Listening to a Report

Doug Hennig

Microsoft has opened the architecture of the reporting engine in VFP 9 by having it communicate with the new ReportListener base class. Subclassing ReportListener allows VFP developers to create their own customized output. This month, Doug Hennig looks at ReportListener and shows an example of how it solves a real-world problem.

As I'm sure you're aware by now, the area that received the biggest improvements in VFP 9 is the reporting system. Both the Report Designer and the reporting engine, responsible for running reports, received dramatic and exciting enhancements and new features.

Before VFP 9, the reporting engine was monolithic: it handled everything–data handling, object positioning, rendering, previewing, and printing. The new reporting engine in VFP 9 splits responsibility for reporting between the reporting engine, which now just deals with data handling and object positioning, and a new VFP base class, ReportListener, which handles rendering and output. VFP 9 includes both the old reporting engine and the new one, so you can run reports under either engine as you see fit. Microsoft refers to the new reporting engine as "object-assisted" reporting.

Using object-assisted reporting

There are three ways you can tell VFP to use the new reporting engine:

- Instantiate a ReportListener (either a base class or a subclass) and specify it in the new OBJECT clause of a REPORT or LABEL command. This is the most flexible approach because you can specify exactly which listener class to use, but does require you to edit the existing REPORT and LABEL commands in your application.

```
loListener = createobject('MyReportListener')
report form MyReport object MyReportListener
```

- Specify a listener type using the OBJECT TYPE clause. There are several built-in types: 0 means print, 1 means preview, 4 means XML output, and 5 means HTML output. You can also define and use custom types.

```
report form MyReport object type 1 && preview
```

- Issue the new SET REPORTBEHAVIOR 90 command before running a report; usually, this will be placed near the start of your application so all reports use the new engine. Specifying TO PRINTER uses the built-in type 0 listener and PREVIEW uses the type 1 listener. This is obviously more convenient that the other approaches, but doesn't give you the level of control you get when instantiating your own listener. SET REPORTBEHAVIOR 80 to revert to the old reporting engine.

When you run a report using either of the last two methods, the application specified in the new _REPORTOUTPUT system variable (by default, ReportOutput.APP in the VFP home directory) is called to figure out which listener class to instantiate for the specified type. ReportOutput.APP is primarily an object factory; it simply instantiates the appropriate listener. However, because it's just a VFP application, you can substitute your own application for it by setting _REPORTOUTPUT accordingly. Be sure to distribute ReportOutput.APP (or your replacement for it) to your users so your applications use object-assisted reporting.

Inside ReportListener

Because ReportListener is a VFP base class, you can subclass it to implement whatever reporting behavior you wish. Before you can create your own listener class, you need to understand what properties, methods,

and events (PEMs) are available. For space reasons, I'll only discuss the more important PEMs; see the VFP help file for details on the complete set.

One of the most important properties is ListenerType. This property tells the report listener how to do output. This property defaults to -1, which produces no output. Set it to 0 to output to a printer or 1 to output to a preview window. Specifying 2 or 3 produces interesting results; the report is run and pages are rendered in memory, but nothing is actually output. You can use these values when you want control over the type of output to create. Specifying 2 renders the first page and calls the OutputPage method, then renders the next page and calls the OutputPage method, and so on. Using 3 causes all pages to be rendered to memory; OutputPage is not automatically called.

Before a report is actually run, the reporting engine opens a copy of the report as a read-only cursor named FRX in a private datasession. The ID for this datasession is stored in the FRXDataSession property of ReportListener. If you need access to the data being reported on, the CurrentDataSession property tells you which datasession to use.

The CommandClauses property contains a reference to an object containing properties with information about how the report is being run. For example, its Preview property is .T. if the report is being previewed and its OutputTo property is 1 if the report is being printed.

The reporting engine fires events of the report listener as the report is run. There are also some methods available you can call as necessary. Some of the more important events and methods are shown in **Table 1**.

Table 1. Some of the events and methods of ReportListener.

Event	Description
BeforeReport	Fired before the report is run
AfterReport	Fired after the report is run
EvaluateContents	Fired before a field is rendered
AdjustObjectSize	Fired before a picture or shape is rendered
Render	Fired as each object is rendered
OutputPage	Call this to output the specified page to the specified device
CancelReport	Call this to cancel the report

EvaluateContents, AdjustObjectSize, and Render are especially useful because they allow you to change something about the object before it's rendered. Amongst other parameters (we'll look at them later), these events receive the record number for the current object in the FRX cursor. You can find this record in the cursor to determine if the object should be rendered differently than normal.

_ReportListener

The FFC subdirectory of the VFP home directory contains a new class library in VFP 9: _ReportListener.VCX. This library contains several ReportListener subclasses. You may wish to consider using one of these, _ReportListener, as the starting point for your own ReportListener subclasses because it adds some very useful functionality to the base class.

One of the most useful enhancements is support for chaining different listeners together through a successor mechanism. Setting the Successor property of one listener to a reference to another one allows both of them to interact with the report process. This means you can write small listeners that do just one thing and hook them together as needed. The IsSuccessor property tells you whether this listener is the "lead" one (the one that the reporting engine communicates with because it's specified in the OBJECT clause of a REPORT or LABEL command).

_ReportListener also provides several utility methods. SetFRXDataSession switches to the FRX cursor's datasession. SetCurrentDataSession switches to the datasession the report's data is in. ResetDataSession restores the datasession ID to the one the listener is in.

Now that you have the background on report listeners, it's time for some practical examples.

Dynamic formatting

One of the first things I thought of using a listener for is to dynamically format a field. I'm sure you've run into this before: your client wants a field to be printed in red under some conditions and blank under others. You could do this in earlier versions of VFP by creating two copies of the same field, one in red and one in black, with mutually exclusive Print When conditions (such as AMOUNT >= 100 and AMOUNT < 100),

and overlap them on the report. While this works, it's tough to maintain, especially if you have a lot of such fields on the report.

With a report listener, you can change the formatting for a field when the report is run rather than in the Report Designer. The key to this is the EvaluateContents event, which fires just before each field is rendered. This event is passed the record number of the current object in the FRX cursor and a reference to an object containing properties with information about the field (see **Table 2**).

Table 2. Properties of the oObjProperties object passed to EvaluateContents.

Property	Type	Description
FillAlpha	N	The alpha, or transparency, of the fill color. The values range from 0 for transparent to 255 for opaque.
FillBlue	N	The blue portion of an RGB() value for the fill color.
FillGreen	N	The green portion of an RGB() value for the fill color.
FillRed	N	The red portion of an RGB() value for the fill color.
FontName	C	The font name.
FontSize	N	The font size.
FontStyle	N	A value representing the font style. Additive values of 1 (bold), 2 (italics), 4 (underlined), and 128 (strikethrough).
PenAlpha	N	The alpha of the pen color.
PenBlue	N	The blue portion of an RGB() value for the pen color.
PenGreen	N	The green portion of an RGB() value for the pen color.
PenRed	N	The red portion of an RGB() value for the pen color.
Reload	L	Set this to .T. to notify the report engine that you changed one or more of the other properties.
Text	C	The text to be output for the field object.
Value	varies	The actual value of the field to output.

DynamicFormatting.PRG, included with this month's Subscriber Downloads, defines three classes. DynamicListener defines what a dynamic listener must do, and two subclasses, DynamicForeColorListener and DynamicStyleListener, change the foreground color and style, respectively, of a field that has a directive in its USER memo. (You can access the USER memo for a field from the Other page of the Field Properties dialog.) The directive is one of the following:

```
*:LISTENER FORECOLOR = ColorExpression
*:LISTENER STYLE = StyleExpression
```

ColorExpression is an expression that evaluates to an RGB value, such as IIF(AMOUNT > 50, RGB(255, 0, 0), RGB (0, 0, 0)), which means use red if the amount is more than 50 and black if not. StyleExpression is an expression that evaluates to a valid style value (see the FontStyle property in Table 2), such as IIF(AMOUNT > 50, 1, 0), which means use bold if the amount is more than 50 and normal if not.

The first task the listener must do is identify which fields have directives. Rather than doing that every time a field is evaluated, DynamicListener does it in the BeforeReport method. It selects the FRX cursor's datasession by calling SetFRXDataSession, then goes through the cursor, looking for records with the appropriate directive (specified in the cDirective property) in the USER memo, and putting the expression following the directive into that record's element in an array.

The next task is to apply the directive as necessary. EvaluateContents checks if the current field's array element has an expression, and if so, evaluates it. It then calls the ApplyDirective method, which is abstract in DynamicListener but implemented in its two subclasses. For example, DynamicForeColorListener sets the appropriate color properties of the toObjProperties object and sets its Reload property to .T. so the reporting engine knows the field's format was changed. Finally, since the EvaluateContents method of the _ReportListener class this class is based on doesn't handle successors, the code calls the EvaluateContents method of its successor if there is one.

There's one other housekeeping chore: ensuring ListenerType is set properly. The default value, -1, produces no output, and specifying PREVIEW or TO PRINTER in the REPORT or LABEL command doesn't change that. So, the LoadReport method sets ListenerType to the appropriate value if necessary.

Here's the code for these classes:

```
define class DynamicListener as _ReportListener of ;
  home() + 'ffc\_ReportListener.vcx'
```

```
  dimension aRecords[1]
    && an array of information for each record in the
    && FRX
  cDirective = ''
    && the directive we're expecting to find

* If ListenerType hasn't already been set, set it
* based on whether the report is being printed or
* previewed.

  function LoadReport
    with This
      do case
        case .ListenerType <> -1
        case .CommandClauses.Preview
          .ListenerType = 1
        case .CommandClauses.OutputTo = 1
          .ListenerType = 0
      endcase
    endwith
    dodefault()
  endfunc

* Before we run the report, go through the FRX and
* store information about any field with our expected
* directive in its USER memo into the aRecords array.

  function BeforeReport
    dodefault()
    with This
      .SetFRXDataSession()
      dimension .aRecords[reccount()]
      scan for .cDirective $ USER
        .aRecords[recno()] = strextract(USER, ;
          .cDirective + ' =', chr(13), 1, 3)
      endscan for .cDirective $ USER
      .ResetDataSession()
    endwith
  endfunc

* If the field about to be rendered has a directive,
* apply it.

  function EvaluateContents(tnFRXRecno, toObjProperties)
    local lcExpression, ;
      luValue
    with This
      lcExpression = .aRecords[tnFRXRecno]
      if not empty(lcExpression)
        luValue = evaluate(lcExpression)
        .ApplyDirective(tnFRXRecno, ;
          toObjProperties, luValue)
      endif not empty(lcExpression)

* If we have a successor, let it get in on the fun
* too.

      if vartype(.Successor) = 'O'
        .Successor.EvaluateContents(tnFRXRecno, ;
          toObjProperties)
      endif vartype(.Successor) = 'O'
    endwith
  endfunc

* Abstract method to apply our directive.

  function ApplyDirective(tnFRXRecno, ;
    toObjProperties, tuValue)
  endfunc
enddefine
```

```
define class DynamicForeColorListener ;
  as DynamicListener
  cDirective = '*:LISTENER FORECOLOR'

* Apply the directive.

  function ApplyDirective(tnFRXRecno, ;
    toObjProperties, tuValue)
    local lnPenRed, ;
      lnPenGreen, ;
      lnPenBlue
    if vartype(tuValue) = 'N'
      lnPenRed   = bitand(tuValue, 0x0000FF)
      lnPenGreen = bitrshift(bitand(tuValue, ;
        0x00FF00),  8)
      lnPenBlue  = bitrshift(bitand(tuValue, ;
        0xFF0000), 16)
      with toObjProperties
        if .PenRed <> lnPenRed or ;
          .PenGreen <> lnPenGreen or ;
          .PenBlue <> lnPenBlue
          .PenRed   = lnPenRed
          .PenGreen = lnPenGreen
          .PenBlue  = lnPenBlue
          .Reload   = .T.
        endif .PenRed <> lnPenRed ...
      endwith
    endif vartype(tuValue) = 'N'
  endfunc
enddefine

define class DynamicStyleListener as DynamicListener
  cDirective = '*:LISTENER STYLE'

* Apply the directive.

  function ApplyDirective(tnFRXRecno, ;
    toObjProperties, tuValue)
    if vartype(tuValue) = 'N'
      toObjProperties.FontStyle = tuValue
      toObjProperties.Reload    = .T.
    endif vartype(lnStyle) = 'N'
  endfunc
enddefine
```

TestDynamicFormatting.PRG illustrates how to chain these listeners together so both are used for a report.

```
use _samples + 'Northwind\Orders'
loListener = newobject('DynamicForeColorListener', ;
  'DynamicFormatting.prg')
loListener.Successor = ;
  newobject('DynamicStyleListener', ;
  'DynamicFormatting.prg')
report form TestDynamicFormatting.FRX preview ;
  object loListener
```

Figure 1 show the result of running this program. Notice that in some records, the Shipped Date appears in red and in other cases, it's black. That's because it has the following directive in its USER memo:

```
*:LISTENER FORECOLOR = iif(SHIPPEDDATE > ORDERDATE +
10, rgb(255, 0, 0), rgb(0, 0, 0))
```

The Ship Via field sometimes appears in bold and sometimes normal because it has the following directive in its USER memo:

```
*:LISTENER STYLE = iif(SHIPVIA = 3, 1, 0)
```

(Note that although this field is numeric, it displays as "Fedex," "UPS," or "Mail" because of the expression in the field.)

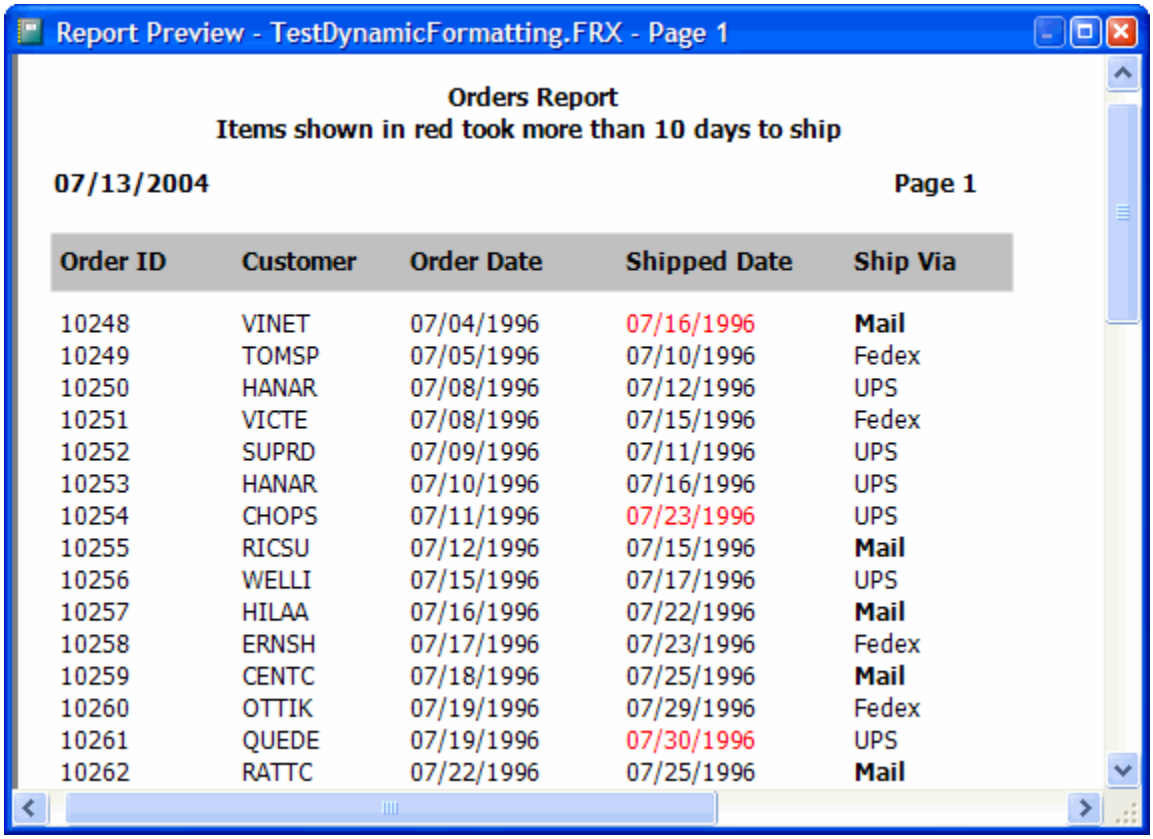

Figure 1. Report listeners can dynamically format the text in fields.

What else can you do?

Just about anything you want. In future articles, I'll show you other listeners that output to image files, rotate labels, output HTML with a table of contents, and lots of other types of output.

VFP guru Ed Leafe has created a Web site (http://reportlistener.com) that serves as a central repository for report listener classes. There are several sample listeners there now and more will be uploaded as VFP developers start figuring out what type of cool things they can do with report listeners.

Summary

Microsoft has blown the lid off extensibility in VFP 9 in many ways, including in the reporting engine. Because it's a base class you can subclass, ReportListener lets you create your own customized output. Please let me know of any cool listeners you've created or ideas you have for listeners.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT) and Stonefield Query, and the MemberData Editor, Anchor Editor, New Property/Method Dialog, and CursorAdapter and DataEnvironment builders that come with VFP. He is co-author of the "What's New in Visual FoxPro" series and "The Hacker's Guide to Visual FoxPro 7.0," all from Hentzenwerke Publishing. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a long-time Microsoft Most Valuable Professional (MVP), having first been honored with this award in 1996. Web: www.stonefield.com and www.stonefieldquery.com Email: dhennig@stonefield.com