CHAPTER 29

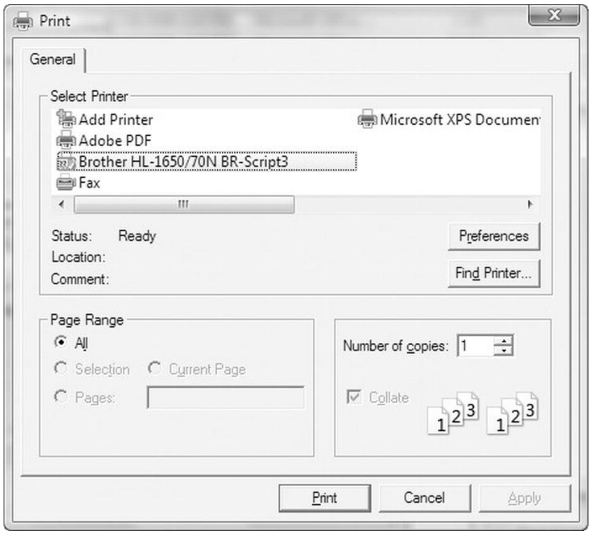
Printing

WPF includes a revamped printing model that organizes all your coding around a single ingredient: the PrintDialog class in the System.Windows.Controls namespace. Using the PrintDialog class, you can show a Print dialog box where the user can pick a printer and change its setting, and you can send elements, documents, and low-level visuals directly to the printer. In this chapter, you'll learn how to use the PrintDialog class to create properly scaled and paginated printouts.

Basic Printing

Although WPF includes dozens of print-related classes (most of which are found in the System.Printing namespace), there's a single starting point that makes life easy: the PrintDialog class.

The PrintDialog wraps the familiar Print dialog box that lets the user choose the printer and a few other standard print options, such as the number of copies (see Figure 29-1). However, the PrintDialog class is more than just a pretty window—it also has the built-in ability to trigger a printout.



Status: Ready
Location:
Comment:
Page Range: [✓ All]
Number of copies: 1
[✓ Collate]

Figure 29-1. *Showing the PrintDialog*

To submit a print job with the PrintDialog class, you need to use one of two methods:

- *PrintVisual()* works with any class that derives from System.Windows.Media.Visual. This includes any graphic you draw by hand and any element you place in a window.
- *PrintDocument()* works with any DocumentPaginator object. This includes the ones that are used to split a FlowDocument (or XpsDocument) into pages and any custom DocumentPaginator you create to deal with your own data.

In the following sections, you'll consider a variety of strategies that you can use to create a printout.

Printing an Element

The simplest approach to printing is to take advantage of the model you're already using for onscreen rendering. Using the PrintDialog.PrintVisual() method, you can send any element in a window (and all its children) straight to the printer.

To see an example in action, consider the window shown in Figure 29-2. It contains a Grid that lays out all the elements. In the topmost row is a Canvas, and in that Canvas is a drawing that consists of a TextBlock and a Path (which renders itself as a rectangle with an elliptic hole in the middle).

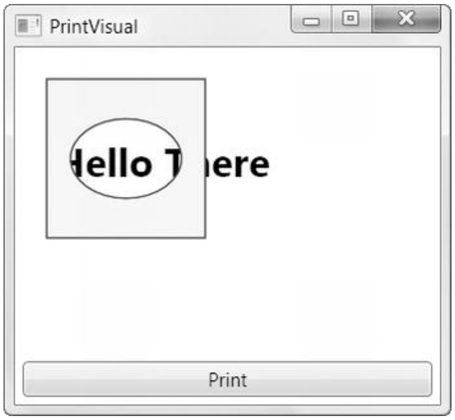


Figure 29-2. *A simple drawing*

To send the Canvas to the printer, complete with all the elements it contains, you can use this snippet of code when the Print button is clicked:

```
PrintDialog printDialog = new PrintDialog();
if (printDialog.ShowDialog() == true)
{
    printDialog.PrintVisual(canvas, "A Simple Drawing");
}
```

The first step is to create a PrintDialog object. The next step is to call ShowDialog() to show the Print dialog box. ShowDialog returns a nullable Boolean value. A return value of true indicates that the user clicked OK, a return value of false indicates that the user clicked Cancel, and a null value indicates that the dialog box was closed without either button being clicked.

When calling the PrintVisual() method, you pass two arguments. The first is the element that you want to print, and the second is a string that's used to identify the print job. You'll see it appear in the Windows print queue (under the Document Name column).

When printing this way, you don't have much control over the output. The element is always lined up with the top-left corner of the page. If your element doesn't include nonzero Margin values, the edge of your content might land in the nonprintable area of the page, which means it won't appear in the printed output.

The lack of margin control is only the beginning of the limitations that you'll face using this approach. You also can't paginate your content if it's extremely long, so if you have more content than can fit on a single page, some will be left out at the bottom. Finally, you have no control over the scaling that's used to render your job to the printing. Instead, WPF uses the same device-independent rendering system based on 1/96th-inch units. For example, if you have a rectangle that's 96 units wide, that rectangle will appear to be an inch wide on your monitor (assuming you're using the standard 96 dpi Windows system setting) and an inch wide on the printed page. Often, this results in a printout that's quite a bit smaller than what you want.

■ **Note** Obviously, WPF fills in much more detail in the printed page, because virtually no printer has a resolution as low as 96 dpi (600 dpi and 1200 dpi are much more common printer resolutions). However, WPF keeps your content the same size in the printout as it is on your monitor.

Figure 29-3 shows the full-page printout of the Canvas from the window shown in Figure 29-2.

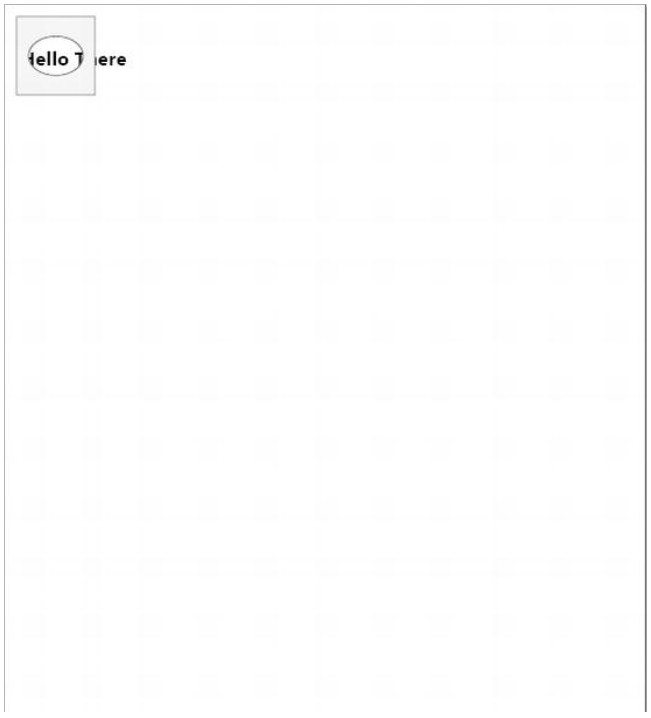


Figure 29-3. *A printed element*

PRINTDIALOG QUIRKS

The PrintDialog class wraps a lower-level internal .NET class named Win32PrintDialog, which in turns wraps the Print dialog box that's exposed by the Win32 API. Unfortunately, these extra layers remove a little bit of your flexibility.

One potential problem is the way that the PrintDialog class works with modal windows. Buried in the inaccessible Win32PrintDialog code is a bit of logic that always makes the Print dialog box modal with respect to your application's *main* window. This leads to an odd problem if you show a modal window from your main window and then call the PrintDialog.ShowDialog() method from that window. Although you'd expect the Print dialog box to be modal to your second window, it will actually be modal with respect to your main window, which means the user can return to your second window and interact with it (even clicking the Print button to show multiple instances of the Print dialog box)! The somewhat clumsy solution is to manually change your application's main window to the current window before you call PrintDialog.ShowDialog() and then switch it back immediately afterward.

There's another limitation to the way the PrintDialog class works. Because your main application thread owns the content you're printing, it's not possible to perform your printing on a background thread. This becomes a concern if you have time-consuming printing logic. Two possible solutions exist. If you construct the visuals you want to print on the background thread (rather than pulling them out of an existing window), you'll be able to perform your printing on the background thread. However, a simpler solution is to use the PrintDialog box to let the user specify the print settings and then use the XpsDocumentWriter class to actually print the content instead of the printing methods of the PrintDialog class. The XpsDocumentWriter can send content to the printer asynchronously, and it's described in the "Printing Through XPS" section later in this chapter.

Transforming Printed Output

You may remember (from Chapter 12) that you can attach the Transform object to the RenderTransform or LayoutTransform property of any element to change the way it's rendered. Transform objects could solve the problem of inflexible printouts, because you could use them to resize an element (ScaleTransform), move it around the page (TranslateTransform), or both (TransformGroup). Unfortunately, visuals can lay themselves out only one way at a time. That means there's no way to scale an element one way in a window and another way in a printout—instead, any Transform objects you apply will change both the printed output and the onscreen appearance of your element.

If you aren't intimidated by a bit of messy compromise, you can work around this issue in several ways. The basic idea is to apply your transform objects just before you create the printout and then remove them. To prevent the resized element from appearing in the window, you can temporarily hide it.

You might expect to hide your element by changing its Visibility property, but this hides your element from both the window and the printout, which obviously isn't what you want. One possible solution is to change the Visibility of the parent (in this example, the layout Grid). This works because the PrintVisual() method considers only the element you specify and its children, not the details of the parent.

Here's the code that puts it all together and prints the Canvas shown in Figure 29-2, but five times bigger in both dimensions:

```
PrintDialog printDialog = new PrintDialog();
if (printDialog.ShowDialog() == true)
{
    // Hide the Grid.
    grid.Visibility = Visibility.Hidden;
    // Magnify the output by a factor of 5.
    canvas.LayoutTransform = new ScaleTransform(5, 5);

    // Print the element.
    printDialog.PrintVisual(canvas, "A Scaled Drawing");

    // Remove the transform and make the element visible again.
    canvas.LayoutTransform = null;
    grid.Visibility = Visibility.Visible;
}
```

This example has one missing detail. Although the Canvas (and its contents) is stretched, the Canvas is still using the layout information from the containing Grid. In other words, the Canvas still believes it has an amount of space to work with that's equal to the dimensions of the Grid cell in which it's placed. In this example, this oversight doesn't cause a problem, because the Canvas doesn't limit itself to the available space (unlike some other containers). However, you will run into trouble if you have text and you want it to wrap to fit the bounds of the printed page or if your Canvas has a background (which, in this example, will occupy the smaller size of the Grid cell rather than the whole area behind the Canvas).

The solution is easy. After you set the LayoutTransform (but before you print the Canvas), you need to trigger the layout process manually using the Measure() and Arrange() methods that every element inherits from the UIElement class. The trick is that when you call these methods, you'll pass in the size of the page, so the Canvas stretches itself to fit. (Incidentally, this is also why you set the LayoutTransform instead of the RenderTransform property, because you want the layout to take the newly expanded size into account.) You can get the page size from the PrintableAreaWidth and PrintableAreaHeight properties.

■ **Note** Based on the property names, it's reasonable to assume that PrintableAreaWidth and PrintableAreaHeight reflect the *printable* area of the page—in other words, the part of the page on which the printer can actually print. (Most printers can't reach the very edges, usually because that's where the rollers grip onto the page.) But in truth, PrintableAreaWidth and PrintableAreaHeight simply return the *full* width and height of the page in device-independent units. For a sheet of 8.5x11 paper, that's 816 and 1056. (Try dividing these numbers by 96 dpi, and you'll get the full paper size.)

The following example demonstrates how to use the PrintableAreaWidth and PrintableAreaHeight properties. To be a bit nicer, it leaves off 10 units (about 0.1 of an inch) as a border around all edges of the page.

```
PrintDialog printDialog = new PrintDialog();
if (printDialog.ShowDialog() == true)
{
    // Hide the Grid.
    grid.Visibility = Visibility.Hidden;

    // Magnify the output by a factor of 5.
    canvas.LayoutTransform = new ScaleTransform(5, 5);

    // Define a margin.
    int pageMargin = 5;
    // Get the size of the page.
    Size pageSize = new Size(printDialog.PrintableAreaWidth - pageMargin * 2,
      printDialog.PrintableAreaHeight - 20);

    // Trigger the sizing of the element.
    canvas.Measure(pageSize);
    canvas.Arrange(new Rect(pageMargin, pageMargin,
      pageSize.Width, pageSize.Height));

    // Print the element.
    printDialog.PrintVisual(canvas, "A Scaled Drawing");

    // Remove the transform and make the element visible again.
    canvas.LayoutTransform = null;
    grid.Visibility = Visibility.Visible;
}
```

The end result is a way to print any element and scale it to suit your needs (see the full-page printout in Figure 29-4). This approach works perfectly well, but you can see the (somewhat messy) glue that's holding it all together.

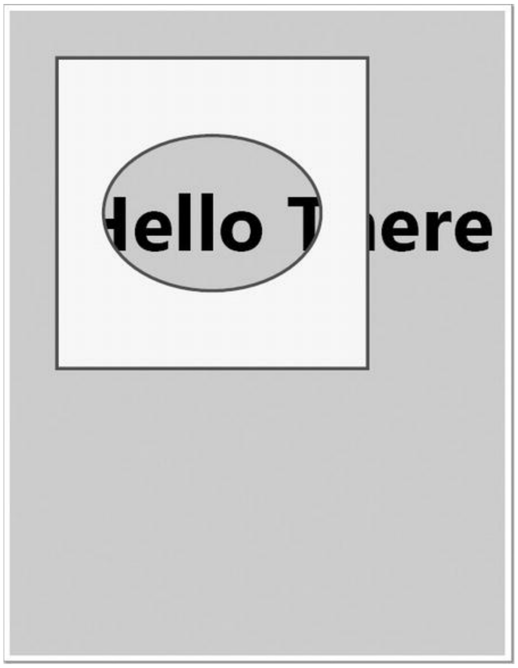


Figure 29-4. *A scaled printed element*

Printing Elements Without Showing Them

Because the way you want to show data in your application and the way you want it to appear in a printout are often different, it sometimes makes sense to create your visual programmatically (rather than using one that appears in an existing window). For example, the following code creates an in-memory TextBlock object, fills it with text, sets it to wrap, sizes it to fit the printed page, and then prints it:

```
PrintDialog printDialog = new PrintDialog();
if (printDialog.ShowDialog() == true)
{
    // Create the text.
    Run run = new Run("This is a test of the printing functionality " +
      "in the Windows Presentation Foundation.");

    // Wrap it in a TextBlock.
    TextBlock visual = new TextBlock();
    TextBlock.Inlines.Add(run);

    // Use margin to get a page border.
    visual.Margin = new Thickness(15);

    // Allow wrapping to fit the page width.
    visual.TextWrapping = TextWrapping.Wrap;

    // Scale the TextBlock up in both dimensions by a factor of 5.
    // (In this case, increasing the font would have the same effect,
    // because the TextBlock is the only element.)
    visual.LayoutTransform = new ScaleTransform(5, 5);

    // Size the element.
    Size pageSize = new Size(printDialog.PrintableAreaWidth,
      printDialog.PrintableAreaHeight);
    visual.Measure(pageSize);
    visual.Arrange(new Rect(0, 0, pageSize.Width, pageSize.Height));

    // Print the element.
    printDialog.PrintVisual(visual, "A Scaled Drawing");
}
```

Figure 29-5 shows the printed page that this code creates.

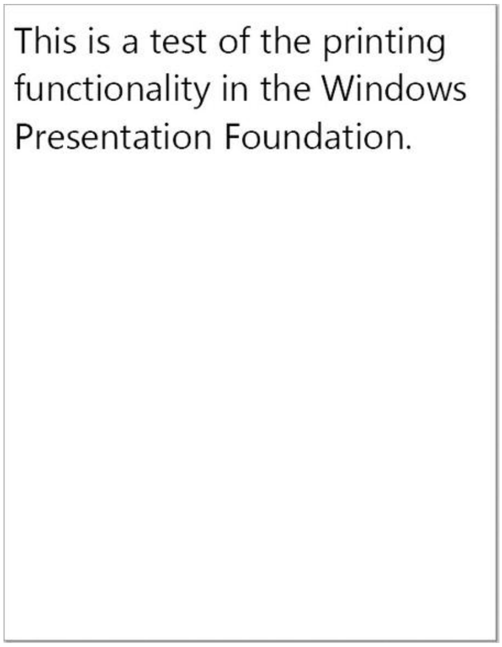

This is a test of the printing functionality in the Windows Presentation Foundation.

Figure 29-5. *Wrapped text using a TextBlock*

This approach allows you to grab the content you need out of a window but customize its printed appearance separately. However, it's of no help if you have content that needs to span more than one page (in which case you'll need the printing techniques described in the following sections).

Printing a Document

The PrintVisual() method may be the most versatile printing method, but the PrintDialog class also includes another option. You can use PrintDocument() to print the content from a flow document. The advantage of this approach is that a flow document can handle a huge amount of complex content and can split that content over multiple pages (just as it does onscreen).

You might expect that the PrintDialog.PrintDocument() method would require a FlowDocument object, but it actually takes a DocumentPaginator object. The DocumentPaginator is a specialized class whose sole role in life is to take content, split it into multiple pages, and supply each page when requested. Each page is represented by a DocumentPage object, which is really just a wrapper for a single Visual object with a little bit of sugar on top. You'll find just three more properties in the DocumentPage class. Size returns the size of the page, ContentBox is the size of the box where content is placed on the page after margins are added, and BleedBox is the area where print production-related bleeds, registration marks, and crop marks appear on the sheet, outside the page boundaries.

What this means is that PrintDocument() works in much the same way as PrintVisual(). The difference is that it prints several visuals—one for each page.

■ **Note** Although you could split your content into separate pages without using a DocumentPaginator and make repeated calls to PrintVisual(), this isn't a good approach. If you do, each page will become a separate print job.

So how do you get a DocumentPaginator object for a FlowDocument? The trick is to cast the FlowDocument to an IDocumentPaginatorSource and then use the DocumentPaginator property. Here's an example:

```
PrintDialog printDialog = new PrintDialog();
if (printDialog.ShowDialog() == true)
{
    printDialog.PrintDocument(
      ((IDocumentPaginatorSource)docReader.Document).DocumentPaginator,
      "A Flow Document");
}
```

This code may or may not produce the desired result, depending on the container that's currently housing your document. If your document is in-memory (but not in a window) or if it's stored in RichTextBox or FlowDocumentScrollViewer, this codes works fine. You'll end up with a multipaged printout with two columns (on a standard sheet of 8.5x11 paper in portrait orientation). This is the same result you'll get if you use the ApplicationCommands.Print command.

■ **Note** As you learned in Chapter 9, some controls include built-in command wiring. The FlowDocument containers (like the FlowDocumentScrollViewer used here) is one example. It handles the ApplicationCommands.Print command to perform a basic printout. This hardwired printing code is similar to the code shown previously, although it uses the XpsDocumentWriter, which is described in the "Printing Through XPS" section of this chapter.

However, if your document is stored in a FlowDocumentPageViewer or a FlowDocumentReader, the result isn't as good. In this case, your document is paginated the same way as the current view in the container. So if there are 24 pages required to fit the content into the current window, you'll get 24 pages in the printed output, each with a tiny window worth of data. Again, the solution is a bit messy, but it works. (It's also essentially the same solution that the ApplicationCommands.Print command takes.) The trick is to force the FlowDocument to paginate itself for the printer. You can do this by setting the FlowDocument.PageHeight and FlowDocument.PageWidth properties to the boundaries of the page, not the boundaries of the container. (In containers such as the FlowDocumentScrollViewer, these properties aren't set because pagination isn't used. That's why the printing feature works without a hitch—it paginates itself automatically when you create the printout.)

```
FlowDocument doc = docReader.Document;

doc.PageHeight = printDialog.PrintableAreaHeight;
doc.PageWidth = printDialog.PrintableAreaWidth;
printDialog.PrintDocument(
  ((IDocumentPaginatorSource)doc).DocumentPaginator,
   "A Flow Document");
```

You'll probably also want to set properties such as ColumnWidth and ColumnGap so you can get the number of columns you want. Otherwise, you'll get whatever is used in the current window.

The only problem with this approach is that once you've changed these properties, they apply to the container that displays your document. As a result, you'll end up with a compressed version of your document that's probably too small to read in the current window. A proper solution takes this into account by storing all these values, changing them, and then reapplying the original values.

Here's the complete code printing a two-column printout with a generous margin (added through the FlowDocument.PagePadding property):

```
PrintDialog printDialog = new PrintDialog();
if (printDialog.ShowDialog() == true)
{
    FlowDocument doc = docReader.Document;

    // Save all the existing settings.
    double pageHeight = doc.PageHeight;
    double pageWidth = doc.PageWidth;
    Thickness pagePadding = doc.PagePadding;
    double columnGap = doc.ColumnGap;
    double columnWidth = doc.ColumnWidth;

    // Make the FlowDocument page match the printed page.
    doc.PageHeight = printDialog.PrintableAreaHeight;
    doc.PageWidth = printDialog.PrintableAreaWidth;
    doc.PagePadding = new Thickness(50);

    // Use two columns.
    doc.ColumnGap = 25;
    doc.ColumnWidth = (doc.PageWidth - doc.ColumnGap
      - doc.PagePadding.Left - doc.PagePadding.Right) / 2;

    printDialog.PrintDocument(
      ((IDocumentPaginatorSource)doc).DocumentPaginator, "A Flow Document");

    // Reapply the old settings.
    doc.PageHeight = pageHeight;
    doc.PageWidth = pageWidth;
    doc.PagePadding = pagePadding;
    doc.ColumnGap = columnGap;
    doc.ColumnWidth = columnWidth;
}
```

This approach has a few limitations. Although you're able to tweak properties that adjust the margins and number of columns, you don't have much control. Of course, you can modify the FlowDocument programmatically (for example, temporarily increasing its FontSize), but you can't tailor the printout with details such as page numbers. You'll learn one way to get around this restriction in the next section.

PRINTING ANNOTATIONS

WPF includes two classes that derive from DocumentPaginator. FlowDocumentPaginator paginates flow documents—it's what you get when you examine the FlowDocument.DocumentPaginator property. Similarly, FixedDocumentPaginator paginates XPS documents, and it's used automatically by the XpsDocument class. However, both of these classes are marked internal and aren't accessible to your code. Instead, you can interact with these paginators by using the members of the base DocumentPaginator class.

WPF includes just one public, concrete paginator class, AnnotationDocumentPaginator, which is used to print a document with its associated annotations. (Chapter 28 discussed annotations.) AnnotationDocumentPaginator is public so that you can create it, if necessary, to trigger a printout of an annotated document.

To use the AnnotationDocumentPaginator, you must wrap an existing DocumentPaginator in a new AnnotationDocumentPaginator object. To do so, simply create an AnnotationDocumentPaginator, and pass in two references. The first reference is the original paginator for your document, and the second reference is the annotation store that contains all the annotations. Here's an example:

```
// Get the ordinary paginator.
DocumentPaginator oldPaginator =
  ((IDocumentPaginatorSource)doc).DocumentPaginator;

// Get the (currently running) annotation service for a
// specific document container.
AnnotationService service = AnnotationService.GetService(docViewer);

// Create the new paginator.
AnnotationDocumentPaginator newPaginator = new AnnotationDocumentPaginator(
  oldPaginator, service.Store);
```

Now, you can print the document with the superimposed annotations (in their current minimized or maximized state) by calling PrintDialog.PrintDocument() and passing in the AnnotationDocumentPaginator object.

Manipulating the Pages in a Document Printout

You can gain a bit more control over how a FlowDocument is printed by creating your own DocumentPaginator. As you might guess from its name, a DocumentPaginator divides the content of a document into distinct pages for printing (or displaying in a page-based FlowDocument viewer). The DocumentPaginator is responsible for returning the total number of pages based on a given page size and providing the laid-out content for each page as a DocumentPage object.

Your DocumentPaginator doesn't need to be complex—in fact, it can simply wrap the DocumentPaginator that's provided by the FlowDocument and allow it to do all the hard work of breaking up the text into individual pages. However, you can use your DocumentPaginator to make minor alterations, such as adding a header and a footer. The basic trick is to intercept every request the PrintDialog makes for a page and then alter that page before passing it along.

The first ingredient of this solution is building a HeaderedFlowDocumentPaginator class that derives from DocumentPaginator. Because DocumentPaginator is an abstract class, HeaderedFlowDocument needs to implement several methods. However, HeaderedFlowDocument can pass most of the work on to the standard DocumentPaginator that's provided by the FlowDocument.

Here's the basic skeleton of the HeaderedFlowDocumentPaginator class:

```
public class HeaderedFlowDocumentPaginator : DocumentPaginator
{
    // The real paginator (which does all the pagination work).
    private DocumentPaginator flowDocumentPaginator;

    // Store the FlowDocument paginator from the given document.
    public HeaderedFlowDocumentPaginator(FlowDocument document)
    {
        flowDocumentPaginator =
          ((IDocumentPaginatorSource)document).DocumentPaginator;
    }
    public override bool IsPageCountValid
    {
        get { return flowDocumentPaginator.IsPageCountValid;  }
    }

    public override int PageCount
    {
        get { return flowDocumentPaginator.PageCount; }
    }

    public override Size PageSize
    {
        get { return flowDocumentPaginator.PageSize; }
        set { flowDocumentPaginator.PageSize = value; }
    }

    public override IDocumentPaginatorSource Source
    {
        get { return flowDocumentPaginator.Source; }
    }

    public override DocumentPage GetPage(int pageNumber)
    { ... }
}
```

Because the HeaderedFlowDocumentPaginator hands off its work to its private DocumentPaginator, this code doesn't indicate how the PageSize, PageCount, and IsPageCountValid properties work. The PageSize is set by the DocumentPaginator consumer (the code that's using the DocumentPaginator). This property tells the DocumentPaginator how much space is available in each printed page (or onscreen). The PageCount and IsPageCountValid properties are provided *to* the DocumentPaginator consumer to indicate the pagination result. Whenever PageSize is changed, the DocumentPaginator will recalculate the size of each page. (Later in this chapter, you'll see a more complete DocumentPaginator that was created from scratch and includes the implementation details for these properties.)

The GetPage() method is where the action happens. This code calls the GetPage() method of the real DocumentPaginator and then gets to work on the page. The basic strategy is to pull the Visual object out of the page and place it in a new ContainerVisual object. You can then add the text you want to that ContainerVisual. Finally, you can create a new DocumentPage that wraps the ContainerVisual, with its newly inserted header.

■ **Note** This code uses visual-layer programming (Chapter 14). That's because you need a way to create visuals that represent your printed output. You don't need the full overhead of elements, which include event handling, dependency properties, and other plumbing. Custom print routines (as described in the next section) almost always use visual-layer programming and the ContainerVisual, DrawingVisual, and DrawingContext classes.

Here's the complete code:

```
public override DocumentPage GetPage(int pageNumber)
{
    // Get the requested page.
    DocumentPage page = flowDocumentPaginator.GetPage(pageNumber);

    // Wrap the page in a Visual object. You can then apply transformations
    // and add other elements.
    ContainerVisual newVisual = new ContainerVisual();
    newVisual.Children.Add(page.Visual);

    // Create a header.
    DrawingVisual header = new DrawingVisual();
    using (DrawingContext dc = header.RenderOpen())
    {
        Typeface typeface = new Typeface("Times New Roman");
        FormattedText text = new FormattedText("Page " +
          (pageNumber + 1).ToString(), CultureInfo.CurrentCulture,
          FlowDirection.LeftToRight, typeface, 14, Brushes.Black);

        // Leave a quarter inch of space between the page edge and this text.
        dc.DrawText(text, new Point(96*0.25, 96*0.25));
    }

    // Add the title to the visual.
    newVisual.Children.Add(header);

    // Wrap the visual in a new page.
    DocumentPage newPage = new DocumentPage(newVisual);
    return newPage;
}
```

This implementation assumes the page size doesn't change because of the addition of your header. Instead, the assumption is that there's enough empty space in the margin to accommodate the header. If you use this code with a small margin, the header will be printed overtop of your document content. This is the same way headers work in programs such as Microsoft Word. Headers aren't considered part of the main document, and they're positioned separately from the main document content.

There's one minor messy bit. You won't be able to add the Visual object for the page to the ContainerVisual while it's displayed in a window. The workaround is to temporarily remove it from the container, perform the printing, and then add it back.

```
FlowDocument document = docReader.Document;
docReader.Document = null;

HeaderedFlowDocumentPaginator paginator =
  new HeaderedFlowDocumentPaginator(document);
printDialog.PrintDocument(paginator, "A Headered Flow Document");

docReader.Document = document;
```

The HeaderedFlowDocumentPaginator is used for the printing, but it's not attached to the FlowDocument, so it won't change the way the document appears onscreen.

Custom Printing

By this point, you've probably realized the fundamental truth of WPF printing. You can use the quick-and-dirty techniques described in the previous section to send content from a window to your printer and even tweak it a bit. But if you want to build a first-rate printing feature for your application, you'll need to design it yourself.

Printing with the Visual Layer Classes

The best way to construct a custom printout is to use the visual-layer classes. Two classes are particularly useful:

- *ContainerVisual* is a stripped-down visual that can hold a collection of one or more other Visual objects (in its Children collection).
- *DrawingVisual* derives from ContainerVisual and adds a RenderOpen() method and a Drawing property. The RenderOpen() method creates a DrawingContext object that you can use to draw content in the visual (such as text, shapes, and so on), and the Drawing property lets you retrieve the final product as a DrawingGroup object.

Once you understand how to use these classes, the process for creating a custom printout is fairly straightforward.

1. Create your DrawingVisual. (You can also create a ContainerVisual in the less common case that you want to combine more than one separate drawn DrawingVisual object on the same page.)
2. Call DrawingVisual.RenderOpen() to get the DrawingContext object.
3. Use the methods of the DrawingContext to create your output.
4. Close the DrawingContext. (If you've wrapped the DrawingContext in a using block, this step is automatic.)
5. Using PrintDialog.PrintVisual() to send your visual to the printer.

Not only does this approach give you more flexibility than the print-an-element techniques you've used so far, it also has less overhead.

Obviously, the key to making this work is knowing what methods the DrawingContext class has for you to create your output. Table 29-1 describes the methods you can use. The Push*Xxx*() methods are particularly interesting, because they apply settings that will apply to future drawing operations. You can use Pop() to reverse the most recent Push*Xxx*() method. If you call more than one Push*Xxx*() method, you can switch them off one at a time with subsequent Pop() calls.

Table 29-1. *DrawingContext Methods*

Name	Description
DrawLine(), DrawRectangle(), DrawRoundedRectangle(), and DrawEllipse()	Draws the specified shape at the point you specify, with the fill and outline you specify. These methods mirror the shapes you saw in Chapter 12.
DrawGeometry () and DrawDrawing()	Draws more complex Geometry and Drawing objects. You saw these in Chapter 13.

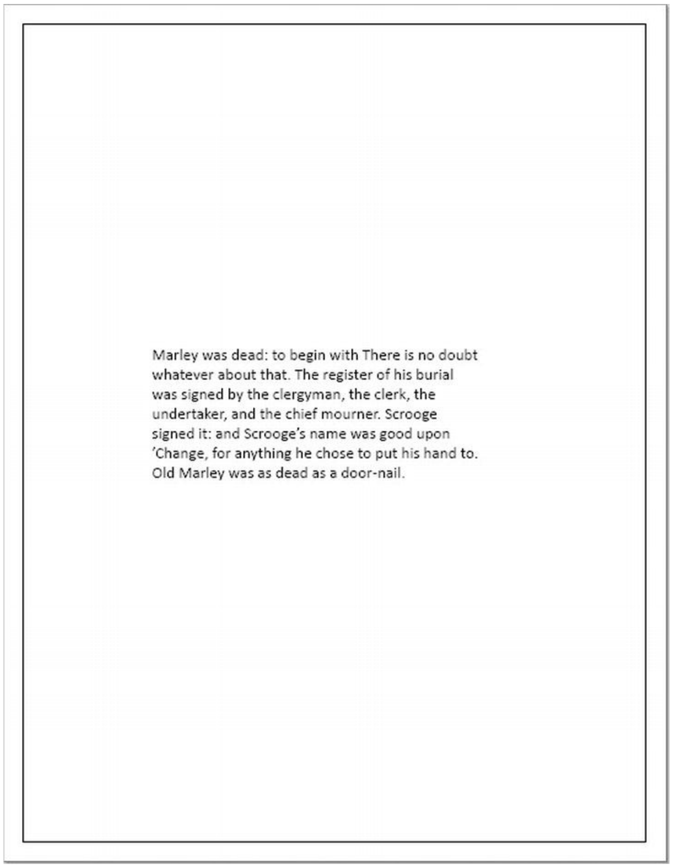
DrawText()	Draws text at the specified location. You specify the text, font, fill, and other details by passing a FormattedText object to this method. You can use DrawText() to draw wrapped text if you set the FormattedText.MaxTextWidth property.
DrawImage()	Draws a bitmap image in a specific region (as defined by a Rect).
Pop()	Reverses the last Push*Xxx*() method that was called. You use the Push*Xxx*() method to temporarily apply one or more effects and the Pop() method to reverse them.
PushClip()	Limits drawing to a specific clip region. Content that falls outside of this region isn't drawn.
PushEffect ()	Applies a BitmapEffect to subsequent drawing operations.
PushOpacity()	Applies a new opacity setting to make subsequent drawing operations partially transparent.
PushTransform()	Sets a Transform object that will be applied to subsequent drawing operations. You can use a transformation to scale, displace, rotate, or skew content.

These are all the ingredients that are required to create a respectable printout (along with a healthy dash of math to work out the optimum placement of all your content). The following code uses this approach to center a block of formatted text on a page and add a border around the page:

```
PrintDialog printDialog = new PrintDialog();
if (printDialog.ShowDialog() == true)
{
    // Create a visual for the page.
    DrawingVisual visual = new DrawingVisual();

    // Get the drawing context.
    using (DrawingContext dc = visual.RenderOpen())
    {
        // Define the text you want to print.
        FormattedText text = new FormattedText(txtContent.Text,
          CultureInfo.CurrentCulture, FlowDirection.LeftToRight,
          new Typeface("Calibri"), 20, Brushes.Black);

        // You must pick a maximum width to use text wrapping.
        text.MaxTextWidth = printDialog.PrintableAreaWidth / 2;

        // Get the size required for the text.
        Size textSize = new Size(text.Width, text.Height);

        // Find the top-left corner where you want to place the text.
        double margin = 96*0.25;
        Point point = new Point(
          (printDialog.PrintableAreaWidth - textSize.Width) / 2 - margin,
          (printDialog.PrintableAreaHeight - textSize.Height) / 2 - margin);

        // Draw the content.
        dc.DrawText(text, point);
        // Add a border (a rectangle with no background).
        dc.DrawRectangle(null, new Pen(Brushes.Black, 1),
          new Rect(margin, margin, printDialog.PrintableAreaWidth - margin * 2,
          printDialog.PrintableAreaHeight - margin * 2));
        }

    // Print the visual.
    printDialog.PrintVisual(visual, "A Custom-Printed Page");
}
```

■ **Tip** To improve this code, you'll probably want to move your drawing logic to a separate class (possibly the document class that wraps the content you're printing). You can then call a method in that class to get your visual and pass the visual to the PrintVisual() method in the event handling in your window code.

Figure 29-6 shows the output.



Marley was dead: to begin with There is no doubt
whatever about that. The register of his burial
was signed by the clergyman, the clerk, the
undertaker, and the chief mourner. Scrooge
signed it: and Scrooge's name was good upon
'Change, for anything he chose to put his hand to.
Old Marley was as dead as a door-nail.

Figure 29-6. *A custom printout*

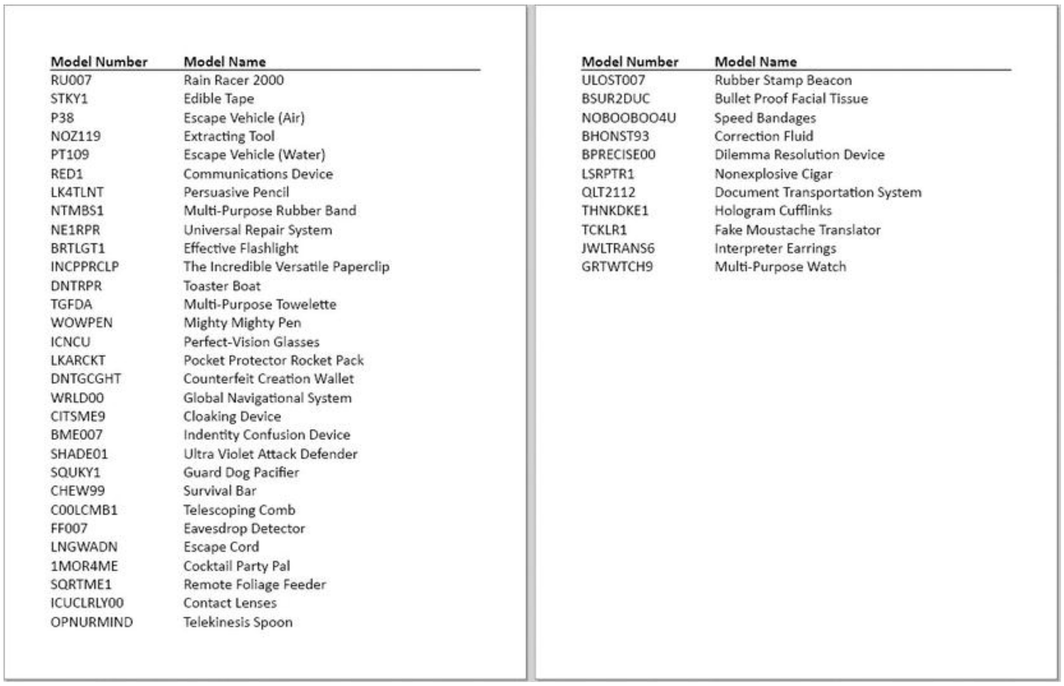
Custom Printing with Multiple Pages

A visual can't span pages. If you want a multipage printout, you need to use the same class you used when printing a FlowDocument: the DocumentPaginator. The difference is that you need to create the DocumentPaginator yourself from scratch. And this time you won't have a private DocumentPaginator on the inside to take care of all the heavy lifting.

Implementing the basic design of a DocumentPaginator is easy enough. You need to add a method that splits your content into pages, and you need to store the information about those pages internally. Then, you simply respond to the GetPage() method to provide the page that the PrintDialog needs. Each page is generated as a DrawingVisual, but the DrawingVisual is wrapped by the DocumentPage class.

The tricky part is separating your content into pages. There's no WPF magic here—it's up to you to decide how to divide your content. Some content is relatively easy to separate (like the long table you'll see in the next example), while some types of content are much more problematic. For example, if you want to print a long, text-based document, you'll need to move word by word through all your text, adding words to lines and lines to pages. You'll need to measure each separate piece of text to see whether it fits in the line. And that's just to split text content using ordinary left justification—if you want something comparable to the best-fit justification used for the FlowDocument, you're better off using the PrintDialog. PrintDocument() method, as described earlier, because there's a huge amount of code to write and some very specialized algorithms to use.

The following example demonstrates a typical not-too-difficult pagination job. The contents of a DataTable are printed in a tabular structure, putting each record on a separate row. The rows are split into pages based on how many lines fit on a page using the chosen font. Figure 29-7 shows the final result.



Model Number	Model Name
RU007	Rain Racer 2000
STKY1	Edible Tape
P38	Escape Vehicle (Air)
NOZ119	Extracting Tool
PT109	Escape Vehicle (Water)
RED1	Communications Device
LK4TLNT	Persuasive Pencil
NTMBS1	Multi-Purpose Rubber Band
NE1RPR	Universal Repair System
BRTLGT1	Effective Flashlight
INCPPRCLP	The Incredible Versatile Paperclip
DNTRPR	Toaster Boat
TGFDA	Multi-Purpose Towelette
WOWPEN	Mighty Mighty Pen
ICNCU	Perfect-Vision Glasses
LKARCKT	Pocket Protector Rocket Pack
DNTGCGHT	Counterfeit Creation Wallet
WRLD00	Global Navigational System
CITSME9	Cloaking Device
BME007	Indentity Confusion Device
SHADE01	Ultra Violet Attack Defender
SQUKY1	Guard Dog Pacifier
CHEW99	Survival Bar
C00LCMB1	Telescoping Comb
FF007	Eavesdrop Detector
LNGWADN	Escape Cord
1MOR4ME	Cocktail Party Pal
SQRTME1	Remote Foliage Feeder
ICUCLRLY00	Contact Lenses
OPNURMIND	Telekinesis Spoon

Model Number	Model Name
ULOST007	Rubber Stamp Beacon
BSUR2DUC	Bullet Proof Facial Tissue
NOBOOBOO4U	Speed Bandages
BHONST93	Correction Fluid
BPRECISE00	Dilemma Resolution Device
LSRPTR1	Nonexplosive Cigar
QLT2112	Document Transportation System
THNKDKE1	Hologram Cufflinks
TCKLR1	Fake Moustache Translator
JWLTRANS6	Interpreter Earrings
GRTWTCH9	Multi-Purpose Watch

Figure 29-7. A table of data split over two pages

In this example, the custom DocumentPaginator contains the code for splitting the data into pages and the code for printing each page to a Visual object. Although you could factor this into two classes (for example, if you want to allow the same data to be printed in the same way but paginated differently), usually you won't because the code required to calculate the page size is tightly bound to the code that actually prints the page.

The custom DocumentPaginator implementation is fairly long, so I'll break it down piece by piece. First, the StoreDataSetPaginator stores a few important details in private variables, including the DataTable that you plan to print and the chosen typeface, font size, page size, and margin:

```
public class StoreDataSetPaginator : DocumentPaginator
{
    private DataTable dt;

    private Typeface typeface;
    private double fontSize;
    private double margin;

    private Size pageSize;
    public override Size PageSize
    {
        get { return pageSize; }
        set
        {
            pageSize = value;
            PaginateData();
        }
    }

    public StoreDataSetPaginator(DataTable dt, Typeface typeface,
      double fontSize, double margin, Size pageSize)
    {
        this.dt = dt;
        this.typeface = typeface;
        this.fontSize = fontSize;
        this.margin = margin;
        this.pageSize = pageSize;
        PaginateData();
    }
    ...
```

Notice that these details are supplied in the constructor and then can't be changed. The only exception is the PageSize property, which is a required abstract property from the DocumentPaginator class. You could create properties to wrap the other details if you wanted to allow your code to alter these details after creating the paginator. You'd simply need to make sure you call PaginateData() when any of these details are changed.

The PaginateData() isn't a required member. It's just a handy place to calculate how many pages are needed. The StoreDataSetPaginator paginates its data as soon as the DataTable is supplied in the constructor.

When the PaginateData() method runs, it measures the amount of space required for a line of text and compares that against the size of the page to find out how many lines will fit on each page. The result is stored in a field named rowsPerPage.

```
...
private int rowsPerPage;
private int pageCount;

private void PaginateData()
{
    // Create a test string for the purposes of measurement.
    FormattedText text = GetFormattedText("A");

    // Count the lines that fit on a page.
    rowsPerPage = (int)((pageSize.Height-margin*2) / text.Height);

    // Leave a row for the headings
    rowsPerPage -= 1;

    pageCount = (int)Math.Ceiling((double)dt.Rows.Count / rowsPerPage);
}
...
```

This code assumes that a capital letter *A* is sufficient for calculating the line height. However, this might not be true for all fonts, in which case you'd need to pass a string that includes a complete list of all characters, numbers, and punctuation to GetFormattedText().

■ **Note** To calculate the number of lines that fit on a page, you use the FormattedText.Height property. You *don't* use FormattedText.LineHeight, which is 0 by default. The LineHeight property is provided for you to override the default line spacing when drawing a block with multiple lines of text. However, if you don't set it, the FormattedText class uses its own calculation, which uses the Height property.

In some cases, you'll need to do a bit more work and store a custom object for each page (for example, an array of strings with the text for each line). However, this isn't required in the StoreDataSetPaginator example because all the lines are the same, and there isn't any text wrapping to worry about.

The PaginateData() uses a private helper method named GetFormattedText(). When printing text, you'll find that you need to construct a great number of FormattedText objects. These FormattedText objects will always share the same culture and left-to-right text flow options. In many cases, they'll also use the same typeface. The GetFormattedText() encapsulates these details and so simplifies the rest of your code. The StoreDataSetPaginator uses two overloaded versions of GetFormattedText(), one of which accepts a different typeface to use:

```
...
private FormattedText GetFormattedText(string text)
{
    return GetFormattedText(text, typeface);
}
private FormattedText GetFormattedText(string text, Typeface typeface)
{
    return new FormattedText(
      text, CultureInfo.CurrentCulture, FlowDirection.LeftToRight,
      typeface, fontSize, Brushes.Black);
}
```

```
...
```

Now that you have the number of pages, you can implement the remainder of the required DocumentPaginator properties:

```
...
// Always returns true, because the page count is updated immediately,
// and synchronously, when the page size changes.
// It's never left in an indeterminate state.
public override bool IsPageCountValid
{
    get { return true; }
}

public override int PageCount
{
    get { return pageCount; }
}

public override IDocumentPaginatorSource Source
{
    get { return null; }
}
...
```

There's no factory class that can create this custom DocumentPaginator, so the Source property returns null.

The last implementation detail is also the longest. The GetPage() method returns a DocumentPage object for the requested page, with all the data.

The first step is to find the position where the two columns will begin. This example sizes the columns relative to the width of one capital letter *A*, which is a handy shortcut when you don't want to perform more detailed calculations.

```
...
public override DocumentPage GetPage(int pageNumber)
{
    // Create a test string for the purposes of measurement.
    FormattedText text = GetFormattedText("A");

    double col1_X = margin;
    double col2_X = col1_X + text.Width * 15;
    ...
```

The next step is to find the offsets that identify the range of records that belong on this page:

```
    ...
    // Calculate the range of rows that fits on this page.
    int minRow = pageNumber * rowsPerPage;
    int maxRow = minRow + rowsPerPage;
    ...
```

Now the print operation can begin. There are three elements to print: column headers, a separating line, and the rows. The underlined header is drawn using DrawText() and DrawLine() methods from the DrawingContext class. For the rows, the code loops from the first row to the last row, drawing the text from the corresponding DataRow in the two columns and then increasing the Y-coordinate position by an amount equal to the line height of the text.

```
        ...
        // Create the visual for the page.
        DrawingVisual visual = new DrawingVisual();

        // Set the position to the top-left corner of the printable area.
        Point point = new Point(margin, margin);

        using (DrawingContext dc = visual.RenderOpen())
        {
            // Draw the column headers.
            Typeface columnHeaderTypeface = new Typeface(
              typeface.FontFamily, FontStyles.Normal, FontWeights.Bold,
              FontStretches.Normal);
            point.X = col1_X;
            text = GetFormattedText("Model Number", columnHeaderTypeface);
            dc.DrawText(text, point);
            text = GetFormattedText("Model Name", columnHeaderTypeface);
            point.X = col2_X;
            dc.DrawText(text, point);

            // Draw the line underneath.
            dc.DrawLine(new Pen(Brushes.Black, 2),
              new Point(margin, margin + text.Height),
              new Point(pageSize.Width - margin, margin + text.Height));

            point.Y += text.Height;

            // Draw the column values.
            for (int i = minRow; i < maxRow; i++)
            {
                // Check for the end of the last (half-filled) page.
                if (i > (dt.Rows.Count - 1)) break;

                point.X = col1_X;
                text = GetFormattedText(dt.Rows[i]["ModelNumber"].ToString());
                dc.DrawText(text, point);

                // Add second column.
                text = GetFormattedText(dt.Rows[i]["ModelName"].ToString());
                point.X = col2_X;
                dc.DrawText(text, point);
                point.Y += text.Height;
        }
    }
    return new DocumentPage(visual, pageSize, new Rect(pageSize),
      new Rect(pageSize));
}
```

Now that the StoreDateSetDocumentPaginator is complete, you can use it whenever you want to print the contents of the DataTable with the product list, as shown here:

```
PrintDialog printDialog = new PrintDialog();
if (printDialog.ShowDialog() == true)
{
    StoreDataSetPaginator paginator = new StoreDataSetPaginator(ds.Tables[0],
      new Typeface("Calibri"), 24, 96*0.75,
      new Size(printDialog.PrintableAreaWidth, printDialog.PrintableAreaHeight));

    printDialog.PrintDocument(paginator, "Custom-Printed Pages");
}
```

The StoreDataSetPaginator has a certain amount of flexibility built in—for example, it can work with different fonts, margins, and paper sizes—but it can't deal with data that has a different schema. Clearly, there's still room in the WPF library for a handy class that could accept data, column and row definitions, headers and footers, and so on, and then print a properly paginated table. WPF doesn't have anything like this currently, but you can expect third-party vendors to provide components that fill the gaps.

Print Settings and Management

So far, you've focused all your attention on two methods of the PrintDialog class: PrintVisual() and PrintDocument(). This is all you need to use to get a decent printout, but you have more to do if you want to manage printer settings and jobs. Once again, the PrintDialog class is your starting point.

Maintaining Print Settings

In the previous examples, you saw how the PrintDialog class allows you to choose a printer and its settings. However, if you've used these examples to make more than one printout, you may have noticed a slight anomaly. Each time you return to the Print dialog box, it reverts to the default print settings. You need to pick the printer you want and adjust it all over again.

Life doesn't need to be this difficult. You have the ability to store this information and reuse it. One good approach is to store the PrintDialog as a member variable in your window. That way, you don't need to create the PrintDialog before each new print operation—you just keep using the existing object. This works because the PrintDialog encapsulates the printer selection and printer settings through two properties: PrintQueue and PrintTicket.

The PrintQueue property refers to a System.Printing.PrintQueue object, which represents the print queue for the selected printer. And as you'll discover in the next section, the PrintQueue also encapsulates a good deal of features for managing your printer and its jobs.

The PrintTicket property refers to a System.Printing.PrintTicket object, which defines the settings for a print job. It includes details such as print resolution and duplexing. If you want, you're free to tweak the settings of a PrintTicket programmatically. The PrintTicket class even has a GetXmlStream() method and a SaveTo() method, both of which let you serialize the ticket to a stream, and a constructor that lets you re-create a PrintTicket object based on the stream. This is an interesting option if you want to persist specific print settings between application sessions. (For example, you could use this ability to create a "print profile" feature.)

As long as these PrintQueue and PrintTicket properties remain consistent, the selected printer and its properties will remain the same each time you show the Print dialog box. So even if you need to create the PrintDialog box multiple times, you can simply set these properties to keep the user's selections.

Printing Page Ranges

You haven't yet considered one of the features in the PrintDialog class. You can allow the user to choose to print only a subset of a larger printout using the Pages text box in the Page Range box. The Pages text box lets the user specify a group of pages by entering the starting and ending page (for example, *4–6*) or pick a specific page (for example, *4*). It doesn't allow multiple page ranges (such as *1–3,5*).

The Pages text box is disabled by default. To switch it on, you simply need to set the PrintDialog.UserPageRangeEnabled property to true before you call ShowDialog(). The Selection and Current Page options remain disabled, because they aren't supported by the PrintDialog class. You can also set the MaxPage and MinPage properties to constrain the pages that the user can pick.

After you've shown the Print dialog box, you can determine whether the user entered a page range by checking the PageRangeSelection property. If it provides a value of UserPages, there's a page range present. The PageRange property provides a PageRange property that indicates the starting page (PageRange.PageFrom) and ending page (PageRange.PageTo). It's up to your printing code to take these values into account and print only the requested pages.

Managing a Print Queue

Typically, a client application has a limited amount of interaction with the print queue. After a job is dispatched, you may want to display its status or (rarely) provide the option to pause, resume, or cancel the job. The WPF print classes go far beyond this level and allow you to build tools that can manage local or remote print queues.

The classes in the System.Printing namespace provide the support for managing print queues. You can use a few key classes to do most of the work, and they're outlined in Table 29-2.

Table 29-2. *Key Classes for Print Management*

Name	Description
PrintServer and LocalPrintServer	Represents a computer that provides printers or another device that does. (This "other device" might include a printer with built-in networking or a dedicated piece of network hardware that acts as a print server.) Using the PrintServer class, you can get a collection of PrintQueue objects for that computer. You can also use the LocalPrintServer class, which derives from PrintServer and always represents the current computer. It adds a DefaultPrintQueue property that you can use to get (or set) the default printer and a static GetDefaultPrintQueue() method that you can use without creating a LocalPrintServer instance.
PrintQueue	Represents a configured printer on a print server. The PrintQueue class allows you to get information about that printer's status and manage the print queue. You can also get a collection of PrintQueueJobInfo objects for that printer.
PrintSystemJobInfo	Represents a job that's been submitted to a print queue. You can get information about its status and modify its state or delete it.

Using these basic ingredients, you can create a program that launches a printout without any user intervention.

```
PrintDialog dialog = new PrintDialog();

// Pick the default printer.
dialog.PrintQueue = LocalPrintServer.GetDefaultPrintQueue();
```

```
// Print something.
dialog.PrintDocument(someContent, "Automatic Printout");
```

You can also create and apply a PrintTicket object to the PrintDialog to configure other print-related settings. More interestingly, you can delve deeper in the PrintServer, PrintQueue, and PrintSystemJobInfo classes to study what's taking place.

Figure 29-8 shows a simple program that allows you to browse the print queues on the current computer and see the outstanding jobs for each one. This program also allows you to perform some basic printer management tasks, such as suspending a printer (or a print job), resuming the printer (or print job), and canceling one job or all the jobs in a queue. By considering how this application works, you can learn the basics of the WPF print management model.

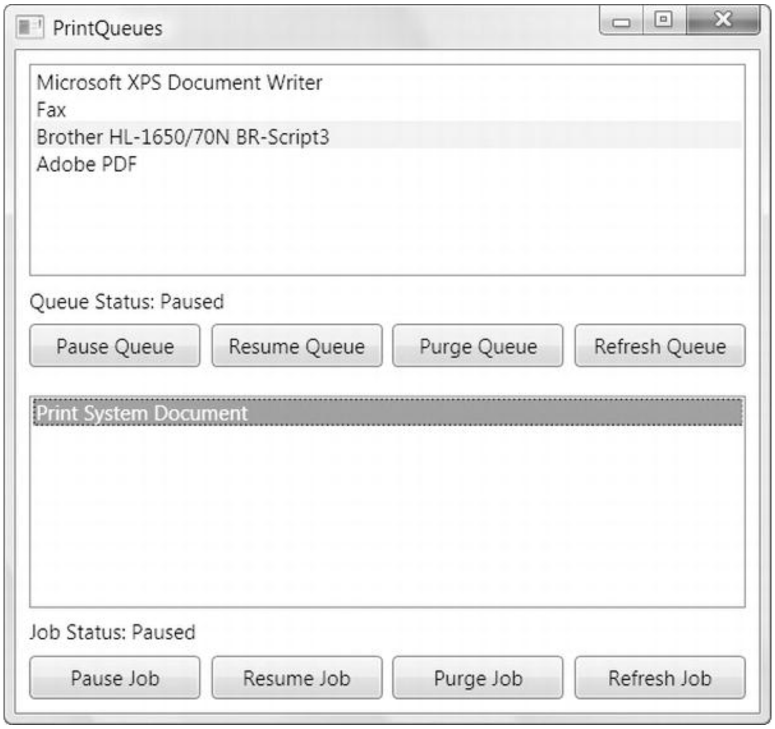


Figure 29-8. *Browsing printer queues and jobs*

This example uses a single PrintServer object, which is created as a member field in the window class:

```
private PrintServer printServer = new PrintServer();
```

When you create a PrintServer object without passing any arguments to the constructor, the PrintServer represents the current computer. Alternatively, you could pass the UNC path that points to a print server on the network, like this:

```
private PrintServer printServer = new PrintServer(@"\\Warehouse\PrintServer");
```

Using the PrintServer object, the code grabs a list of print queues that represent the printers that are configured on the current computer. This step is easy—all you need to do is call the PrintServer.GetPrintQueues() method when the window is first loaded:

```
private void Window_Loaded(object sender, EventArgs e)
{
    lstQueues.DisplayMemberPath = "FullName";
    lstQueues.SelectedValuePath = "FullName";
    lstQueues.ItemsSource = printServer.GetPrintQueues();
}
```

The only piece of information this code snippet uses is the PrintQueue.FullName property. However, the PrintQueue class is stuffed with properties you can examine. You can get the default print settings (using properties such as DefaultPriority, DefaultPrintTicket, and so on), you can get the status and general information (using properties such as QueueStatus and NumberOfJobs), and you can isolate specific problems using Boolean Is*Xxx* and Has*Xxx* properties (such as IsManualFeedRequired, IsWarmingUp, IsPaperJammed, IsOutOfPaper, HasPaperProblem, and NeedUserIntervention).

The current example reacts when a printer is selected in the list by displaying the status for that printer and then fetching all the jobs in the queue. The PrintQueue.GetPrintJobInfoCollection() performs this task.

```
private void lstQueues_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    try
    {
        PrintQueue queue =
          printServer.GetPrintQueue(lstQueues.SelectedValue.ToString());
        lblQueueStatus.Text = "Queue Status: " + queue.QueueStatus.ToString();
        lstJobs.DisplayMemberPath = "JobName";
        lstJobs.SelectedValuePath = "JobIdentifier";

        lstJobs.ItemsSource = queue.GetPrintJobInfoCollection();
    }
    catch (Exception err)
    {
        MessageBox.Show(err.Message,
          "Error on " + lstQueues.SelectedValue.ToString());
    }
}
```

Each job is represented as a PrintSystemJobInfo object. When a job is selected in the list, this code shows its status:

```
private void lstJobs_SelectionChanged(object sender, SelectionChangedEventArgs e)
{
    if (lstJobs.SelectedValue == null)
    {
        lblJobStatus.Text = "";
    }
    else
    {
        PrintQueue queue =
          printServer.GetPrintQueue(lstQueues.SelectedValue.ToString());
        PrintSystemJobInfo job = queue.GetJob((int)lstJobs.SelectedValue);

        lblJobStatus.Text = "Job Status: " + job.JobStatus.ToString();
    }
}
```

The only remaining detail is the event handlers that manipulate the queue or job when you click one of the buttons in the window. This code is extremely straightforward. All you need to do is get a reference to the appropriate queue or job and then call the corresponding method. For example, here's how to pause a PrintQueue:

```
PrintQueue queue = printServer.GetPrintQueue(lstQueues.SelectedValue.ToString());
queue.Pause();
```

And here's how to pause a print job:

```
PrintQueue queue = printServer.GetPrintQueue(lstQueues.SelectedValue.ToString());
PrintSystemJobInfo job = queue.GetJob((int)lstJobs.SelectedValue);
job.Pause();
```

■ **Note** It's possible to pause (and resume) an entire printer or a single job. You can do both tasks using the Printers icon in the Control Panel. Right-click a printer to pause or resume a queue, or double-click a printer to see its jobs, which you can manipulate individually.

Obviously, you'll need to add error handling when you perform this sort of task, because it won't necessarily succeed. For example, Windows security might stop you from attempting to cancel someone else's print job or an error might occur if you try to print to a networked printer after you've lost your connection to the network.

WPF includes quite a bit of print-related functionality. If you're interested in using this specialized functionality (perhaps because you're building some sort of tool or creating a long-running background task), check out the classes in the System.Printing namespace in the MSDN help.

Printing Through XPS

As you learned in Chapter 28, WPF supports two complementary types of documents. Flow documents handle flexible content that flows to fit any page size you specify. XPS documents store print-ready content that's based on a fixed-page size. The content is frozen in place and preserved in its precise, original form.

As you'd expect, printing an XpsDocument is easy. The XpsDocument class exposes a DocumentPaginator, just like the FlowDocument. However, the DocumentPaginator of an XpsDocument has little to do, because the content is already laid out in fixed, unchanging pages.

Here's the code you might use to load an XPS file into memory, show it in a DocumentViewer, and then send it to the printer:

```
// Display the document.
XpsDocument doc = new XpsDocument("filename.xps", FileAccess.ReadWrite);
docViewer.Document = doc.GetFixedDocumentSequence();
doc.Close();

// Print the document.
if (printDialog.ShowDialog() == true)
```

```
{
    printDialog.PrintDocument(docViewer.Document.DocumentPaginator,
      "A Fixed Document");
}
```

Obviously, you don't need to show a fixed document in a DocumentViewer before you print it. This code includes that step because it's the most common option. In many scenarios, you'll load up the XpsDocument for review and print it after the user clicks a button.

As with the viewers for FlowDocument objects, the DocumentViewer also handles the ApplicationCommands.Print command, which means you can send an XPS document from the DocumentViewer to the printer with no code required.

Creating an XPS Document for a Print Preview

WPF also includes all the support you need to programmatically create XPS documents. Creating an XPS document is conceptually similar to printing some content—once you've built your XPS document, you've chosen a fixed page size and frozen your layout. So why bother taking this extra step? There are two good reasons:

- *Print preview.* You can use your generated XPS document as a print preview by displaying it in a DocumentViewer. The user can then choose whether to go ahead with the printout.
- *Asynchronous printing.* The XpsDocumentWriter class includes both a Write() method for synchronous printing and a WriteAsync() method that lets you send content to the printer asynchronously. For a long, complex print operation, the asynchronous option is preferred. It allows you to create a more responsive application.

The basic technique for creating an XPS document is create an XpsDocumentWriter object using the static XpsDocument.CreateXpsDocumentWriter() method. Here's an example:

```
XpsDocument xpsDocument = new XpsDocument("filename.xps", FileAccess.ReadWrite);
XpsDocumentWriter writer = XpsDocument.CreateXpsDocumentWriter(xpsDocument);
```

The XpsDocumentWriter is a stripped-down class—its functionality revolves around the Write() and WriteAsync() methods that write content to your XPS document. Both of these methods are overloaded multiple times, allowing you to write different types of content, including another XPS document, a page that you've extracted from an XPS document, a visual (which allows you to write any element), and a DocumentPaginator. The last two options are the most interesting, because they duplicate the options you have with printing. For example, if you've created a DocumentPaginator to enable custom printing (as described earlier in this chapter), you can also use it to write an XPS document.

Here's an example that opens an existing flow document and then writes it to a temporary XPS document using the XpsDocumentWriter.Write() method. The newly created XPS document is then displayed in a DocumentViewer, which acts as a print preview.

```
using (FileStream fs = File.Open("FlowDocument1.xaml", FileMode.Open))
{
    FlowDocument flowDocument = (FlowDocument)XamlReader.Load(fs);
    writer.Write(((IDocumentPaginatorSource)flowDocument).DocumentPaginator);

    // Display the new XPS document in a viewer.
    docViewer.Document = xpsDocument.GetFixedDocumentSequence();
    xpsDocument.Close();
}
```

You can get a visual or paginator in a WPF application in an endless variety of ways. Because the XpsDocumentWriter supports these classes, it allows you to write any WPF content to an XPS document.

Writing to an In-Memory XPS Document

The XpsDocument class assumes that you want to write your XPS content to a file. This is a bit awkward for situations like the one shown previously, where the XPS document is a temporary stepping stone that's used to create a preview. Similar problems occur if you want to serialize XPS content to some other storage location, like a field in a database record.

It's possible to get around this limitation, and write XPS content directly to a MemoryStream. However, it takes a bit more work, as you first need to create a package for your XPS content. Here's the code that does the trick:

```
// Get ready to store the content in memory.
MemoryStream ms = new MemoryStream();

// Create a package usign the static Package.Open() method.
Package package = Package.Open(ms, FileMode.Create, FileAccess.ReadWrite);

// Every package needs a URI. Use the pack:// syntax.
// The actual file name is unimportant.
Uri documentUri = new Uri("pack://InMemoryDocument.xps");

// Add the package.
PackageStore.AddPackage(documentUri, package);

// Create the XPS document based on this package. At the same time, choose
// the level of compression you want for the in-memory content.
XpsDocument xpsDocument = new XpsDocument(package, CompressionOption.Fast,
  DocumentUri.AbsoluteUri);
```

When you're finished using the XPS document, you can close the stream to recover the memory.

■ **Note** Don't use the in-memory approach if you might have a larger XPS document (for example, if you're generating an XPS document based on content in a database, and you don't know how many records there will be). Instead, use a method like Path.GetTempFileName() to get a suitable temporary path for a file-based XPS document.

Printing Directly to the Printer via XPS

As you've learned in this chapter, the printing support in WPF is built on the XPS print path. If you use the PrintDialog class, you might not see any sign of this low-level reality. If you use the XpsDocumentWriter, it's impossible to miss.

So far, you've been funneling all your printing through the PrintDialog class. This isn't necessary—in fact, the PrintDialog delegates the real work to the XpsDocumentWriter. The trick is to create an

XpsDocumentWriter that wraps a PrintQueue rather than a FileStream. The actual code for writing the printed output is identical—you simply rely on the Write() and WriteAsync() methods.

Here's a snippet of code that shows the Print dialog box, gets the selected printer, and uses it to create an XpsDocumentWriter that submits the print job:

```
string filePath = Path.Combine(appPath, "FlowDocument1.xaml");

if (printDialog.ShowDialog() == true)
{
    PrintQueue queue = printDialog.PrintQueue;
    XpsDocumentWriter writer = PrintQueue.CreateXpsDocumentWriter(queue);

    using (FileStream fs = File.Open(filePath, FileMode.Open))
    {
        FlowDocument flowDocument = (FlowDocument)XamlReader.Load(fs);
        writer.Write(((IDocumentPaginatorSource)flowDocument).DocumentPaginator);
    }
}
```

Interestingly, this example still uses the PrintDialog class. However, it simply uses it to display the standard Print dialog box and allow the user to choose a printer. The actual printing is performed through the XpsDocumentWriter.

Asynchronous Printing

The XpsDocumentWriter makes asynchronous printing easy. In fact, you can convert the previous example to use asynchronous printing by simply replacing the call to the Write() method with a call to WriteAsync().

■ **Note** In Windows, all print jobs are printed asynchronously. However, the process of *submitting* the print job takes place synchronously if you use Write() and asynchronously if you use WriteAsync(). In many cases, the time taken to submit a print job won't be significant, and you won't need this feature. Another consideration is that if you want to build (and paginate) the content you want to print asynchronously, this is often the most time-consuming stage of printing, and if you want this ability, you'll need to write the code that runs your printing logic on a background thread. You can use the techniques described in Chapter 31 (such as the BackgroundWorker) to make this process relatively easy.

The signature of the WriteAsync() method matches the signature of the Write() method—in other words, WriteAsync() accepts a paginator, visual, or one of a few other types of objects. Additionally, the WriteAsync() method includes overloads that accept an optional second parameter with state information. This state information can be any object you want to use to identify the print job. This object is provided through the WritingCompletedEventArgs object when the WritingCompleted event fires. This allows you to fire off multiple print jobs at once, handle the WritingCompleted event for each one with the same event handler, and determine which one has been submitted each time the event fires.

When an asynchronous print job is underway, you can cancel it by calling the CancelAsync() method. The XpsDocumentWriter also includes a small set of events that allow you to react as a print job is submitted, including WritingProgressChanged, WritingCompleted, and WritingCancelled. Keep in mind that the WritingCompleted event fires when the print job has been written to the print queue, but this doesn't mean the printer has printed it yet.

The Last Word

In this chapter, you learned about WPF's printing model. First you considered the easiest entry point: the all-in-one PrintDialog class that allows users to configure print settings and allows your application to send a document or visual to the printer. After considering a variety of ways to extend the PrintDialog and use it with onscreen and dynamically generated content, you looked at the lower-level XPS printing model. You then learned about the XpsDocumentWriter, which supports the PrintDialog and can be used independently. The XpsDocumentWriter gives you an easy way to create a print preview (because WPF doesn't include any print preview control), and it allows you to submit your print job asynchronously.