

In the previous example, the annotation is created without any author information. If you plan to have multiple users annotating the same document, you'll almost certainly want to store some identifying information. Just pass a string that identifies the author as a parameter to the command, as shown here:

```
<Button Command="annot:AnnotationService.CreateTextStickyNoteCommand"
  CommandParameter="{StaticResource AuthorName}">
  Text Note
</Button>
```

This markup assumes the author name is set as a resource:

```
<sys:String x:Key="AuthorName">[Anonymous]</sys:String>
```

This allows you to set the author name when the window first loads, at the same time as you initialize the annotation service. You can use a name that the user supplies, which you'll probably want to store in a user-specific .config file as an application setting. Alternatively, you can use the following code to grab the current user's Windows user account name with the help of the `System.Security.Principal.WindowsIdentity` class:

```
WindowsIdentity identity = WindowsIdentity.GetCurrent();
this.Resources["AuthorName"] = identity.Name;
```

To create the window shown in Figure 28-17, you'll also want to create buttons that use the `CreateInkStickyNoteCommand` (to create a note window that accepts hand-drawn ink content) and `DeleteStickyNotesCommand` (to remove previously created sticky notes):

```
<Button Command="annot:AnnotationService.CreateInkStickyNoteCommand"
  CommandParameter="{StaticResource AuthorName}">
  Ink Note
</Button>
<Button Command="annot:AnnotationService.DeleteStickyNotesCommand">
  Delete Note(s)
</Button>
```

The `DeleteStickyNotesCommand` removes all the sticky notes in the currently selected text. Even if you don't provide this command, the user can still remove annotations using the Edit menu in the note window (unless you've given the note window a different control template that doesn't include this feature).

The final detail is to create the buttons that allow you to apply highlighting. To add a highlight, you use the `CreateHighlightCommand` and you pass the `Brush` object that you want to use as the `CommandParameter`. However, it's important to make sure you use a brush that has a partially transparent color. Otherwise, your highlighted content will be completely obscured, as shown in Figure 28-19.

For example, if you want to use the solid color #FF32CD32 (for lime green) to highlight your text, you should reduce the alpha value, which is stored as a hexadecimal number in the first two characters. (The alpha value ranges from 0 to 255, where 0 is fully transparent and 255 is fully opaque.) For example, the color #54FF32CD32 gives you a semitransparent version of the lime green color, with an alpha value of 84 (or 54 in hexadecimal notation).

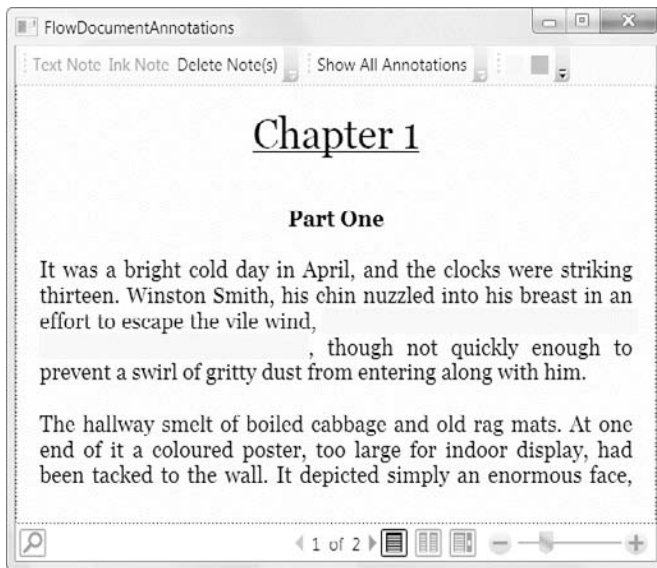


Figure 28-19. Highlighting content with a nontransparent color

The following markup defines two highlighting buttons, one for applying yellow highlights and one for green highlights. The button itself doesn't include any text. It simply shows a 15-by-15 square of the appropriate color. The `CommandParameter` defines a `SolidColorBrush` that uses the same color but with reduced opacity so the text is still visible:

```
<Button Background="Yellow" Width="15" Height="15" Margin="2,0"
  Command="annot:AnnotationService.CreateHighlightCommand">
  <Button.CommandParameter>
    <SolidColorBrush Color="#54FFFF00"></SolidColorBrush>
  </Button.CommandParameter>
</Button>

<Button Background="LimeGreen" Width="15" Height="15" Margin="2,0"
  Command="annot:AnnotationService.CreateHighlightCommand">
  <Button.CommandParameter>
    <SolidColorBrush Color="#5432CD32"></SolidColorBrush>
  </Button.CommandParameter>
</Button>
```

You can add a final button to remove highlighting in the selected region:

```
<Button Command="annot:AnnotationService.ClearHighlightsCommand">
  Clear Highlights
</Button>
```

■ **Note** When you print a document that includes annotations using the `ApplicationCommands.Print` command, the annotations are printed just as they appear. In other words, minimized annotations will appear minimized, visible annotations will appear overtop of content (and may obscure other parts of the document), and so on. If you want to create a printout that doesn't include annotations, simply disable the annotation service before you begin your printout.

Examining Annotations

At some point, you may want to examine all the annotations that are attached to a document. There are many possible reasons—you may want to display a summary report about your annotations, print an annotation list, export annotation text to a file, and so on.

The `AnnotationStore` makes it relatively easy to get a list of all the annotations it contains using the `GetAnnotations()` method. You can then examine each annotation as an `Annotation` object:

```

IList<Annotation> annotations = service.Store.GetAnnotations();
foreach (Annotation annotation in annotations)
{
    ...
}

```

In theory, you can find annotations in a specific portion of a document using the overloaded version of the `GetAnnotations()` method that takes a `ContentLocator` object. In practice, however, this is tricky, because the `ContentLocator` object is difficult to use correctly and you need to match the starting position of the annotation precisely.

Once you've retrieved an `Annotation` object, you'll find that it provides the properties listed in Table 28-8.

Table 28-8. Annotation Properties

Name	Description
Id	A global identifier (GUID) that uniquely identifies this annotation. If you know the GUID for an annotation, you can retrieve the corresponding <code>Annotation</code> object using the <code>AnnotationStore.GetAnnotation()</code> method. (Of course, there's no reason you'd know the GUID of an existing annotation unless you had previously retrieved it by calling <code>GetAnnotations()</code> , or you had reacted to an <code>AnnotationStore</code> event when the annotation was created or changed.)
AnnotationType	The XML element name that identifies this type of annotation, in the format <code>namespace:localname</code> .
Anchors	A collection of zero, one, or more <code>AnnotationResource</code> objects that identify what text is being annotated.

Name	Description
Cargos	A collection of zero, one, or more AnnotationResource objects that contain the user data for the annotation. This includes the text of a text note, or the ink strokes for an ink note.
Authors	A collection of zero, one, or more strings that identify who created the annotation.
CreationTime	The date and time when the annotation was created.
LastModificationTime	The date and time the annotation was last updated.

The Annotation object is really just a thin wrapper over the XML data that's stored for the annotation. One consequence of this design is that it's difficult to pull information out of the Anchors and Cargos properties. For example, if you want to get the actual text of an annotation, you need to look at the second item in the Cargos selection. This contains the text, but it's stored as a Base64-encoded string (which avoids problems if the note contains characters that wouldn't otherwise be allowed in XML element content). If you want to actually view this text, it's up to you to write tedious code like this to crack it open:

```
// Check for text information.
if (annotation.Cargos.Count > 1)
{
    // Decode the note text.
    string base64Text = annotation.Cargos[1].Contents[0].InnerText;
    byte[] decoded = Convert.FromBase64String(base64Text);

    // Write the decoded text to a stream.
    MemoryStream m = new MemoryStream(decoded);

    // Using the StreamReader, convert the text bytes into a more
    // useful string.
    StreamReader r = new StreamReader(m);
    string annotationXaml = r.ReadToEnd();
    r.Close();

    // Show the annotation content.
    MessageBox.Show(annotationXaml);
}
```

This code gets the text of the annotation, wrapped in a XAML <Section> element. The opening <Section> tag includes attributes that specify a wide range of typography details. Inside the <Section> element are more <Paragraph> and <Run> elements.

■ **Note** Like a text annotation, an ink annotation will also have a Cargos collection with more than one item. However, in this case the Cargos collection will contain the ink data but no decodable text. If you use the previous

code on an ink annotation, you'll get an empty message box. Thus, if your document contains both text and ink annotations, you should check the `Annotation.AnnotationType` property to make sure you're dealing with a text annotation before you use this code.

If you just want to get the text without the surrounding XML, you can use the `XamlReader` to deserialize it (and avoid using the `StreamReader`). The XML can be deserialized into a `Section` object, using code like this:

```
if (annotation.Cargos.Count > 1)
{
    // Decode the note text.
    string base64Text = annotation.Cargos[1].Contents[0].InnerText;
    byte[] decoded = Convert.FromBase64String(base64Text);

    // Write the decoded text to a stream.
    MemoryStream m = new MemoryStream(decoded);

    // Deserialize the XML into a Section object.
    Section section = XamlReader.Load(m) as Section;
    m.Close();

    // Get the text inside the Section.
    TextRange range = new TextRange(section.ContentStart, section.ContentEnd);

    // Show the annotation content.
    MessageBox.Show(range.Text);
}
```

As Table 28-8 shows, text isn't the only detail you can recover from an annotation. It's easy to get the annotation author, the time it was created, and the time it was last modified.

You can also retrieve information about where an annotation is anchored in your document. The `Anchor` collection isn't much help for this task, because it provides a low-level collection of `AnnotationResource` objects that wrap additional XML data. Instead, you need to use the `GetAnchorInfo()` method of the `AnnotationHelper` class. This method takes an annotation and returns an object that implements `IAnchorInfo`.

```
IAnchorInfo anchorInfo = AnnotationHelper.GetAnchorInfo(service, annotation);
```

`IAnchorInfo` combines the `AnnotationResource` (the `Anchor` property), the annotation (`Annotation`), and an object that represents the location of the annotation in the document tree (`ResolvedAnchor`), which is the most useful detail. Although the `ResolvedAnchor` property is typed as an object, text annotations and highlights always return a `TextAnchor` object. The `TextAnchor` describes the starting point of the anchored text (`BoundingStart`) and the ending point (`BoundingEnd`).

Here's how you could determine the highlighted text for an annotation using the `IAnchorInfo`:

```
IAnchorInfo anchorInfo = AnnotationHelper.GetAnchorInfo(service, annotation);
TextAnchor resolvedAnchor = anchorInfo.ResolvedAnchor as TextAnchor;
if (resolvedAnchor != null)
{
    TextPointer startPointer = (TextPointer)resolvedAnchor.BoundingStart;
```

```

    TextPointer endPointer = (TextPointer)resolvedAnchor.BoundingEnd;

    TextRange range = new TextRange(startPointer, endPointer);
    MessageBox.Show(range.Text);
}

```

You can also use the `TextAnchor` objects as a jumping-off point to get to the rest of the document tree, as shown here:

```

// Scroll the document so the paragraph with the annotated text is displayed.
TextPointer textPointer = (TextPointer)resolvedAnchor.BoundingStart;
textPointer.Paragraph.BringIntoView();

```

The samples for this chapter include an example that uses this technique to create an annotation list. When an annotation is selected in the list, the annotated portion of the document is shown automatically.

In both cases, the `AnnotationHelper.GetAnchorInfo()` method allows you to travel from the annotation to the annotated text, much as the `AnnotationStore.GetAnnotations()` method allows you to travel from the document content to the annotations.

Although it's relatively easy to examine existing annotations, the WPF annotation feature isn't as strong when it comes to manipulating these annotations. It's easy enough for the user to open a sticky note, drag it to a new position, change the text, and so on, but it's not easy for you to perform these tasks programmatically. In fact, all the properties of the `Annotation` object are read-only. There are no readily available methods to modify an annotation, so annotation editing involves deleting and re-creating the annotation. You can do this using the methods of the `AnnotationStore` or the `AnnotationHelper` (if the annotation is attached to the currently selected text). However, both approaches require a fair bit of grunt work. If you use the `AnnotationStore`, you need to construct an `Annotation` object by hand. If you use the `AnnotationHelper`, you need to explicitly set the text selection to include the right text before you create the annotation. Both approaches are tedious and unnecessarily error-prone.

Reacting to Annotation Changes

You've already learned how the `AnnotationStore` allows you to retrieve the annotations in a document (with `GetAnnotations()`) and manipulate them (with `DeleteAnnotation()` and `AddAnnotation()`). The `AnnotationStore` provides one additional feature—it raises events that inform you when annotations are changed.

The `AnnotationStore` provides four events: `AnchorChanged` (which fires when an annotation is moved), `AuthorChanged` (which fires when the author information of an annotation changes), `CargoChanged` (which fires when annotation data, including text, is modified), and `StoreContentChanged` (which fires when an annotation is created, deleted, or modified in any way).

The online samples for this chapter include an annotation-tracking example. An event handler for the `StoreContentChanged` event reacts when annotation changes are made. It retrieves all the annotation information (using the `GetAnnotations()` method) and then displays the annotation text in a list.

Note The annotation events occur after the change has been made. That means there's no way to plug in custom logic that extends an annotation action. For example, you can't add just-in-time information to an annotation or selectively cancel a user's attempt to edit or delete an annotation.

Storing Annotations in a Fixed Document

The previous examples used annotations on a flow document. In this scenario, annotations can be stored for future use, but they must be stored separately—for example, in a distinct XML file.

When using a fixed document, you can use the same approach, but you have an additional option—you can store annotations directly in the XPS document file. In fact, you could even store multiple sets of distinct annotations, all in the same document. You simply need to use the package support in the `System.IO.Packaging` namespace.

As you learned earlier, every XPS document is actually a ZIP archive that includes several files. When you store annotations in an XPS document, you are actually creating another file inside the ZIP archive.

The first step is to choose a URI to identify your annotations. Here's an example that uses the name `AnnotationStream`:

```
Uri annotationUri = PackUriHelper.CreatePartUri(
    new Uri("AnnotationStream", UriKind.Relative));
```

Now you need to get the `Package` for your XPS document using the static `PackageStore.GetPackage()` method:

```
Package package = PackageStore.GetPackage(doc.Uri);
```

You can then create the package part that will store your annotations inside the XPS document. However, you need to check if the annotation package part already exists (in case you've loaded the document before and already added annotations). If it doesn't exist, you can create it now:

```
PackagePart annotationPart = null;
if (package.PartExists(annotationUri))
{
    annotationPart = package.GetPart(annotationUri);
}
else
{
    annotationPart = package.CreatePart(annotationUri, "Annotations/Stream");
}
```

The last step is to create an `AnnotationStore` that wraps the annotation package part, and then enable the `AnnotationService` in the usual way:

```
AnnotationStore store = new XmlStreamStore(annotationPart.GetStream());
service = new AnnotationService(docViewer);
service.Enable(store);
```

In order for this technique to work, you must open the XPS file using `FileMode.ReadWrite` mode rather than `FileMode.Read`, so the annotations can be written to the XPS file. For the same reason, you need to keep the XPS document open while the annotation service is at work. You can close the XPS document when the window is closed (or you choose to open a new document).

Customizing the Appearance of Sticky Notes

The note windows that appear when you create a text note or ink note are instances of the `StickyNoteControl` class, which is found in the `System.Windows.Controls` namespace. Like all WPF controls, you can customize the visual appearance of the `StickyNoteControl` using style setters or applying a new control template.

For example, you can easily create a style that applies to all `StickyNoteControl` instances using the `Style.TargetType` property. Here's an example that gives every `StickyNoteControl` a new background color:

```
<Style TargetType="{x:Type StickyNoteControl}">
  <Setter Property="Background" Value="LightGoldenrodYellow"/>
</Style>
```

To make a more dynamic version of the `StickyNoteControl`, you can write a style trigger that responds to the `StickyNoteControl.IsActive` property, which is true when the sticky note has focus.

For more control, you can use a completely different control template for your `StickyNoteControl`. The only trick is that the `StickyNoteControl` template varies depending on whether it's used to hold an ink note or a text note. If you allow the user to create both types of notes, you need a trigger that can choose between two templates. Ink notes must include an `InkCanvas`, and text notes must contain a `RichTextBox`. In both cases, this element should be named `PART_ContentControl`.

Here's a style that applies the bare minimum control template for both ink and text sticky notes. It sets the dimensions of the note window and chooses the appropriate template based on the type of note content:

```
<Style x:Key="MinimumStyle" TargetType="{x:Type StickyNoteControl}">
  <Setter Property="OverridesDefaultStyle" Value="true" />
  <Setter Property="Width" Value="100" />
  <Setter Property="Height" Value="100" />
  <Style.Triggers>
    <Trigger Property="StickyNoteControl.StickyNoteType"
      Value="{x:Static StickyNoteType.Ink}">
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate>
            <InkCanvas Name="PART_ContentControl" Background="LightYellow" />
          </ControlTemplate>
        </Setter.Value>
      </Setter>
    </Trigger>
    <Trigger Property="StickyNoteControl.StickyNoteType"
      Value="{x:Static StickyNoteType.Text}">
      <Setter Property="Template">
        <Setter.Value>
          <ControlTemplate>
            <RichTextBox Name="PART_ContentControl" Background="LightYellow"/>
          </ControlTemplate>
        </Setter.Value>
      </Setter>
    </Trigger>
  </Style.Triggers>
</Style>
```

The Last Word

Most developers already know that WPF offers a new model for drawing, layout, and animation. However, its rich document features are often overlooked.

In this chapter, you've seen how to create flow documents, lay out text inside them in a variety of ways, and control how that text is displayed in different containers. You also learned how to use the `FlowDocument` object model to change portions of the document dynamically, and you considered the `RichTextBox`, which provides a solid base for advanced text editing features.

Lastly, you took a quick look at fixed documents and the `XpsDocument` class. The XPS model provides the plumbing for WPF's new printing feature, which is the subject of the next chapter.



Printing

Printing in WPF is vastly more powerful than it was with Windows Forms. Tasks that weren't possible using the .NET libraries and that would have forced you to use the Win32 API or WMI (such as checking a print queue) are now fully supported using the classes in the new System.Printing namespace.

Even more dramatic is the thoroughly 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 (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.

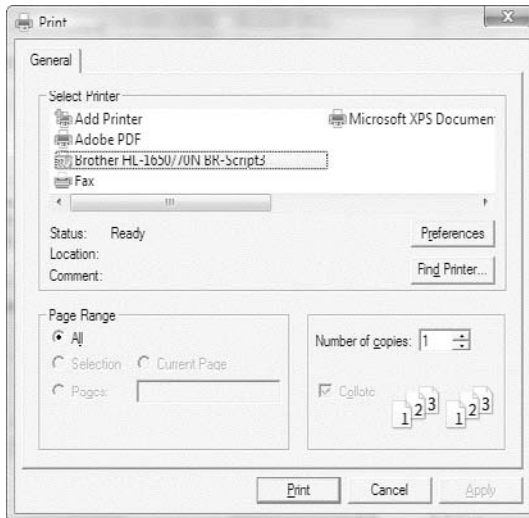


Figure 29-1. Showing the `PrintDialog`

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).

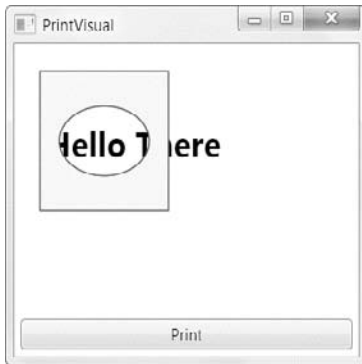


Figure 29-2. A simple drawing

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 will fill 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 will keep 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.

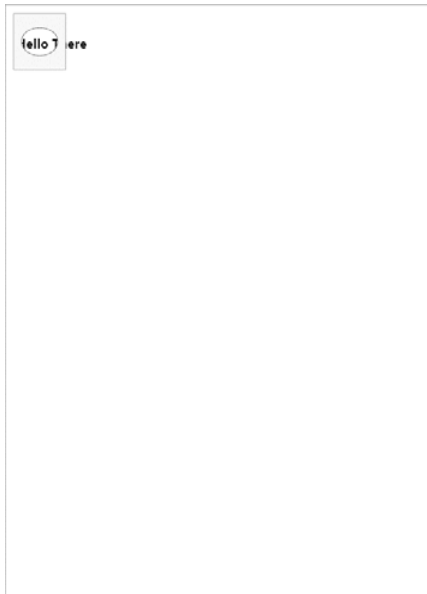


Figure 29-3. A printed element

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` includes the ability to 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 have the ability to 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 will hide 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 8.5~TMS11 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.

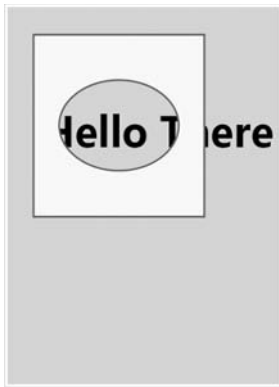


Figure 29-4. A scaled printed element

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);

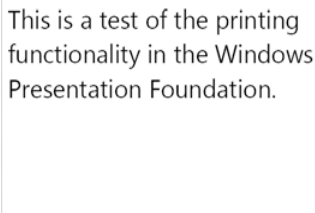
    // Allow wrapping to fit the page width.
    visual.TextWrapping = TextWrapping.Wrap;

    // 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 code works fine. You'll end up with a multipaged printout with two columns (on a standard sheet of 8.5-TMS11 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.Padding.Left - doc.Padding.Right) / 2;

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

// Reapply the old settings.
doc.PageHeight = pageHeight;
doc.PageWidth = pageWidth;
doc.Padding = 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 the text up 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; }
    }
}
```