

# Creating a Custom Table Style for Printing using PrintAdapters for .NET

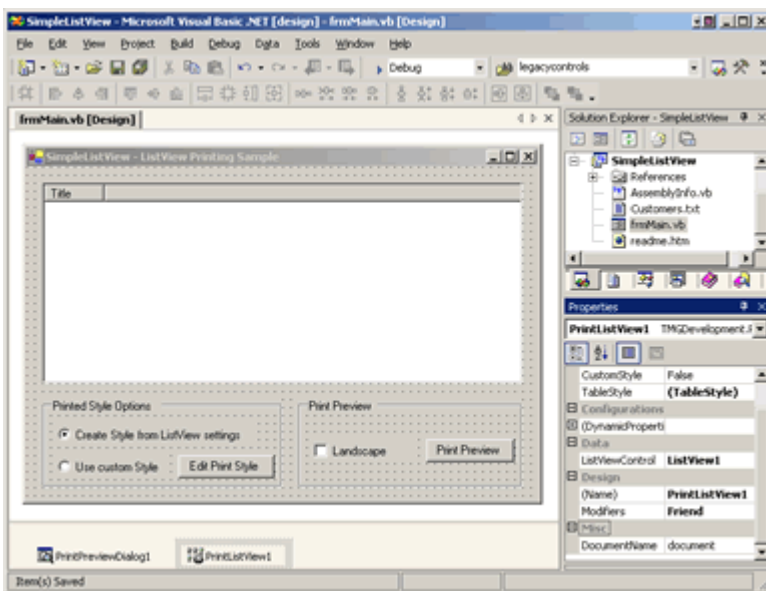
The PrintAdapters library provides a set of components that enable you to print some of the more common scrollable controls in the .NET Windows Forms control library: ListView, TreeView and RichTextBox. The features provided by PrintListView are also available in a similar component that can print any DataTable.

## Overview

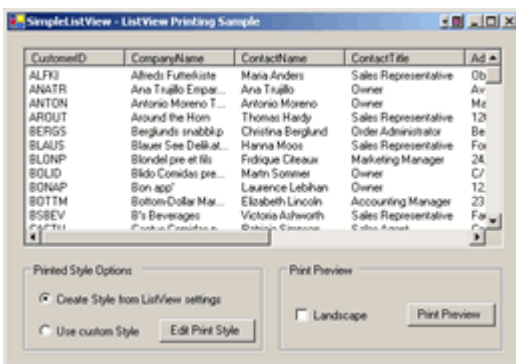
This how-to document will focus on working with just one of the PrintAdapters suite: PrintListView. PrintListView provides very good design time integration with Visual Studio that allows you to edit the style to be used when printing out your ListView, but it can also be accessed programmatically. We will develop a custom style using the designers, and then show how the same effect can be created from code.

## Using the PrintListView Designers

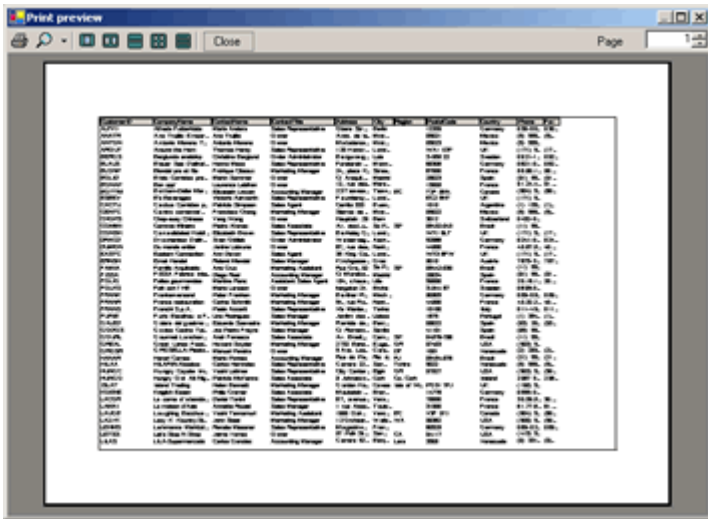
As a basis for these examples we will take the SimpleListView sample provided with the PrintAdapters library, which provides us with a populated ListView to work with. Here's SimpleListView in the Visual Studio designer:



If we hit F5 to run the project we see:



Checking 'Landscape' and clicking 'Print Preview' gives us the following print preview:



Note that because we left the radio button 'Create Style from ListView Settings' selected, PrintListView has automatically created a style that mirrors the current ListView appearance, including Font settings, Colors, Column widths and icon and checkbox settings.

If we now return to Visual Studio, and add another PrintPreviewDialog and PrintListView component to the Form. Then:

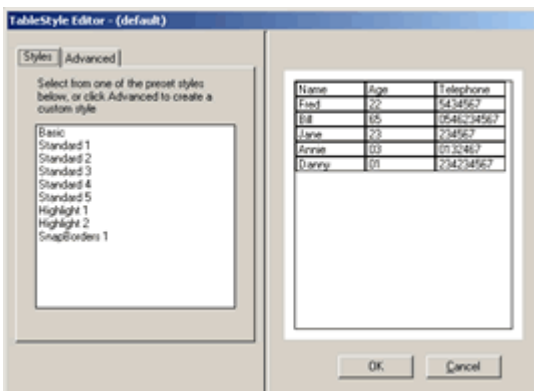
1. Set the new PrintPreviewDialog's Document property to reference the new PrintListView component
2. Set the new PrintListView's ListViewControl property to reference the ListView control
3. Set the PrintListView.CustomStyle property to True. This allows us to specify a custom TableStyle, otherwise the TableStyle would automatically be recreated from the ListView settings at print preview time.
4. Add a new button to the Form, and add a click handler with the following code:

```
Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As System.EventArgs) Handles Button1.Click
    Me.PrintPreviewDialog2.ShowDialog()
End Sub
```

Hit F5 and run this, and when you click on the new button you'll see the TableStyle in action (it is still the one picked up from ListView settings as in the Print Preview above).

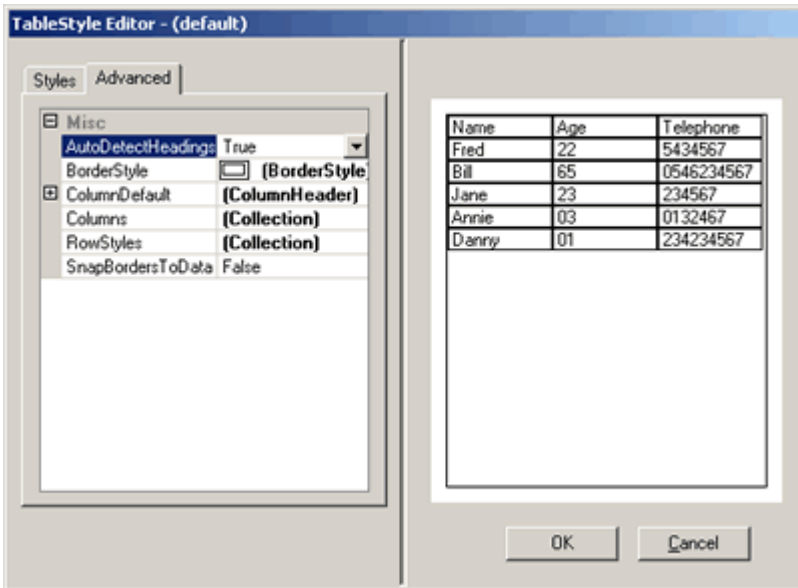
## Resetting the TableStyle

Now, right click the TableStyle property in the Property grid and select Reset, then click the ellipsis ('...') to the right of the property to bring up the TableStyle editor. Note that the TableStyle in the preview has reverted to a basic grid as a result of the reset:



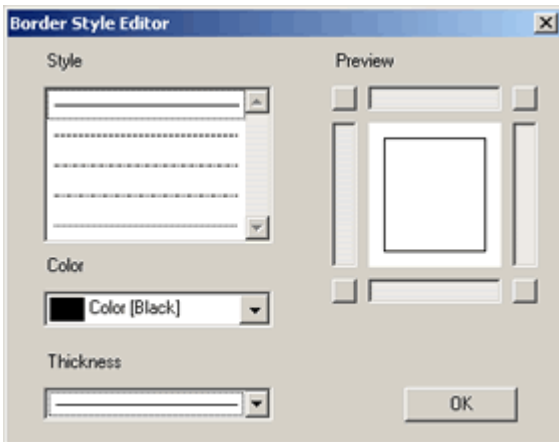
You can experiment with applying one of the predefined styles here by selecting from the list and clicking OK. When you run the sample and generate a print preview you should see the style applied to the printout.

We are going to build a style from scratch using the TableStyle designers however, so reset your TableStyle again to get back to the basic grid. Bring up the TableStyle designer and click on the 'Advanced' tab:



## TableStyle BackGrounder

Before we go any further; a bit of background about how a TableStyle is built up. It has a **BorderStyle**, which can be edited in the designer by clicking on the BorderStyle property. This allows you to independently set the style, color and thickness of each edge of the border. This editor is used to edit the border of individual cells in a table as well as the border of the entire table area.



The TableStyle also has the following properties:

Property	<input type="checkbox"/> Type	<input type="checkbox"/> Description
AutoDetectHeadings	Boolean	Indicates whether column heading text is detected automatically from ListView column headings, or specified via a ColumnHeader in the the Columns collection or the ColumnDefault. Setting true for this value will override any text specified in the corresponding ColumnHeader or the ColumnDefault. The column widths specified in the ColumnHeaders will be honored though even if the text is picked up automatically from the ListView.
BorderStyle	BorderStyle	Specifies the style of the border drawn around the bounding edge of the table.
ColumnDefault	ColumnHeader	

<input type="checkbox"/>	Columns	ColumnHeaderCollection	Allows you to specify the default text and column width to use when column headings aren't being auto detected with AutoDetectHeadings, and aren't specified in the Columns collection.
<input type="checkbox"/>	RowStyles	RowStyleCollection	ColumnHeader objects in this list are applied to columns in the table starting from the left hand side, when AutoDetectHeadings isn't set. For example, if you add two items to this collection the first two columns in the table will have the heading text and width you specify, and any subsequent columns will be based on the settings in ColumnDefault.
<input type="checkbox"/>	RowStyles	RowStyleCollection	Each row in the table can have its own RowStyle. By default the RowStyles you add to the collection are used as follows: the first one is used for the header row only, subsequent ones are used for data rows. If there are more data rows in the table than RowStyles, the RowStyle used loops back to the first (non- header) one. For example, if you add three RowStyles - the first is used for the header row, the second for all odd rows and the third for all even rows. If you add a further RowStyle you get a repeating pattern of three styles in the body of the table. If this default behavior is not what you want, you can handle the PrintListView.BeginRow event and set the RowStyle to be used for the row based on some other criteria like the data values in the row. For example you could use a highlighting style to pick out rows where values were negative etc.
<input type="checkbox"/>	SnapBordersToData	Boolean	This flag indicates whether the border is border drawn around the bounding edge of the table, or the bounding edge of the data. For example, when PrintListView behaves as a PrintDocument the edge of the table is considered to be the MarginBounds of the printed page, setting this value true in this situation would mean that the border would be drawn around the edge of the table, whatever size that was, rather than round the MarginBounds rectangle.

So we see from the above that each data row in the table can have a RowStyle. The first RowStyle in the RowStyleCollection is always applied to the header row, and subsequent RowStyles are applied in rotation to the data rows in the table. What does a RowStyle consist of?

### RowStyle Properties

Property	Type	Description	
<input type="checkbox"/>	CellStyles	CellStyleCollection	Specifies the collection of named styles that can be applied to cells in this row.
<input type="checkbox"/>	CellStyleMappings	CellStyleMappingCollection	Specifies a collection of CellStyleMapping objects, which map a CellStyle name to a column index. The CellStyleMapping.ColumnIndexNumbering property allows you to specify that the specified index is counted from either the left hand column or the right.
<input type="checkbox"/>	CellStyleDefault	CellStyle	This specifies the default CellStyle to use for cells that do not have a specific CellStyle allocated to them by a CellStyleMapping entry in the CellStyleMappings. If no specific styles are added to the CellStyles collection all the cells in the row will have this style.

Finally we reach the individual CellStyle, which has the following properties:

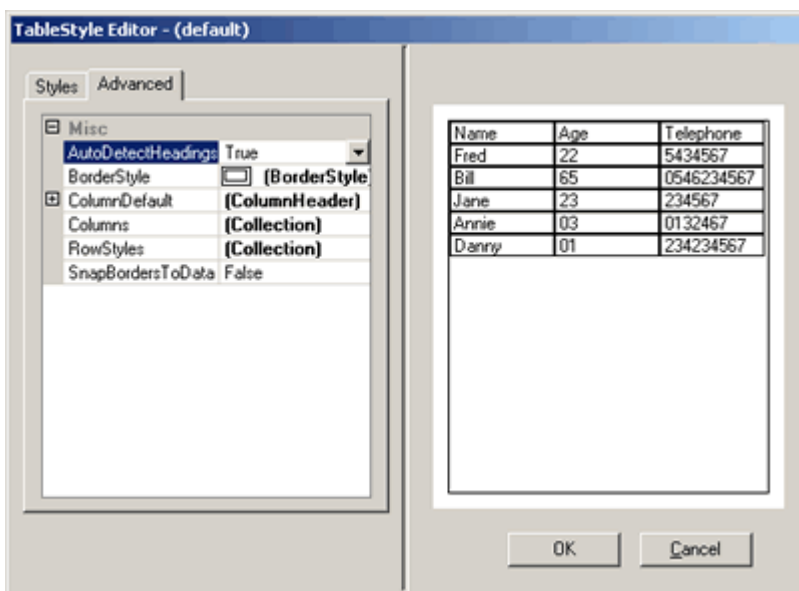
<input type="checkbox"/>
--------------------------

Property	Type	<input type="checkbox"/> Description
Name	String	Specifies the name used to identify the CellStyle in CellStyleMappings.
<input type="checkbox"/> Font	Font	Specifies the Font to be used for any text drawn in the cell.
<input type="checkbox"/> BackColor	Color	Specifies the background fill color of the cell.
<input type="checkbox"/> ForeColor	Color	Specifies the foreground color to be used for any content if the cell (e.g. text).
<input type="checkbox"/> BorderStyles	BorderStyle	This specifies borders to be drawn around the individual cell.

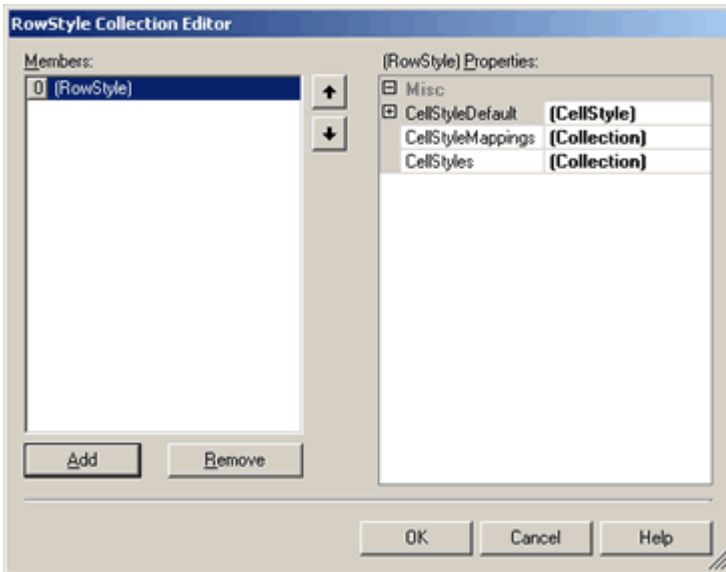
## Building a Custom Style

Now that may seem pretty complicated, but we're going to show that we can produce quite complicated results with just a little bit of clicking in the designer. First, back to our project... we have a TableStyle which has been reset to the default i.e. no style settings set at all.

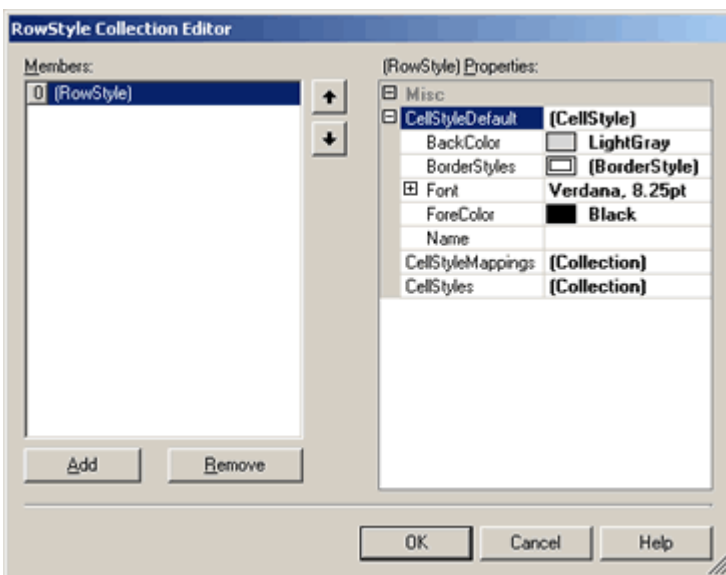
Starting with the designer in advanced mode again:



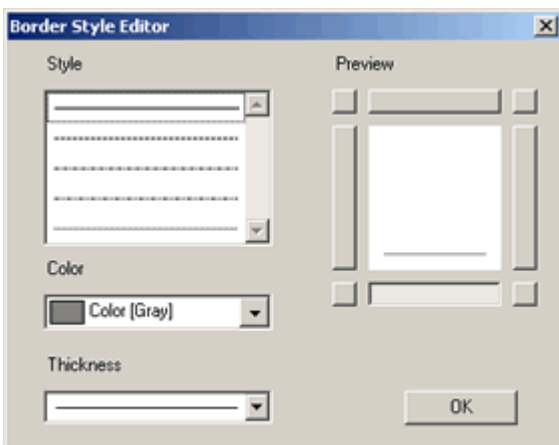
Click on the RowStyles property to bring up the editor:



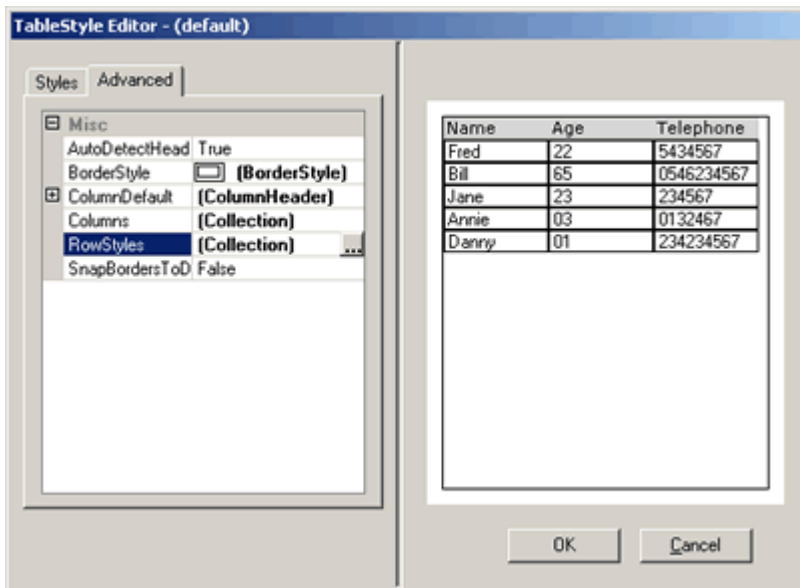
Expand CellStyleDefault, and pick a BackGround color of LightGray and Font of Verdana 8pt:



On the same dialog, click on BorderStyles to bring up the border style editor, select Color Gray, then use the buttons around the edge of the Preview to turn the line for that edge on or off. Turn all but the bottom line off ( Tip: to apply the new color and thickness settings to an edge, turn the line off, then back on again with the new settings):

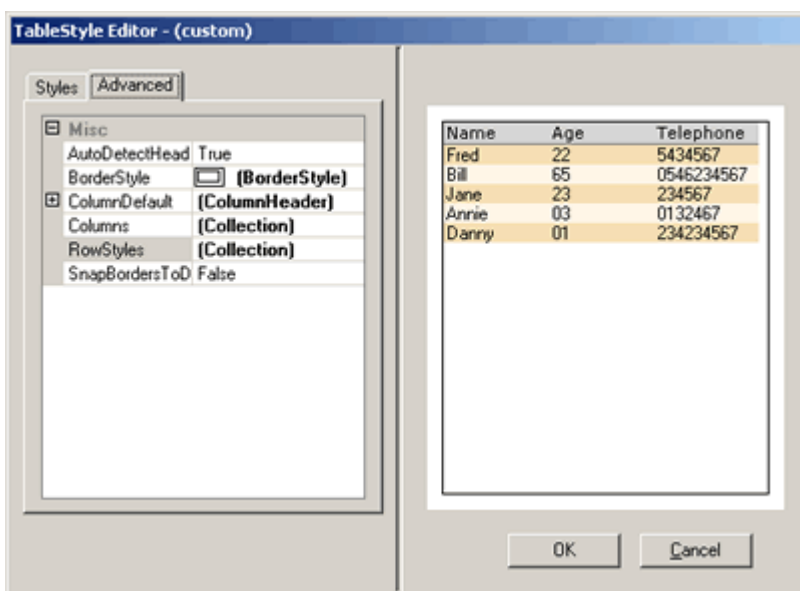


Clicking OK until we return to the TableStyle editor, where we can see the effects of our editing:



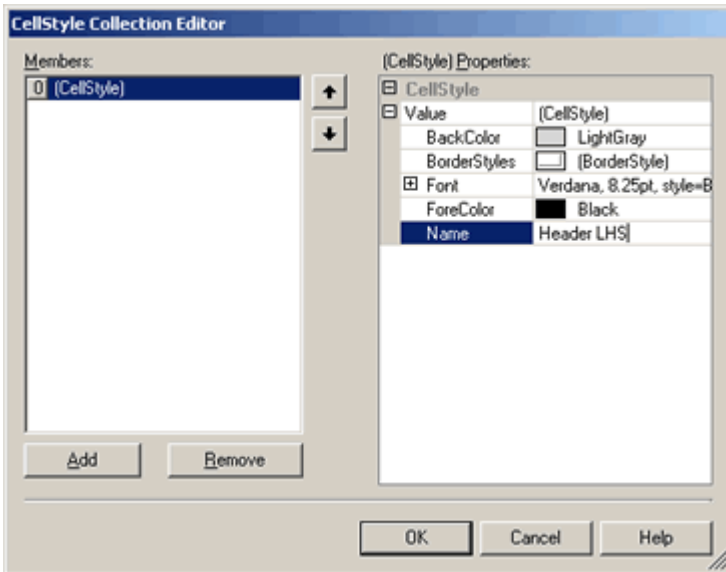
Now we'll add two more RowStyles, one with a BackColor of Wheat, and the other with a BackColor of OldLace. Turn off all the edges on the BorderStyle for both.

Note the way that the two new RowStyles are repeatedly applied to the rows of the table, while the first RowStyle is only ever used for the first (header) row:



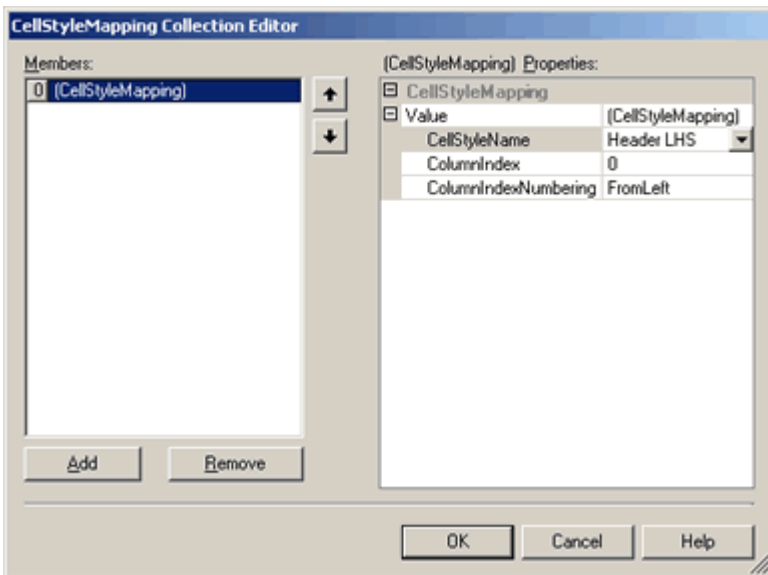
Lets make the Font of the left hand column **Bold**, and put a vertical line down between it and the second column. To do this we need to create a custom CellStyle for the first column in each of the RowStyles we've created.

1. Starting with the header row, bring up the RowStyles editor
2. Select RowStyle 0 then click on its CellStyles collection to bring up the CellStyles editor
3. Add a new CellStyle and expand its properties
4. First we need to duplicate the settings we have already made for this row (Color=LightGray, Font=Verdana,8pt, BorderStyle=BottomEdge:Gray)
5. Then we'll add the following settings: make the Font **Bold**, add a Gray right-hand edge line to the **BorderStyle**
6. Finally give this style a Name, 'Header LHS':

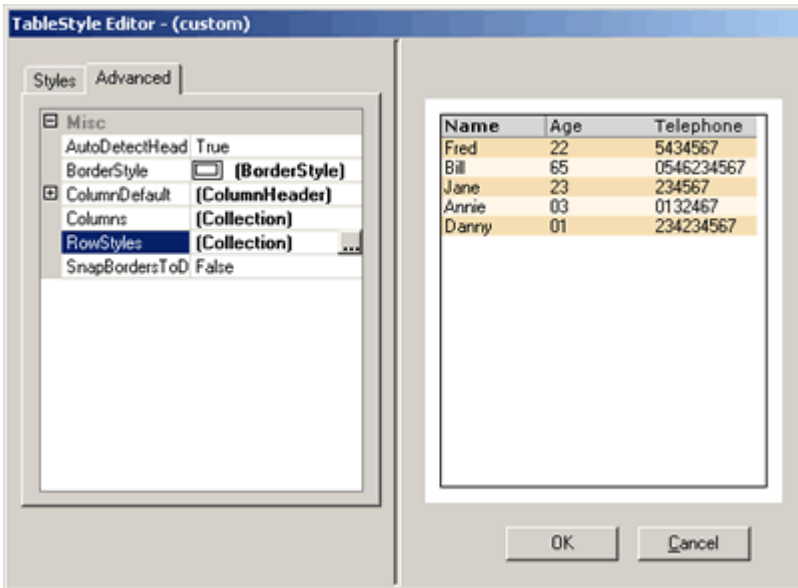


1. Now OK the CellStyles editor, and click to open the CellStyleMappings editor.
2. Add a new CellStyleMapping.
3. Click on CellStyleName, and pick or type 'Header LHS':

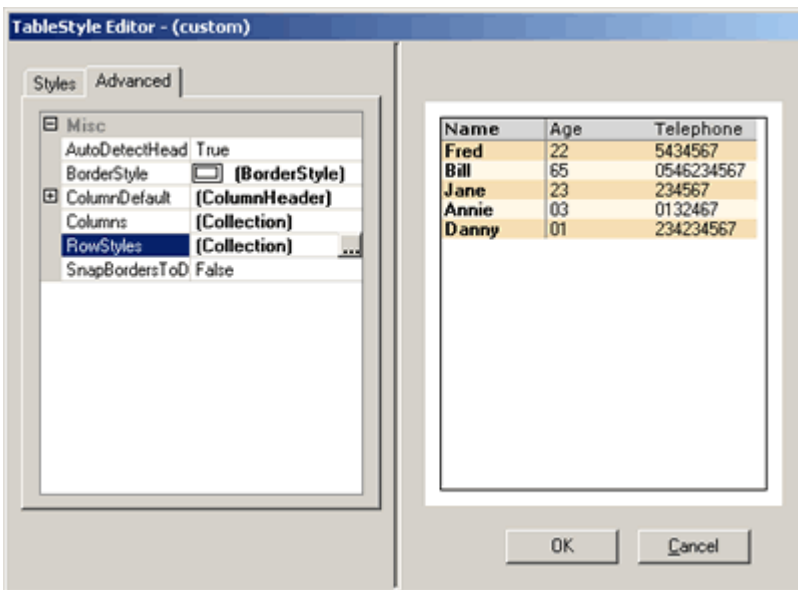
Note: the `ColumnIndexNumbering` property of `CellStyleMapping` can hold values of either `FromLeft` or `FromRight`. We've left this at the default of `FromLeft`, but if we selected `FromRight` the 'Header LHS' style would be applied to the rightmost column in the final table. This can be useful if you're defining a `TableStyle` and want to highlight the righthand column in some way, but you don't yet know how many columns there will be in the table.



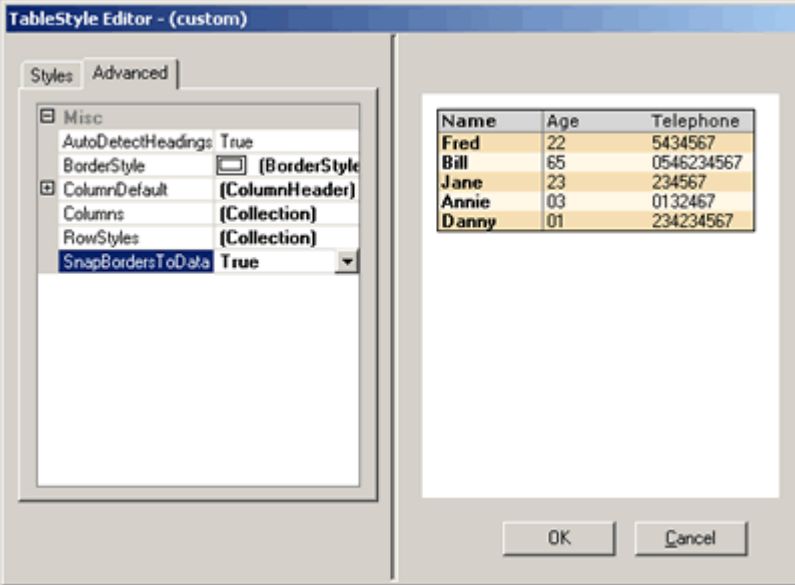
Click OK, and return to the TableStyle editor:



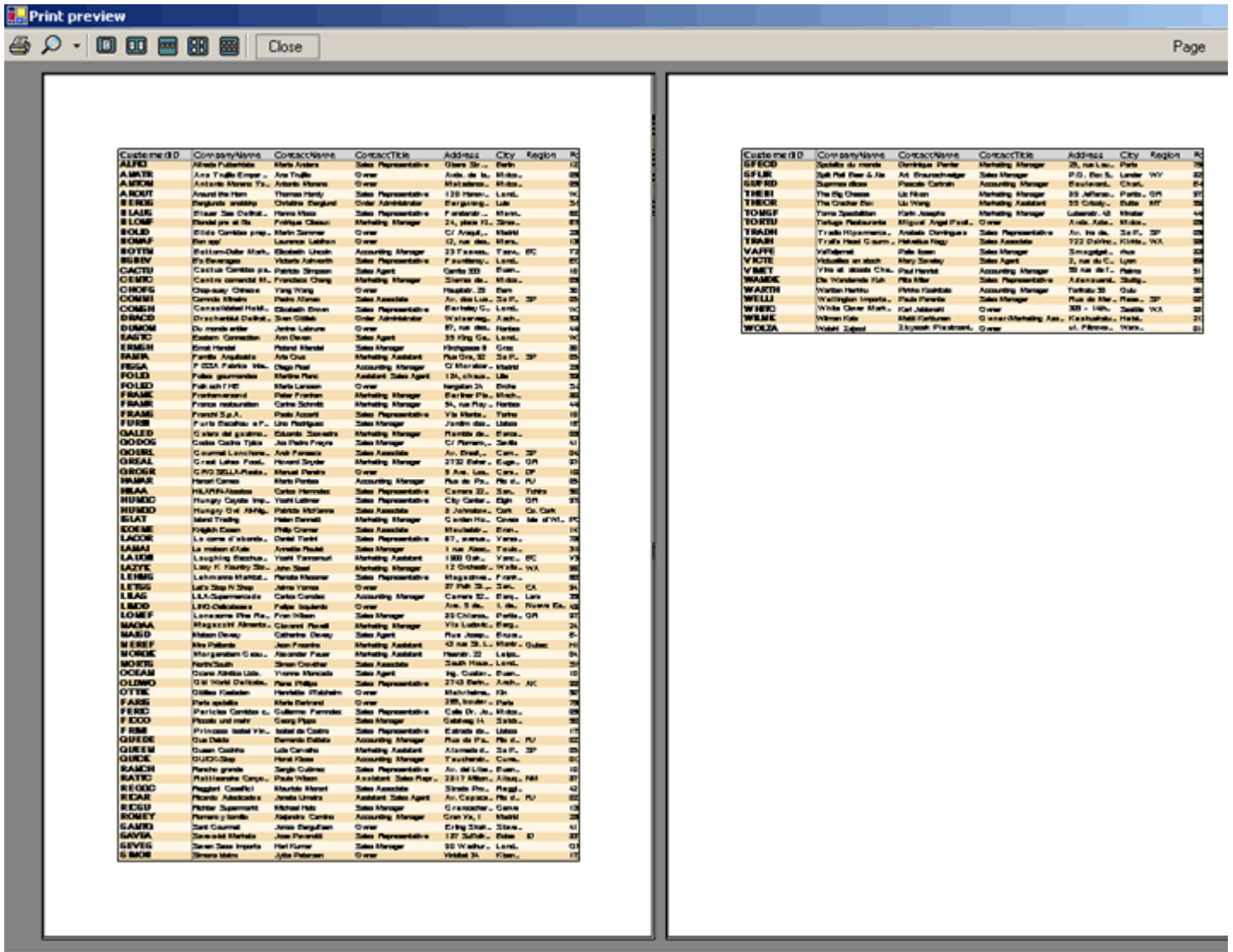
Repeat this process to add a CellStyle and CellStyleMapping to the other two RowStyles and we have:



As a final touch set SnapBordersToData to True:



If we now OK this and run the application, clicking on our new button on the main form gives us a print preview in the style we have just designed:



# Creating TableStyles Programmatically

Now that we've been through the process of designing a style using the PrintAdapters designers, doing the same from code will make a lot more sense. Incidentally, if you just want to pick up the code that represents a certain style you have designed via the UI as a starting point, you can copy the code section for the PrintListView.TableStyle from the "Windows Form Designer generated code" section of your form. This is possible because PrintAdapters serializes out its state into code using the .NET CodeDom serialization mechanism.

Most of the following code is self-explanatory. If you're not familiar with the array initializer construct in VB.NET it can look a bit strange - basically, to instantiate and populate an array in one go we can use:

```
Dim arr As String() = New String() {"one", "two", "three"}
```

Apart from that, the process we're going through in code is exactly the same as the one we went through in the designer and should have identical results.

Putting the following VB.NET code in our button click handler will bypass the TableStyle we set up in the designer, and replace it with the one we're coding manually.

' Need to make sure PrintListView knows we're setting a custom TableStyle

```
Me.PrintListView2.CustomStyle = True
```

' Create the standard Fonts we will use throughout the table

```
Dim fontHeader As Font = New Font("Verdana", 8.25!, System.Drawing.FontStyle.Regular)
```

```
Dim fontHeaderBold As Font = New Font("Verdana", 8.25!, System.Drawing.FontStyle.Bold)
```

```
Dim fontBody As Font = New Font("Microsoft Sans Serif", 8.0!)
```

```
Dim fontBodyBold As Font = New Font("Microsoft Sans Serif", 8.0!, System.Drawing.FontStyle.Bold)
```

' Some standard LineStyle objects for the BorderStyles

```
Dim lineNone As TMGDevelopment.Drawing.LineStyle = New TMGDevelopment.Drawing.LineStyle(Color.Empty,  
Drawing.Drawing2D.DashStyle.Solid, False, 1.0!, 0.0!, 0.0!)
```

```
Dim lineGray As TMGDevelopment.Drawing.LineStyle = New TMGDevelopment.Drawing.LineStyle(Color.Gray,  
Drawing.Drawing2D.DashStyle.Solid, True, 1.0!, 0.0!, 0.0!)
```

```
Dim lineBlack As TMGDevelopment.Drawing.LineStyle = New TMGDevelopment.Drawing.LineStyle(Color.Black,  
Drawing.Drawing2D.DashStyle.Solid, True, 1.0!, 0.0!, 0.0!)
```

' The BorderStyles for our header row, and special styles for the left hand column

```
Dim bsGrayBottom As TMGDevelopment.Drawing.BorderStyle = New TMGDevelopment.Drawing.BorderStyle(lineNone, lineGray,  
lineNone, lineNone)
```

```
Dim bsGrayBottomRHS As TMGDevelopment.Drawing.BorderStyle = New TMGDevelopment.Drawing.BorderStyle(lineNone, lineGray,  
lineNone, lineGray)
```

```
Dim bsGrayRHS As TMGDevelopment.Drawing.BorderStyle = New TMGDevelopment.Drawing.BorderStyle(lineNone, lineNone,  
lineNone, lineGray)
```

```
Dim bsBlack As TMGDevelopment.Drawing.BorderStyle = TMGDevelopment.Drawing.BorderStyle.BorderSingle()
```

' The default CellStyle for the header row

```
Dim csHeaderDefault As TMGDevelopment.PrintAdapters.CellStyle = New TMGDevelopment.PrintAdapters.CellStyle(fontHeader,  
Color.LightGray, Color.Black, bsGrayBottom)
```

' A CellStyle with a Wheat background for the table body row 1

```
Dim csWheatDefault As TMGDevelopment.PrintAdapters.CellStyle = New TMGDevelopment.PrintAdapters.CellStyle(fontBody,  
Color.Wheat, Color.Black, TMGDevelopment.Drawing.BorderStyle.BorderNone())
```

' A CellStyle with a OldLace background for the table body row 2

```
Dim csOldLaceDefault As TMGDevelopment.PrintAdapters.CellStyle = New TMGDevelopment.PrintAdapters.CellStyle(fontBody,  
Color.OldLace, Color.Black, TMGDevelopment.Drawing.BorderStyle.BorderNone())
```

' A special CellStyle for the header row

```
Dim csHeaderLHS As TMGDevelopment.PrintAdapters.CellStyle = New TMGDevelopment.PrintAdapters.CellStyle("csHeaderLHS",
fontHeader, Color.LightGray, Color.Black, bsGrayBottomRHS)
```

' A CellStyleMapping to associate the new CellStyle with the left hand column of the header row

```
Dim csmHeader As TMGDevelopment.PrintAdapters.CellStyleMapping = New TMGDevelopment.PrintAdapters.CellStyleMapping(0,
TMGDevelopment.PrintAdapters.ColumnIndexNumbering.FromLeft, "csHeaderLHS")
```

' A special CellStyle for the Wheat-colored row

```
Dim csWheatLHS As TMGDevelopment.PrintAdapters.CellStyle = New TMGDevelopment.PrintAdapters.CellStyle("csWheatLHS",
fontBody, Color.Wheat, Color.Black, bsGrayRHS)
```

' A CellStyleMapping to associate the new CellStyle with the left hand column of the Wheat-colored row

```
Dim csmWheat As TMGDevelopment.PrintAdapters.CellStyleMapping = New TMGDevelopment.PrintAdapters.CellStyleMapping(0,
TMGDevelopment.PrintAdapters.ColumnIndexNumbering.FromLeft, "csWheatLHS")
```

' A special CellStyle for the OldLace-colored row

```
Dim csOldLaceLHS As TMGDevelopment.PrintAdapters.CellStyle = New TMGDevelopment.PrintAdapters.CellStyle("csOldLaceLHS",
fontBody, Color.OldLace, Color.Black, bsGrayRHS)
```

' A CellStyleMapping to associate the new CellStyle with the left hand column of the OldLace-colored row

```
Dim csmOldLace As TMGDevelopment.PrintAdapters.CellStyleMapping = New TMGDevelopment.PrintAdapters.CellStyleMapping(0,
TMGDevelopment.PrintAdapters.ColumnIndexNumbering.FromLeft, "csOldLaceLHS")
```

' RowStyles for the 3 different types of row we're defining. Note each one takes the default CellStyle

' for the row, and then an array of CellStyles and an array of CellStyleMappings

```
Dim rsHeader As TMGDevelopment.PrintAdapters.RowStyle = New TMGDevelopment.PrintAdapters.RowStyle(csHeaderDefault,
Nothing, New TMGDevelopment.PrintAdapters.CellStyle() {csHeaderLHS}, New TMGDevelopment.PrintAdapters.CellStyleMapping()
{csmHeader})
```

```
Dim rsWheatBody As TMGDevelopment.PrintAdapters.RowStyle = New TMGDevelopment.PrintAdapters.RowStyle(csWheatDefault,
Nothing, New TMGDevelopment.PrintAdapters.CellStyle() {csWheatLHS}, New TMGDevelopment.PrintAdapters.CellStyleMapping()
{csmWheat})
```

```
Dim rsOldLaceBody As TMGDevelopment.PrintAdapters.RowStyle = New TMGDevelopment.PrintAdapters.RowStyle
(csOldLaceDefault, Nothing, New TMGDevelopment.PrintAdapters.CellStyle() {csOldLaceLHS}, New
TMGDevelopment.PrintAdapters.CellStyleMapping() {csmOldLace})
```

' Finally we construct the TableStyle giving it a name, and array of RowStyles, no column definitions,

' the outer BorderStyle and setting AutoDetectHeadings and SnapBordersToData true:

```
Me.PrintListView2.TableStyle() = New TMGDevelopment.PrintAdapters.TableStyle("Hard-coded style1", New
TMGDevelopment.PrintAdapters.RowStyle() {rsHeader, rsWheatBody, rsOldLaceBody}, Nothing, bsBlack, True, True)
```

```
Me.PrintPreviewDialog2.ShowDialog()
```

## Summary

We've shown how complex TableStyles can be built up in the designer, and also how to create the same effect through code. I hope this will give you an idea of the possibilities with the PrintAdapters library, particularly when you combine the style options above with events exposed by PrintListView such as the BeginRow event, which allows you to alter the RowStyle to be used by a given row before it is printed. With this you could adapt the display style to the data (such as showing all sales to a particular customer in a different color, or making the font bold when a value falls below a certain threshold etc). You can even combine the design-time definition of your RowStyles with runtime application of them in a small amount of custom code in the BeginRow event handler.

For more information about PrintAdapters, and other .NET solutions, please visit the WinformReports website at <http://www.winformreports.co.uk>

July 1, 2003

© 2002-2003 TMG Development Ltd