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" and extract the string to a new string. On completion, the user interface layout should match that shown in Figure 71-1:

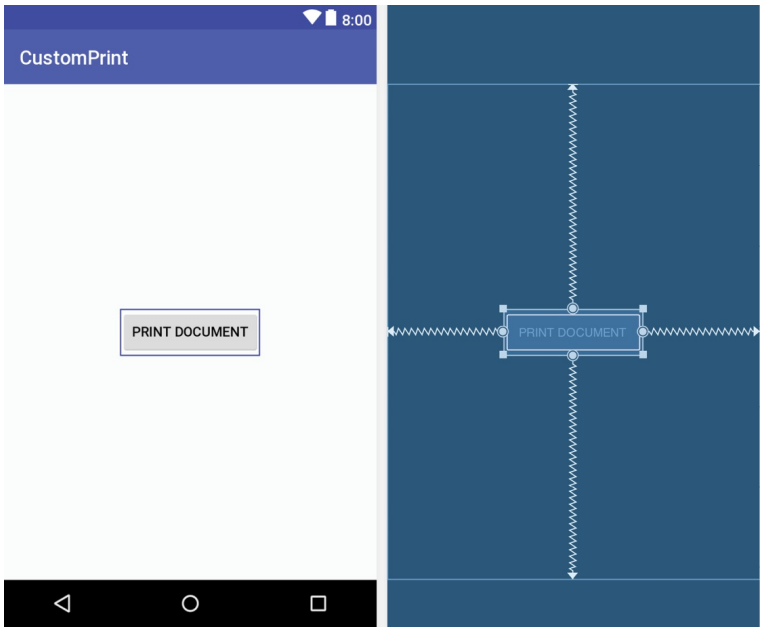

Figure 71-1

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 Attributes 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.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.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.

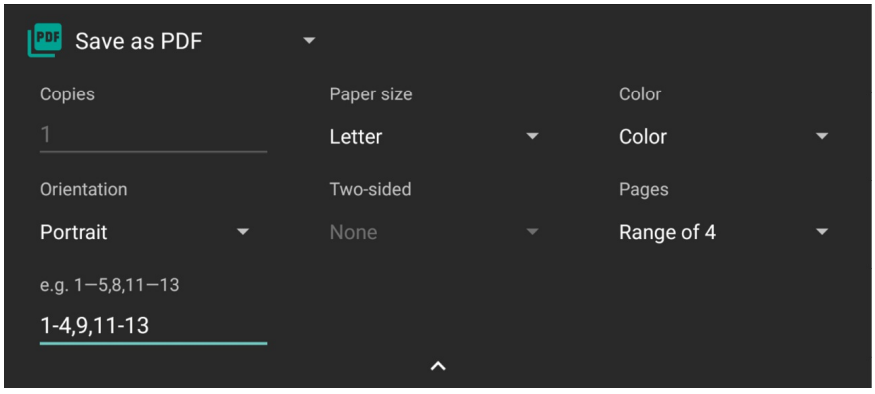

Figure 71-2

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 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 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.

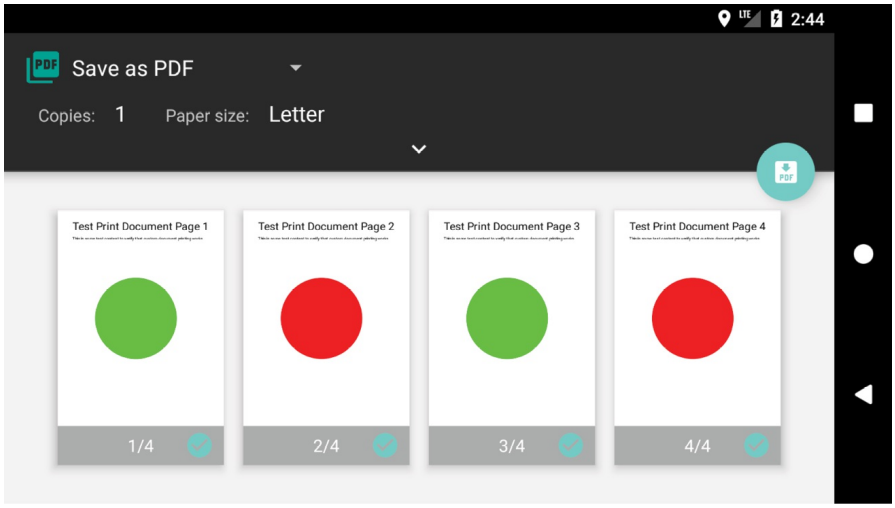

Figure 71-3

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.