocumentAdapter(this),
                     null);
        }
.
.
.
}
```

This method obtains a reference to the Print Manager service running on the device before creating a new String object to serve as the job name for the print task. Finally the *print()* method of the Print Manager is called to start the print job, passing through the job name and an instance of our custom print document adapter class.

71.9 Testing the Application

Compile and run the application on an Android device or emulator that is running Android 4.4 or later. When the application has loaded, touch the "Print Document" button to initiate the print job and select a suitable target for the output (the Save to PDF option is a useful option for avoiding wasting paper and printer ink).

Check the printed output which should consist of 4 pages including text and graphics. Figure 71-3, for example, shows the four pages of the document viewed as a PDF file ready to be saved on the device.

Experiment with other print configuration options such as changing the paper size, orientation and pages settings within the print panel. Each setting change should be reflected in the printed output, indicating that the custom print document adapter is functioning correctly.

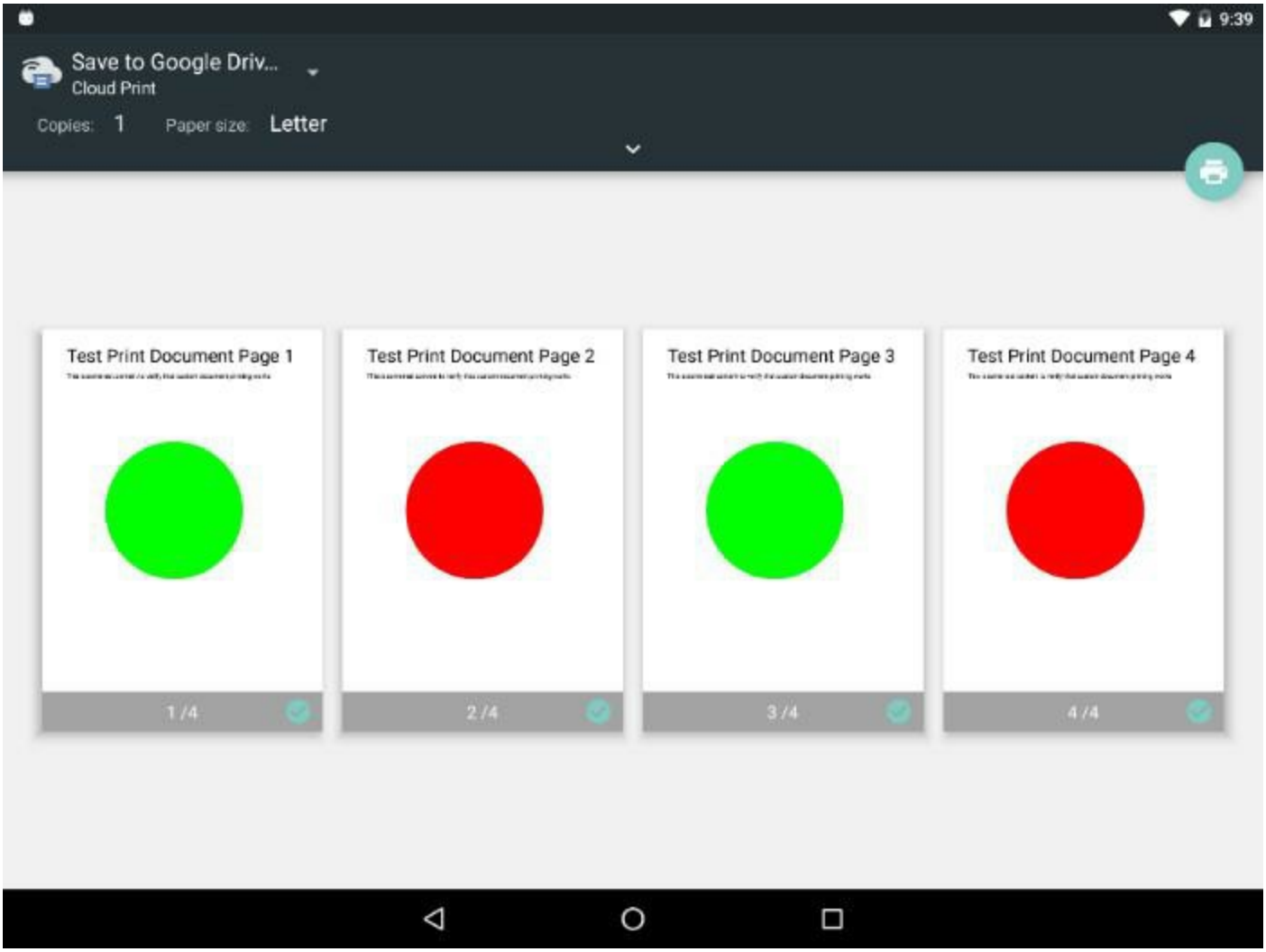

Figure 71-3

71.10 Summary

Although more complex to implement than the Android Printing framework HTML and image printing options, custom document printing provides considerable flexibility in terms of printing complex content from within an Android application. The majority of the work involved in implementing custom document printing involves the creation of a custom Print Adapter class such that it not only draws the content on the document pages, but also responds correctly as changes are made by the user to print settings such as the page size and range of pages to be printed.