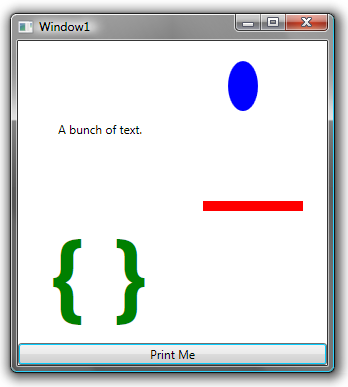
|  |  |
| --- | --- |
|  | [**Printing in WPF**](http://www.switchonthecode.com/tutorials/printing-in-wpf) |

http://www.switchonthecode.com/tutorials/printing-in-wpf

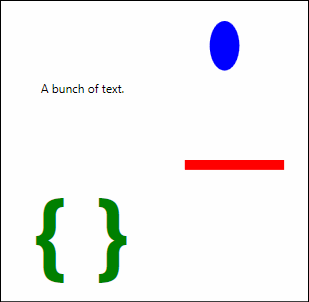
Printing. Ugh. Every programmer I know hates writing code to do printing. Stuff never seems to appear quite how it should - the transition from the screen to the page can often be a very messy one. Printing with the Win32 API is ugly - something more akin to black magic and dark incantations than actual computer code. The .NET framework made it a little bit better in WinForms, but it was still just a very thin wrapper around the Win32 api - and it was still in the world of GDI. But now we are in the land of WPF! The land of lollipops and hope and magical wonders! So everything should be awesome, right?

Well, actually, in many ways it is. It is a lot easier to just step right up and do some printing - and in all likelihood what you see on the screen will match what comes out on paper. This is due to two major factors. The obvious one is that they made a nicer API - it is a much thicker wrapper around the Win32 black magic than what WinForms had. The more subtle factor is the fact that WPF is all vector based graphics and does not deal directly with pixels. This means that many more things scale exactly as you expect when you go from the 96 DPI world of the computer monitor to the 300-1200 DPI world of the printer.

Ok, enough abstract talk. Let's do some printing. Below you can see a screenshot of a simple sample app. When the "Print Me" button is pressed, the canvas area above the button is printed.



And here is an example of what that printed output looks like (in this case, printed through the Windows XPS printer, so that I could take a screenshot):



And it looks like that if you print it out as well (assuming you print it in color :P). So what is the code to do this? Well, we of course defined that interface in a couple lines of XAML:

<Window x:Class="WpfPrinting.Window1"  
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
   Title="Window1" Height="360" Width="325">  
  <Grid>  
    <Grid.RowDefinitions>  
      <RowDefinition Height="\*" />  
      <RowDefinition Height="Auto" />  
    </Grid.RowDefinitions>  
    <Canvas Grid.Row="0" x:Name="\_PrintCanvas" Margin="10">  
      <TextBlock Canvas.Left="30" Canvas.Top="70">  
        A bunch of text.  
      </TextBlock>  
      <Ellipse Width="30" Height="50" Canvas.Left="200"   
              Canvas.Top="10" Fill="Blue" />  
      <Rectangle Width="100" Height="10" Canvas.Left="175"   
                Canvas.Top="150" Fill="Red" />  
      <TextBlock FontSize="100" Foreground="Green"   
                FontWeight="Bold" Canvas.Top="150"   
                Canvas.Left="20">  
        { }  
      </TextBlock>  
    </Canvas>  
    <Button Click="PrintClick" Grid.Row="1">  
      Print Me  
    </Button>  
  </Grid>  
</Window>

So now we have that canvas area defined in XAML. How do we print it? Well, that is the great thing about WPF printing - you can print any Visual! This canvas is a visual, so we can just straight up print it:

private void PrintClick(object sender, RoutedEventArgs e)  
{  
  PrintDialog dialog = new PrintDialog();  
  if (dialog.ShowDialog() == true)  
  { dialog.PrintVisual(\_PrintCanvas, "My Canvas"); }  
}

That is actually all you need to do to print a visual. Well, actually, you don't even need to show the dialog - it would just print to the default printer, then. All we are doing here is creating a [PrintDialog](http://msdn.microsoft.com/en-us/library/system.windows.controls.printdialog.aspx), showing it, and (if the user clicks ok) calling [PrintVisual](http://msdn.microsoft.com/en-us/library/system.windows.controls.printdialog.aspx) (which takes a visual and a name).

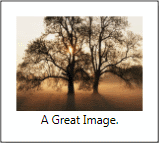
Calling PrintVisual will always print what you give it to a single page, and it will be placed in the very upper left of the page. This means that if your visual is larger than a page, it will get clipped, and if your visual has no built in margin, it will probably clip a little bit on the edges (because most printers can't print on the very edges of pages).

For doing more complex, multiple page printing, there is another method called [PrintDocument](http://msdn.microsoft.com/en-us/library/system.windows.controls.printdialog.printdocument.aspx), which takes a [DocumentPaginator](http://msdn.microsoft.com/en-us/library/system.windows.documents.documentpaginator.aspx). However, that is a topic for another tutorial.

Ok, so we have seen how to print out a visual that already exists - but how do we print out something we have created on the fly? Well, all it does is take a couple extra lines:

private void PrintSomethingNew()  
{  
  PrintDialog dialog = new PrintDialog();  
  if (dialog.ShowDialog() != true)  
  { return; }  
  
  StackPanel myPanel = new StackPanel();  
  myPanel.Margin = new Thickness(15);  
  Image myImage = new Image();  
  myImage.Width = 128;  
  myImage.Stretch = Stretch.Uniform;  
  myImage.Source = new BitmapImage(new Uri("C:\\Tree.jpg", UriKind.Absolute));  
  myPanel.Children.Add(myImage);  
  TextBlock myBlock = new TextBlock();  
  myBlock.Text = "A Great Image.";  
  myBlock.TextAlignment = TextAlignment.Center;  
  myPanel.Children.Add(myBlock);  
  
  myPanel.Measure(new Size(dialog.PrintableAreaWidth,  
    dialog.PrintableAreaHeight));  
  myPanel.Arrange(new Rect(new Point(0, 0),   
    myPanel.DesiredSize));  
  
  dialog.PrintVisual(myPanel, "A Great Image.");  
}

This code prints out something that looks like this:



As you might expect, we have to do a bunch of element creation so we can get something laid out. The interesting pieces are the call to Measure and Arrange. Since these elements have never appeared on the screen, they have never been rendered. But if we don't render them, then the printout will just be blank. So we have to do our own calls to Measure and Arrange. In this case, we do a Measure using the page size of the printer we are printing to. This page size is available off of the PrintDialog, using the PrintableAreaWidth and PrintableAreaHeight properties. And once measuring is done, we arrange (using the desired size produced through the measure pass).

You might think that those two properties represent the printable area on a page, but it is actually just the exact size of the page in DIU (so for an 8.5x11 piece of paper, the width and height are 816 and 1056, respectively). This means that you still need to account for the fact that most printers can't print on the extreme edge of a page, which is why I set a margin on myPanel.

Well, that is it for the basics of printing in WPF. You can grab the the Visual Studio project for this code below. And stay tuned for more printing tutorials - delving into writing your own DocumentPaginator, as well as using the PrintQueue and the XpsDocumentWriter.

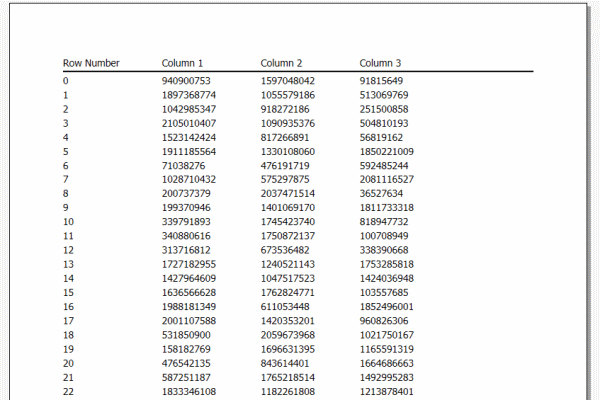
|  |
| --- |
| [**WPF Printing Part 2 - Pagination**](http://www.switchonthecode.com/tutorials/wpf-printing-part-2-pagination) |

http://www.switchonthecode.com/tutorials/wpf-printing-part-2-pagination

About two weeks ago, we had a tutorial here at SOTC on the basics of [printing in WPF](http://www.switchonthecode.com/tutorials/printing-in-wpf). It covered the standard stuff, like popping the print dialog, and what you needed to do to print visuals (both created in XAML and on the fly). But really, that's barely scratching the surface - any decent printing system in pretty much any application needs to be able to do a lot more than that. So today, we are going to take one more baby step forward into the world of printing - we are going to take a look at pagination.

The main class that we will need to do pagination is the [DocumentPaginator](http://msdn.microsoft.com/en-us/library/system.windows.documents.documentpaginator.aspx). I mentioned this class very briefly in the previous tutorial, but only in the context of the printing methods on [PrintDialog](http://msdn.microsoft.com/en-us/library/system.windows.controls.printdialog.aspx), [PrintVisual](http://msdn.microsoft.com/en-us/library/system.windows.controls.printdialog.aspx) (which we focused on last time) and [PrintDocument](http://msdn.microsoft.com/en-us/library/system.windows.controls.printdialog.printdocument.aspx) (which we will be focusing on today). This PrintDocument function takes a DocumentPaginator to print - and this is why we need to create one. Unfortunately, making a DocumentPaginator is not as easy as just creating an instance. It is an abstract class, and so you actually need to create your own class that derives from it, filling in all the abstract methods with the functionality that you need.

Today, we are going to be writing our own DocumentPaginator that spits out random tabular data - I called it the RandomTabularPaginator. Every page printed will have a header, and then as many rows of data as can be fit onto the page. Below you can see the top of an example printed page:



So let's take a look at what we are getting into by trying to write our own DocumentPaginator. The following is what Visual Studio stubs out for you when you create a class that derives from DocumentPaginator:

public class RandomTabularPaginator : DocumentPaginator  
{  
  */// <summary>*  
  */// When overridden in a derived class,*   
  */// gets the DocumentPage for the specified page number*  
  */// </summary>*  
  */// <param name="pageNumber">*  
  */// The zero-based page number of the document*   
  */// page that is needed.*  
  */// </param>*  
  */// <returns>*  
  */// The DocumentPage for the specified pageNumber,*   
  */// or DocumentPage.Missing if the page does not exist.*  
  */// </returns>*  
  public override DocumentPage GetPage(int pageNumber)  
  {  
    throw new NotImplementedException();  
  }  
  
  */// <summary>*  
  */// When overridden in a derived class, gets a value*   
  */// indicating whether PageCount is the total number of pages.*   
  */// </summary>*  
  public override bool IsPageCountValid  
  {  
    get { throw new NotImplementedException(); }  
  }  
  
  */// <summary>*  
  */// When overridden in a derived class, gets a count*   
  */// of the number of pages currently formatted.*  
  */// </summary>*  
  public override int PageCount  
  {  
    get { throw new NotImplementedException(); }  
  }  
  
  */// <summary>*  
  */// When overridden in a derived class, gets or*   
  */// sets the suggested width and height of each page.*  
  */// </summary>*  
  public override Size PageSize  
  {  
    get  
    {  
      throw new NotImplementedException();  
    }  
    set  
    {  
      throw new NotImplementedException();  
    }  
  }  
  
  */// <summary>*  
  */// When overridden in a derived class,*   
  */// returns the element being paginated.*  
  */// </summary>*  
  public override IDocumentPaginatorSource Source  
  {  
    get { throw new NotImplementedException(); }  
  }  
}

Most of those method comments are actually pretty self-explanatory. As you probably figured out, the really important method here is GetPage. This method will get called once for every page in your document (the number of pages in your document being defined by the PageCount property). Each time, you have to return a [DocumentPage](http://msdn.microsoft.com/en-us/library/system.windows.documents.documentpage.aspx) instance. This might sound onerous, but it actually isn't that bad (as we will see in a moment) - a DocumentPage is just a shell around a visual that you want to print.

Ok, time to get started filling out these methods. Below is the code for everything except the GetPage (because that method is a bit more complicated):

public class RandomTabularPaginator : DocumentPaginator  
{  
  private int \_RowsPerPage;  
  private Size \_PageSize;  
  private int \_Rows;  
  
  public RandomTabularPaginator(int rows, Size pageSize)  
  {  
    \_Rows = rows;  
    PageSize = pageSize;  
  }  
  
  public override DocumentPage GetPage(int pageNumber)  
  {  
    throw new NotImplementedException();  
  }  
  
  public override bool IsPageCountValid  
  { get { return true; } }  
  
  public override int PageCount  
  { get { return (int)Math.Ceiling(\_Rows / (double)\_RowsPerPage); } }  
  
  public override Size PageSize  
  {  
    get { return \_PageSize; }  
    set  
    {  
      \_PageSize = value;  
  
      \_RowsPerPage = PageElement.RowsPerPage(PageSize.Height);  
  
      *//Can't print anything if you can't fit a row on a page*  
      Debug.Assert(\_RowsPerPage > 0);  
    }  
  }  
  
  public override IDocumentPaginatorSource Source  
  { get { return null; } }  
}

As you can see, we first added a constructor and a couple of fields. The two things this RandomTabularPaginator needs to know are the number of rows of random data to generate, and the page size that the data will be printed on. We pull those in through the constructor and store them. When setting the page size, we calculate the number of rows of data that can fit on a page, and store that as well. If your wondering what the class PageElement is, don't worry - we will be getting to that real soon. For now, you will just have to believe that PageElement.RowsPerPage actually does return the number of rows that can fit on a page for a given page height.

In the PageCount property, we just return the ceiling of the total number of rows divided by the number of rows that can fit on a page. And since we always know the exact number of pages in our document, we can just flat out return true from IsPageCountValid. In more complex printing, you might be unable to calculate the total number of pages right off the bat - which is why this property is there.

Finally, we return null from the Source because we don't have an [IDocumentPaginatorSource](http://msdn.microsoft.com/en-us/library/system.windows.documents.idocumentpaginatorsource.aspx). Generally, you don't have to worry about this property and can always return null. It is used mostly by [FlowDocument](http://msdn.microsoft.com/en-us/library/system.windows.documents.flowdocument.aspx) and the like.

Now for the GetPage method:

public override DocumentPage GetPage(int pageNumber)  
{  
  int currentRow = \_RowsPerPage \* pageNumber;  
  
  var page = new PageElement(currentRow,   
    Math.Min(\_RowsPerPage, \_Rows - currentRow))  
  {  
    Width = PageSize.Width,  
    Height = PageSize.Height,  
  };  
  
  page.Measure(PageSize);  
  page.Arrange(new Rect(new Point(0,0), PageSize));       
  
  return new DocumentPage(page);  
}

First off, we calculate the current row, based off of the page number passed in and our stored value for the number of rows that fit on a page. Then we create a new PageElement for this page. This is a custom control built just for rendering a page of RandomTabularPaginator content - I generally find it easier to work with printing pages this way, because otherwise the GetPage method can get quite long with all the rendering logic. So before we move on to the rest of this method, let's take a look at the PageElement class:

public class PageElement : UserControl  
{  
  private const int PageMargin = 75;  
  private const int HeaderHeight = 25;  
  private const int LineHeight = 20;  
  private const int ColumnWidth = 140;  
  
  private int \_CurrentRow;  
  private int \_Rows;  
  
  public PageElement(int currentRow, int rows)  
  {  
    Margin = new Thickness(PageMargin);  
    \_CurrentRow = currentRow;  
    \_Rows = rows;  
  }  
  
  public static int RowsPerPage(double height)  
  {  
    return (int)Math.Floor((height - (2 \* PageMargin)  
      - HeaderHeight) / LineHeight);  
  }  
  
  private static FormattedText MakeText(string text)  
  {  
    return new FormattedText(text, CultureInfo.CurrentCulture,  
      FlowDirection.LeftToRight, new Typeface("Tahoma"), 14, Brushes.Black);  
  }  
  
  protected override void OnRender(DrawingContext dc)  
  {  
    Point curPoint = new Point(0, 0);  
  
    dc.DrawText(MakeText("Row Number"), curPoint);  
    curPoint.X += ColumnWidth;  
    for (int i = 1; i < 4; i++)  
    {  
      dc.DrawText(MakeText("Column " + i), curPoint);  
      curPoint.X += ColumnWidth;  
    }  
  
    curPoint.X = 0;  
    curPoint.Y += LineHeight;  
  
    dc.DrawRectangle(Brushes.Black, null,   
      new Rect(curPoint, new Size(Width, 2)));  
    curPoint.Y += HeaderHeight - LineHeight;  
  
    Random numberGen = new Random();  
    for (int i = \_CurrentRow; i < \_CurrentRow + \_Rows; i++)  
    {  
      dc.DrawText(MakeText(i.ToString()), curPoint);  
      curPoint.X += ColumnWidth;  
      for (int j = 1; j < 4; j++)  
      {  
        dc.DrawText(MakeText(numberGen.Next().ToString()), curPoint);  
        curPoint.X += ColumnWidth;  
      }  
      curPoint.Y += LineHeight;  
      curPoint.X = 0;  
    }  
  }  
}

This class takes in the row number to start at and the number of rows to generate, and stores them in fields. It also has a bunch of constants about various heights and widths. The real work in this class is in the OnRender method, which is where everything gets drawn. I choose to do everything in OnRender here, but you could have just as easily built up a visual tree with a grid and a whole bunch of TextBlocks. In this case, using OnRender is probably faster, but there are cases where building a full visual tree is better to do.

The actual contents of OnRender for this class are kind of boring - just a whole bunch of loops drawing [FormattedText](http://msdn.microsoft.com/en-us/library/system.windows.media.formattedtext.aspx) objects - first for the headers, and then for each of the rows. One thing you don't want to forget (and is easy to miss) is that you still need to deal with setting your own page margins (since printers can't print to the edge of pages). Here, we deal with it by setting the margin on the PageElement in its constructor, but you can also deal with it by just making sure you position elements appropriately when drawing or laying things out.

Ok, back to the rest of that GetPage method. Once we have a PageElement instance, we set its width and height, and then run it through a Measure/Arrange pass. You may remember from the previous tutorial that you always have to Measure/Arrange brand new visuals before passing them to the PrintVisual method on PrintDialog. The same concept applies here - if you don't Measure/Arrange, you will end up with blank sheets of paper. Finally, we take the PageElement, package it in a DocumentPage, and return it. And that's it for implementing the RandomTabularPaginator!

But now you are probably wondering how you use a DocumentPaginator that you have written. Well, it is really easy - the following is the XAML and C# code from the simple app that tested the RandomTabularPaginator:

<Window x:Class="PaginatorExample.Window1"  
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
   Title="Window1" Height="100" Width="100">  
  <StackPanel>  
    <StackPanel Orientation="Horizontal">  
      <TextBlock VerticalAlignment="Center">Rows: </TextBlock>  
      <TextBox x:Name="NumRows">50</TextBox>  
    </StackPanel>  
    <Button Click="PrintClick">Print</Button>  
  </StackPanel>  
</Window>

using System.Windows;  
using System.Windows.Controls;  
  
namespace PaginatorExample  
{  
  public partial class Window1 : Window  
  {  
    public Window1()  
    { InitializeComponent(); }  
  
    private void PrintClick(object sender, RoutedEventArgs e)  
    {  
      int rows = 0;  
      if(!int.TryParse(NumRows.Text, out rows) || rows < 0)  
      {  
        MessageBox.Show("Not a valid number of rows.");  
        return;          
      }  
        
      var printDialog = new PrintDialog();  
      if (printDialog.ShowDialog() == true)  
      {  
        var paginator = new RandomTabularPaginator(rows,  
          new Size(printDialog.PrintableAreaWidth,   
            printDialog.PrintableAreaHeight));  
  
        printDialog.PrintDocument(paginator, "My Random Data Table");  
      }  
    }  
  }  
}

This code pretty much comes down to three steps (after the work of getting the button click and grabbing the number of rows to generate). First, pop the print dialog. If the return value is true, you move on to step two - create a RandomTabularPaginator. We have the number of rows parsed already, so all we have to do is grab the currently selected page size off of the PrintDialog through the PrintableAreaWidth and PrintableAreaHeight properties. Finally, we pass the new RandomTabularPaginator to the PrintDocument, and we are done!

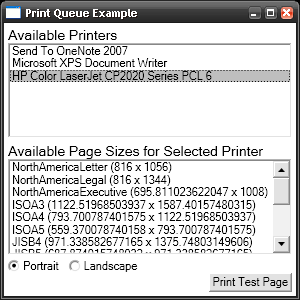
Well, that is it for this installment of printing in WPF. There is plenty more to come, though, because we still haven't event touched the PrintQueue or the XpsDocumentWriter. If you would like the code behind the example in this tutorial, you can grab a zip file containing the Visual Studio solution from the download link in the source files section below. As always, if you have any questions please leave a comment and I'll do my best to answer it.

|  |
| --- |
| [**WPF - Print Queues And Capabilities**](http://www.switchonthecode.com/tutorials/wpf-print-queues-and-capabilities) |

http://www.switchonthecode.com/tutorials/wpf-print-queues-and-capabilities

We have taken a look at printing in WPF twice before here at SOTC - first with a simple tutorial on just [getting something printed](http://www.switchonthecode.com/tutorials/printing-in-wpf), and then a more complex one on [pagination](http://www.switchonthecode.com/tutorials/wpf-printing-part-2-pagination). Today we are not going to focus much on the *printing* side of things, but more on the *printer* side. For example, how do you get a list of the printers available on the system? Or their capabilities? If you need the answers to those questions, then this is the tutorial for you.

Today, we will be creating a little sample application that finds all the printers on your system (both local and network printers) and lists them out. When you pick a particular printer, you will get a list of the supported page sizes for that printer. Once you pick a page size, you can then print a test page to the chosen printer at the chosen page size (and you can even pick landscape or portrait). Oh, and did I mention that we never have to show the standard print dialog for any of this?



Ok, to start off we first have to get a hold of the list of printers attached to the system - well, actually, the list of print queues attached to the system. These are not necessarily physical printers (for example, if you have a PDF Printer installed on your system), but they do represent something you can print to. Doing this is actually pretty easy - you just have to know where to go. First, we need to add the System.Printing dll as a reference to our Visual Studio project, since most of what we need resides in that dll. Once we have that, we want to get a hold of the local printer server - which really couldn't be any easier:

var server = new PrintServer();

By default, when you create a new [PrintServer](http://msdn.microsoft.com/en-us/library/system.printing.printserver.aspx) instance, it connects to the local print server. There are other constructors on PrintServer that take things like a path to a different machine (in case you wanted the print server, for, say, some central network system), but for today, all we care about is the local server.

Now that we have the the print server, to get all the available print queues, we need to call the method GetPrintQueues. This method has a number of different signatures to make it easy for you to get the queues that you want. The no argument version of the function will generally do what you need - it will return all the queues that are attached directly to that print server (in this case, any printer attached to your computer).

For our sample application, we want to go a little bit beyond that - we want to grab any network printers as well. To do this we need to use the GetPrintQueues call that takes an array of [EnumeratedPrintQueueTypes](http://msdn.microsoft.com/en-us/library/system.printing.enumeratedprintqueuetypes.aspx):

var server = new PrintServer();  
var queues = server.GetPrintQueues(new[] { EnumeratedPrintQueueTypes.Local,  
     EnumeratedPrintQueueTypes.Connections});

The no argument version of GetPrintQueues is equivalent to calling this with just EnumeratedPrintQueueTypes.Local, and by adding EnumeratedPrintQueueTypes.Connections, we get network printers.

Now that we have a collection of PrintQueues, let's take a look at what we can do with one. There is a whole bunch of stuff available off of [PrintQueue](http://msdn.microsoft.com/en-us/library/system.printing.printqueue.aspx) - you can look at printer status, what jobs are currently queued, and all sorts of other things. At the moment, though, we are interesting in printer capabilities. To get the capabilities, you call the method GetPrintCapabilities, which returns a [PrintCapabilities](http://msdn.microsoft.com/en-us/library/system.printing.printcapabilities.aspx). The capabilities class covers everything from paper size, to duplexing, even down to if the printer supports automatic stapling. Oh, and an important thing to note - the PrintCapabilities class is in the ReachFramework dll, so to do anything with capabilities, you have to add that dll to your Visual Studio project references.

var server = new PrintServer();  
var queues = server.GetPrintQueues(new[] { EnumeratedPrintQueueTypes.Local,  
     EnumeratedPrintQueueTypes.Connections});  
  
foreach (var queue in queues)  
{  
  Console.WriteLine(queue.Name);  
  var capabilities = queue.GetPrintCapabilities();  
  foreach (PageMediaSize size in capabilities.PageMediaSizeCapability)  
  { Console.WriteLine(size.ToString()); }  
  Console.WriteLine();  
}

The code above will print out, for each printer, every paper size that that printer is capable of handling. For example, on my system:

Send To OneNote 2007  
NorthAmericaLetter (816 x 1056)  
NorthAmericaTabloid (1056 x 1632)  
NorthAmericaLegal (816 x 1344)  
ISOA3 (1122.51968503937 x 1587.40157480315)  
ISOA4 (793.700787401575 x 1122.51968503937)  
ISOA5 (559.370078740158 x 793.700787401575)  
JISB4 (971.338582677165 x 1375.74803149606)  
JISB5 (687.874015748032 x 971.338582677165)  
JapanHagakiPostcard (377.952755905512 x 559.370078740158)  
  
HP Color LaserJet CP2020 Series PCL 6  
NorthAmericaLetter (816 x 1056)  
NorthAmericaLegal (816 x 1344)  
NorthAmericaExecutive (695.811023622047 x 1008)  
ISOA3 (1122.51968503937 x 1587.40157480315)  
ISOA4 (793.700787401575 x 1122.51968503937)  
ISOA5 (559.370078740158 x 793.700787401575)  
JISB4 (971.338582677165 x 1375.74803149606)  
JISB5 (687.874015748032 x 971.338582677165)  
NorthAmerica11x17 (1056 x 1632)  
NorthAmericaNumber10Envelope (395.716535433071 x 912)  
ISODLEnvelope (415.748031496063 x 831.496062992126)  
ISOC5Envelope (612.283464566929 x 865.511811023622)  
ISOB5Envelope (665.196850393701 x 944.88188976378)  
NorthAmericaMonarchEnvelope (371.905511811024 x 720)  
ISOA6 (396.850393700787 x 559.370078740158)

So that covers enough of the basics that we can start putting together the example application. First, we have most of the code behind:

using System;  
using System.Printing;  
using System.Windows;  
using System.Windows.Controls;  
using System.Windows.Data;  
  
namespace PrintQueuesExample  
{  
  public partial class Window1 : Window  
  {  
    PrintQueueCollection \_Printers;  
  
    public Window1()  
    {  
      \_Printers = new PrintServer().GetPrintQueues(new[] {  
          EnumeratedPrintQueueTypes.Local, EnumeratedPrintQueueTypes.Connections});  
  
      InitializeComponent();  
    }  
  
    public PrintQueueCollection Printers  
    { get { return \_Printers; } }  
  
    private void PrintTestPageClick(object sender, RoutedEventArgs e)  
    {  
      *//TODO: Print Test Page*  
    }  
  }  
  
  public class PrintQueueToPageSizesConverter : IValueConverter  
  {  
    public object Convert(object value, Type targetType,  
      object parameter, System.Globalization.CultureInfo culture)  
    {  
      return value == null ? null :  
        ((PrintQueue)value).GetPrintCapabilities().PageMediaSizeCapability;  
    }  
  
    public object ConvertBack(object value, Type targetType,   
      object parameter, System.Globalization.CultureInfo culture)  
    { throw new NotImplementedException(); }  
  }  
}

Nothing new here - this is just a reorganization of the code that we have already covered, getting it into a form that can be easily used by WPF controls (a public collection of the print queues, a converter to get from a print queue to a collection of PageMediaSizes). Now for some XAML:

<Window x:Class="PrintQueuesExample.Window1"  
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  
   xmlns:local="clr-namespace:PrintQueuesExample" x:Name="This"  
   Title="Print Queue Example" Height="300" Width="300">  
  <Window.Resources>  
    <local:PrintQueueToPageSizesConverter x:Key="printQueueToPageSizesConverter" />  
    <Canvas Margin="10" x:Key="MyPrintingExample">  
      <TextBlock Canvas.Left="30" Canvas.Top="70">  
        A bunch of text.  
      </TextBlock>  
      <Ellipse Width="30" Height="50" Canvas.Left="200"   
             Canvas.Top="10" Fill="Blue" />  
      <Rectangle Width="100" Height="10" Canvas.Left="175"   
               Canvas.Top="150" Fill="Red" />  
      <TextBlock FontSize="100" Foreground="Green"   
               FontWeight="Bold" Canvas.Top="150"   
               Canvas.Left="20">  
        { }  
      </TextBlock>  
    </Canvas>  
  </Window.Resources>  
      
  <Grid Margin="4">  
    <Grid.RowDefinitions>  
      <RowDefinition Height="Auto" />  
      <RowDefinition Height="\*" />  
      <RowDefinition Height="Auto" />  
      <RowDefinition Height="\*" />  
      <RowDefinition Height="Auto" />  
      <RowDefinition Height="Auto" />  
    </Grid.RowDefinitions>  
    <TextBlock Grid.Row="0" Text="Available Printers" FontSize="14" />  
    <ListBox x:Name="\_PrinterList" DisplayMemberPath="Name"   
            x:FieldModifier="private" Grid.Row="1"   
            ItemsSource="{Binding ElementName=This, Path=Printers}"  />  
    <TextBlock Text="Available Page Sizes for Selected Printer" FontSize="14"   
              Grid.Row="2" Margin="0 5 0 0"/>        
    <ListBox Grid.Row="3" x:Name="\_SizeList" x:FieldModifier="private"  
            ItemsSource="{Binding ElementName=\_PrinterList, Path=SelectedItem,   
                Converter={StaticResource printQueueToPageSizesConverter}}" />  
    <StackPanel Orientation="Horizontal" Margin="0 5 0 0" Grid.Row="4">  
      <RadioButton Content="Portrait" x:Name="\_PortraitRadio" Margin="0 0 10 0"  
                  x:FieldModifier="private" IsChecked="True" />  
      <RadioButton Content="Landscape" />  
    </StackPanel>  
    <Button Grid.Row="5" HorizontalAlignment="Right" Content="Print Test Page"   
           Click="PrintTestPageClick" />  
  </Grid>  
</Window>

Walking through this XAML, at the top we have some resources - notably the PrintQueueToPageSizesConverter and a Canvas. It might seem a little odd to have a Canvas in resources, but here it is the contents of our test page (the page that will get printed when a user clicks on the "Print Test Page" button). By putting it in the resources, we get the benefit of defining it in XAML, without the downside of it actually being in the visual tree of the Window.

Past that, we get to the meat. We have a ListView bound to the collection of print queues (Printers) that we created in the C# code. We then have a second ListView that is bound to the selected item in the first list view, using the PrintQueueToPageSizesConverter. This way, we get the collection of available PageMediaSizes for the selected print queue.

Then we have two radio buttons for portrait and landscape - these aren't attached to anything, we will just be querying their values when the user clicks "Print Test Page". And finally, we have the "Print Test Page" button, which is hooked to the method PrintTestPageClick.

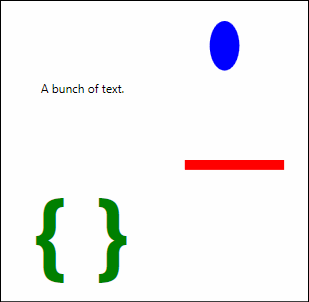
Now the only thing we haven't covered so far is how to take a print queue and some selected configuration information, and actually print something. For that, we have the contents of the PrintTestPageClick method:

private void PrintTestPageClick(object sender, RoutedEventArgs e)  
{  
  var queue = \_PrinterList.SelectedItem as PrintQueue;  
  if (queue == null)  
  {  
    MessageBox.Show("Please select a printer.");  
    return;  
  }  
  
  var size = \_SizeList.SelectedItem as PageMediaSize;  
  if (size == null)  
  {  
    MessageBox.Show("Please select a page size.");  
    return;  
  }  
  
  queue.UserPrintTicket.PageMediaSize = size;  
  queue.UserPrintTicket.PageOrientation = \_PortraitRadio.IsChecked == true ?   
    PageOrientation.Portrait : PageOrientation.Landscape;  
  
  var canvas = (Canvas)Resources["MyPrintingExample"];  
  canvas.Measure(new Size(size.Width.Value, size.Height.Value));  
  canvas.Arrange(new Rect(0, 0, canvas.DesiredSize.Width,   
      canvas.DesiredSize.Height));  
  
  var writer = PrintQueue.CreateXpsDocumentWriter(queue);  
  writer.Write(canvas);  
}

To set up custom settings for a print job, you want to modify the UserPrintTicker on the print queue you want to print to. The UserPrintTicket is what will be looked at when the time comes to print, and is in fact what the standard print dialog modifies as the user changes setting in the dialog. So here, we want to set the PageMediaSize property to the selected size, and the PageOrientation property to the selected orientation.

One important thing to note - just because a printer is capable of a particular paper size does not mean that it currently has a tray filled with that type of paper. Choosing a paper size that a printer supports but does not have any of is valid, and the end result varies depending on the printer. Some printers will print on the nearest possible size or their default size, others will wait until the user puts in the correct size paper. Unfortunately, there isn't a way (that I know of) to query what types of paper a printer has at this very moment - all the print capability stuff is about what a printer can potentially do.

Ok, back to printing out the test page. We grab the canvas out from the resource dictionary, and measure and arrange it according to the chosen paper size. Then we use a static method on PrintQueue called CreateXpsDocumentWriter to create an [XpsDocumentWriter](http://msdn.microsoft.com/en-us/library/system.windows.xps.xpsdocumentwriter.aspx) for the print queue we want to print on. It is this XpsDocumentWriter that we can hand our canvas to to print - and lo and behold, printed output:



Well, that is it for this quick introduction to PrintServers, PrintQueues, and PrintCapabilities. You can grab the Visual Studio project for the example application below if you would like to poke at the printers on your own computer. If you have any questions, or anything you would like me to address in my next printing tutorial, let me know in the comments.