Making invoices with grids

Rafael Copquin, Estudio Copquin

Public Accountant and Certified Internal Auditor; Fox programmer since 1987, starting with FPD and upgrading to VFP. Vast experience in conversions from FPD to VFP, specialized in business and accounting systems. Translator of VFP articles and manuals (English/Spanish).Coordinator of the English - Spanish translation team of the UTMag. Member of the Microsoft Users Group of Argentina's Board of Directors as Treasurer.

Background

Early in my development experience, back in the days of Fox Dos, I had to develop a better way to make invoices than the one I had been using.

What I had been doing was using the SCROLL statement to make the screen move upwards every time a new item was added.

Whenever the user wanted to review the invoice, especially if it contained a large number of lines that would disappear from the screen's limited drawing area, I had to create an array dynamically from the records saved in a local table, and show them with a POPUP.

If any changes had to be made (delete a line, change an item code, a quantity or a unit price), I would select that item from the popup, call a change routine, make all necessary changes, redraw the screen.... What a job!!!

If my client ever wanted to make any changes, no matter how minor, it would take many hours because of how involved the whole construct was.

Then, in search of a better way, I started using BROWSE. That did it. The browse presented such an ease of manipulation, it was so flexible, that I discarded all previous code and switched to browse screens.

Now, in the OOP era, all that is history. The browse can very successfully be replaced by a grid, if you are careful to consider all aspects of its use.

What Is Involved In Making Invoices

I can mention the following items to be considered when making invoices, especially if the application, as most these days, is to be operating in a network environment:

- Use a local cursor to enter and modify invoice data
- After the invoice is accepted and ready to be printed, consider using transactions to save the data in the different tables
- Use a common shared table to produce invoice numbers

Why do I prefer to use a local cursor to enter data? If we use a local cursor, we can add, modify or delete records as needed, without compromising server data in any way. In addition, we do not utilize network resources needlessly nor are we faced with the "gone to lunch" syndrome resulting in locking records indefinitely.

Why use transactions? It depends. If the system is going to be used in a standalone machine, perhaps using transactions is not that necessary. However, a potential power cut or some other program stalling the machine might make it necessary. In a networked system, on the contrary, I think it would be advisable.

When an invoice is issued, many tables intervene in the process: lookup tables such as the customers, items and stock tables, for customer data, item prices and descriptions and on hand quantities, service tables, such as the invoice numbering table or in some countries tax rates tables, and also "save" tables, such as the stock, invoice totals, invoice details, accounts receivable, daily cash tables, and others as the case may be.

As many tables are involved, especially in the saving process, and because saving is done sequentially, table by table, record by record, somewhere along the line, something might fail (Murphy's Law at its best): a power cut, a faulty network card, a corrupt index in one of the tables or a corrupt table header. In those instances, some of the tables get saved, while others are not saved. The result is a nightmare to fix. So I would recommend the use of transactions, cautioning the programmer to make them as efficient as possible, so as not to lock the tables or records for too long a time.

And use a shared table to obtain invoice numbers. This is necessary, because when two or more users are making invoices, you never know who is going to press the OK button first. By using a shared table that controls the invoice numbering sequence, you can very efficiently issue hundreds of invoices every day, and not mix up the saved data.

There are other issues to be considered, such as stuck invoice pre-printed forms in some printers, different forms of payment for the same invoice (coupons, credit cards, cash, checks, etc.), global or individual discounts, different tax rates for different types of customers, as may be the case in certain countries, etc.

In this article, however, I am only going to address the specific issue of using a grid to make the invoice. And I will make it simple, no line discounts and no controls added to the grids columns. Just a simple grid, as comes from "the shelf".

The Form

There are many options for the type of form to use. In some cases, we could use a form to define the customer data (name, address, tax category in some countries, whether the sale is on credit or a cash sale, etc).

You could enter all the necessary data in this form and then call another form with the grid on it to make the invoice.

Or you could use a single form with a page frame. In one of the pages you would get the customer data needed and in the other one you could have your data entry grid.

Or you could just have a single form with enough space in it for the customer data and the data entry grid.

The choice is yours.

In addition to the grid, you need a few labels to show the invoice totals, which will be updated and refreshed every time you make a change in the grid, such as: adding a new record, modifying a price or a quantity, or just delete an unwanted line.

If you enter a global discount, such as for example 10% of the total amount, you would need a text box to enter the discount percentage and appropriate labels to show the discount amount.

In any case, whenever you make any of the above changes, you would call the INV_TOTALS() method, which will do all the calculations, as will be shown below.

The Main Cursor And Its Structure

As mentioned above, use a local cursor for data entry. A local cursor is read write, of course.

The structure should be as follows:

FIELD	TYPE	OBSERVATIONS
ItemCode	C(10)	or whatever length your item code has
Detail	C (50)	or larger, depending on item description
Quantity	N (5)	
UnitPrice	N (12,2)	
LineTotal	N(12,2)	

You can create this local cursor in the LOAD event of the form to use with the following command:

```
Create cursor INVOICE( ;
 ItemCode C(10), Detail C(50), Quantity N(5,0),;
 UnitPrice N(12,2),LineTotal N(12,2) )
Append blank
```

(we need to have at least one record to enter data into the cursor)

The Tables

You can use the DATAENVIRONMENT to load all necessary tables. I will only mention the tables needed for my example.

Their structure, indexes and relationships are not an issue here.

Since this is a cash over the counter sale, with no need to identify the customer, I am going to need the following tables only:

- ITEMS.DBF
  containing the item data (code, description, price)
- STOCK.DBF
  to update the stock data
- INVTOTAL.DBF
  to save one record per invoice with the invoice date, invoice number, sales taxes applicable, salesperson number, global discount and invoice total
- DETAILS.DBF
  for the quantities, unit prices and item id, for each invoice issued
- CASH.DBF
  for the invoice number, invoice date, partial (or total) payment and type of payment (cash, check, coupon, credit card)
- CREDCARD.DBF
  if we will be accepting credit cards for payment, we need a lookup table with all possible credit cards

Depending on your own particular situation, you might need other tables. But, as a minimum, I think the above list will suffice.

The Grid

Following the INVOICE cursor structure, we need five columns in our grid. In order to let the form instantiate itself, our grid will just sit in it, with no recordsource defined or any of its properties set, except perhaps for the grid's width, columncount and the column widths and header captions. This is only so when we open the form for modification purposes, we can have a quick look at it and see how it will show at run time. But we have a choice of just "dumping" a grid on the form and set its properties by calling the SET_GRID method from the form INIT as follows:

Init Method Code

(we would initialize form properties as needed here, before initializing the grid)

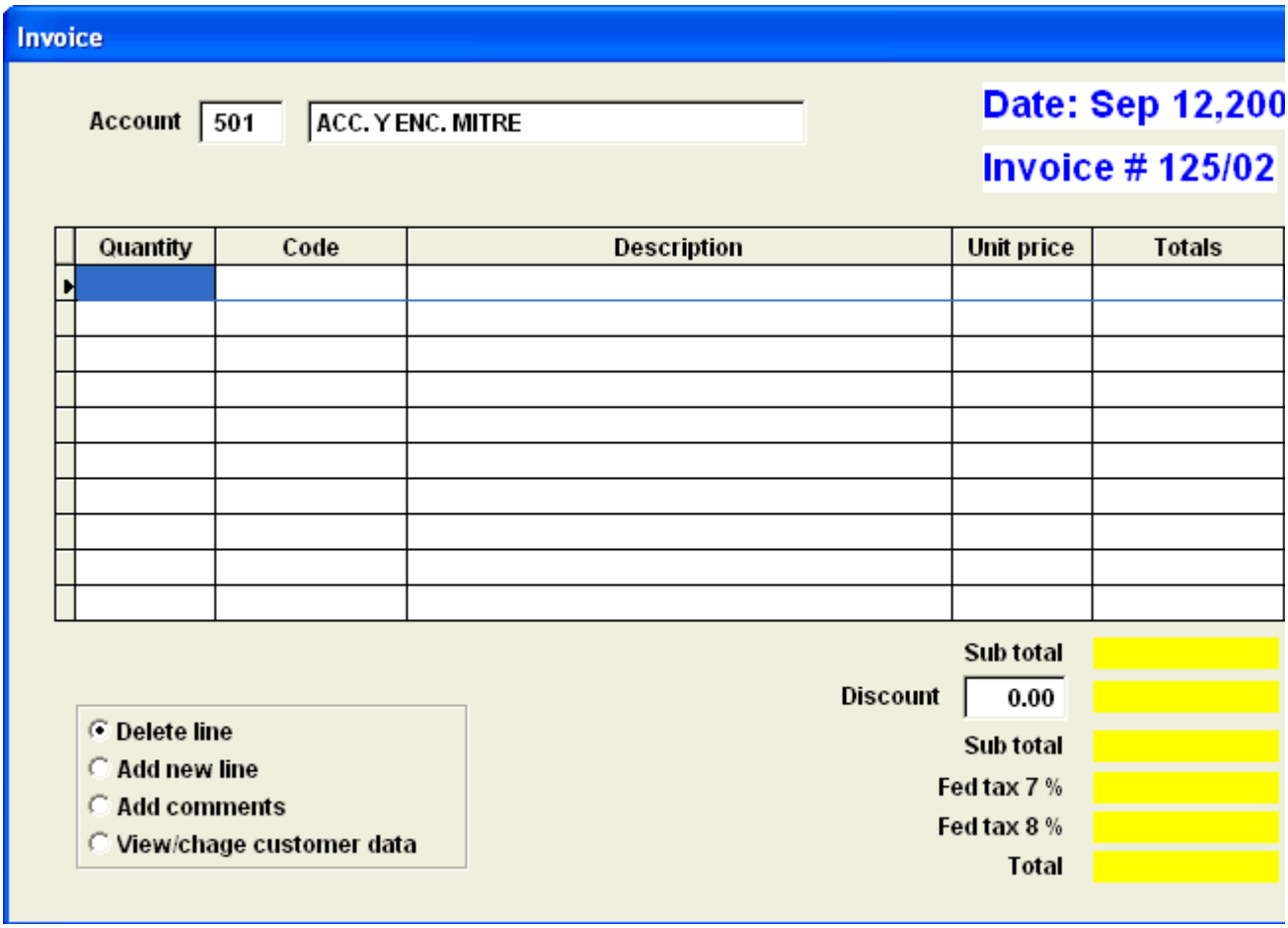
```
this.nPrice = 0.00
this.cDescrip = ""
this.nMaxLinesAllowed = 30   && maximum number of lines that fit
                             && in the preprinted invoice form
this.cDocType = "FC"

this.nFedTax  = 7.00   && this values can be obtained from
this.nProvTax = 8.00   && a special sales tax table
this.nPercent = 0.00

this.lblFTpctg.caption = "Fed tax "+ alltrim(transform( this.nFedTax ))+" %"
this.lblPTpctg.caption = "Prov tax "+ alltrim(transform( this.nProvTax ))+" %"

this.set_grid()
```

The Set_grid Method

As this is intended to be a simple example I will use the default name for the grid:

```
With thisform.grid1
   .fontbold = .t.
   .readonly = .f.
   .columncount = 5
```

```
   .recordsource = "invoice"
   .allowaddnew = .f.  && this is explained further down
   .columncount = 5
   .deletemark = .f.
   .scrollbars = 2
   .width = 642

   with .column1
      .controlsource="invoice.quantity"
      .width = 70
      with .header1
         .caption = "Quantity"
         .alignment=2             && centered
      endwith
   endwith

   with .column2
      .width = 96
      .controlsource="invoice.itemcode"
      with .header1
         .caption = "Code"
         .alignment=2
      endwith
   endwith

   with .column3
      .controlsource="invoice.detail"
      .width = 275
      with .header1
         .caption = "Description"
         .alignment=2             && centered
      endwith
   endwith

   with .column4
      .controlsource="invoice.unitprice"
      .width = 70
      with .header1
         .caption = "Unit price"
         .alignment=2             && centered
      endwith
   endwith

   with .column5
      .controlsource="invoice.linetotal"
      .width = 96
      with .header1
         .caption = "Totals"
         .alignment=2             && centered
      endwith
   endwith
Endwith
```



Additional considerations

Following the natural logic of making an invoice, we have established the grid layout in the above manner, because we simply answered the question: how is an invoice made ?

The answer is very straight forward: first enter the quantities, then we enter the item code, at this point the system would check the items table and fetch the item description and its price, put the information in the grid line we are modifying, then a multiplication of the unit price times the quantity would take place and, finally, the line total would be calculated. The process ends when a new empty line is added into the grid and the cursor is placed in the first column, to accept a new item.

How do we make it all work?

Let us take a look at the Quantities column.

We would like the user to enter quantities greater than zero. In this example, we only want integer values. So we would have to validate the input, not enabling quantities with decimals or zeroes. However, we should allow the user to enter negative quantities, in case the form is used to issue credit notes, or make the form "smart", turning all positive quantities into negatives in case we are dealing with credit notes.

We will therefore put the following code in the INIT method of the textbox in column1:

```
This.value     = 0
This.InputMask = "99999"
This.MaxLength = 5
This.Format    = "Z"
```

The above code ensures that only numbers are entered, that the initial value is zero, that no number is shown if nothing is entered and that the maximum amount is not over 99999.

But the user could enter a zero. So we have to provide a means to either warn the user that a zero value is not acceptable, or transform a zero into a 1.

In my experience, it is faster to let the user just press the enter key when she is on this column, which will cause a zero value to be entered. But with the following code in the VALID event of the textbox, the zero is automatically transformed into a 1, leaving the quantity column and jumping into the next one.

```
this.value = iif(this.value = 0,1,this.value)

this.refresh
```

Assuming we had a form property called cDocType, we could also use the VALID method to make the quantity negative, thus enabling the user to just enter any number and not have to worry about making it negative.

```
this.value = iif(this.value = 0,1,this.value)

if thisform.cDocType = 'CN'        && CN would mean credit note

   if abs(this.value) > 0

      this.value = -this.value

   endif

endif

thisform.inv_totals()  && this method is explained later on
```

(the value of the form's cDocType property would be determined by the user, prior to starting data entry on this form. But showing how to do this is not within the scope of this article)

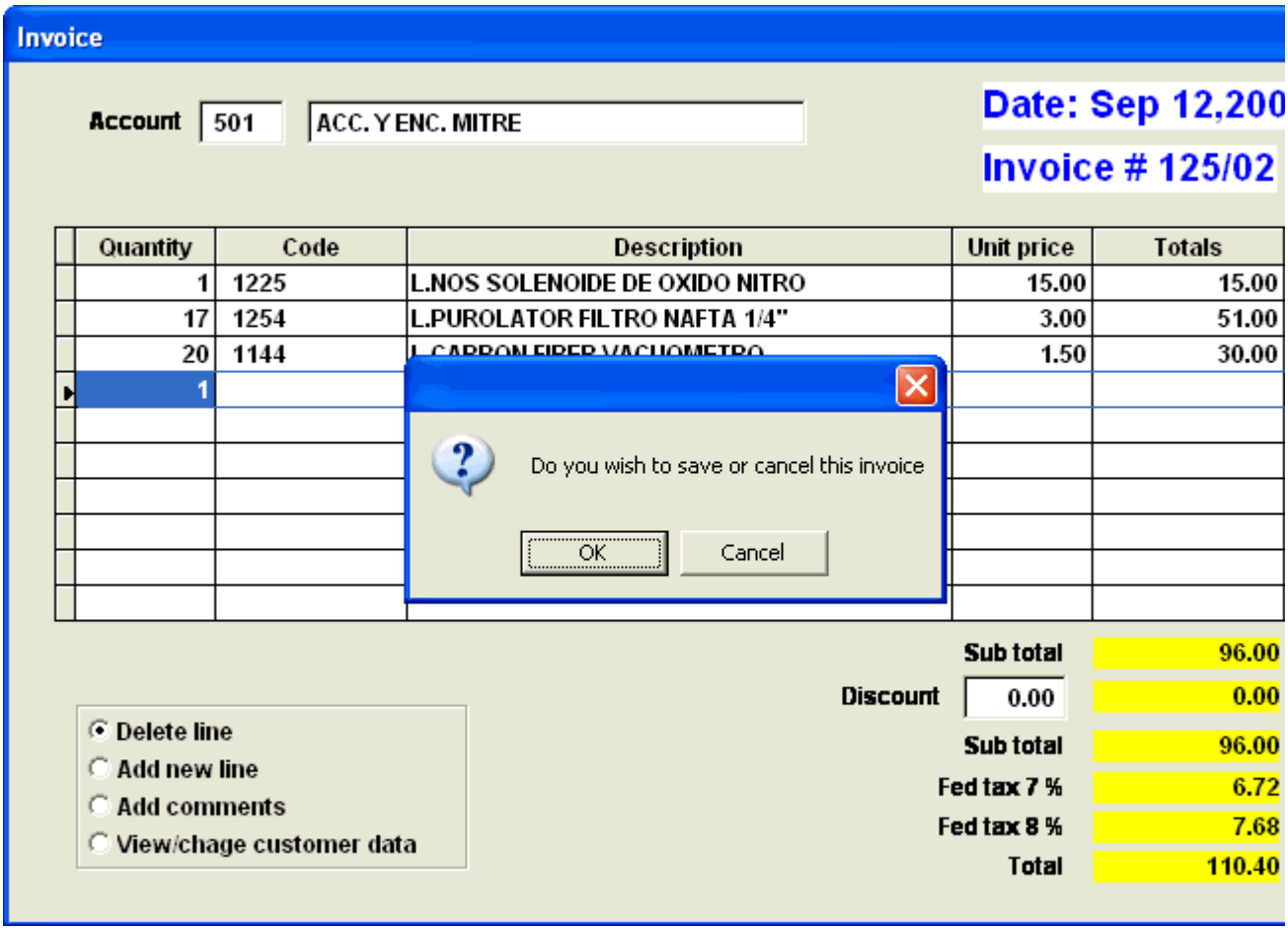
We need to ensure a way to exit the grid, in order to either discard the invoice or save and print it. So we program the KEYPRESS with this:

Keypress method code

```
LPARAMETERS nKeyCode, nShiftAltCtrl

If nKeyCode = 27     && escape key pressed

   Thisform.WhatNow()

Endif
```

We could place code in the WhatNow method asking the user whether he wants to discard the invoice or print it.

The above code will lead us there.



The item code column

The cursor is now on the item code column.

Depending on the item code characteristics, we would use a different validation routine.

For instance, we could have an alphanumerical code such as: DURACELL AA, PKJ/90-85, BAYER ASPIRINE or any other.

We could also have numeric codes, a combination of numbers, hyphens and slashes, etc.

The list is numerous and I can't show in here all the possibilities.

But I said earlier that I want to keep things simple, because the objective is to show how to make an invoice with a grid, not how to validate input.

Therefore I am going to use a simple numerical code, with a length of no more than six digits, stored in a character field.

In the INIT method of this column's textbox we write the following code:

```
This.value     = ""
This.InputMask = "999999"
This.MaxLength = 6
```

This code ensures that only numbers will be allowed, and no more than 6 digits long.

In the VALID method, the code would be:

```
Local cCode
cCode = str(val(this.value),6)
```

```
select items
if seek( cCode , "items", "codeid")

   this.value = cCode

   with thisform
      .nPrice   = items.pesos
      .cDescrip = items.descrip
   endwith

   select invoice

   Replace invoice.detail with thisform.cDescrip ,;
           invoice.unitprice with thisform.nPrice   ,;
           invoice.linetotal with (invoice.quantity * invoice.unitprice)

   thisform.inv_totals()

   return 1
else

   this.value = ""
   messagebox("INVALID CODE",48,"ATTENTION")

   return 0
endif
```

OK, why do we have to replace the fields in the underlying cursor with the obtained values? It could be argued that, because the textboxes controlsources are bound to the cursor's fields, this is not necessary. True. But sometimes, a little redundancy helps. In this case, I make absolutely sure that, in spite of Murphy's Law, the data just fetched from the ITEMS table, that was placed in the form's corresponding properties by the valid method code, gets in fact stored in the record.

However, I did not do this with the quantities entered. Why? I was acting directly on the cursor by entering a value into the quantity field, but in the case of the values fetched from the ITEMS table, I was dealing with another area. I am a practical guy. I found this to work. So, a little redundancy does not hurt.

We are now ready to enter data into the details column but, wait a minute, the item description was fetched from the ITEMS table, its value stored in the INVOICE cursor and because this field is bound to the grid, the item description is already shown in our grid. Therefore, we do not want to enable the user to enter this column. How can we do that?

Very simple: put this code in the WHEN method of the details textbox:

```
Return .f.
```

As easy as that!

The when method fires before the valid method, even before you can get into this column. By returning false, you simply skip this column and are taken into the next one, the unit price column.

Even though we have the price in the ITEMS table, and obtained it from there, it is possible that the price list was not updated before making the invoice, and that we have to change the price right there and then.

So we enable entry into this column and the user can change the price.

But we can have a potential problem here. Because one of the reasons for using a grid was to be able to navigate up and down, left and right the different lines and columns and change the values, if we change the unit price, it will be accepted and, as shown further down, the invoice extension calculation will be performed. However, if the user presses the enter key on the item code column, the valid event on that column's textbox will fire, the item price will be fetched from the items table, and replace whatever other value the user enters in the price column.

This is an annoying behavior and it will make our invoice making routine completely useless.

So, how do we solve it?

I pondered over this for hours, or rather I should say days. Remember that when you program, your time is 10% inspiration and 90% sweat. At least, I am that kind of programmer.

So somewhere in my 10% inspiration time I had an idea: use a textbox property to enable or disable a price change when the user hits the enter key on the item code column, or using the arrow keys, she navigates to it.

So I made the following changes into the item column:

Init event

```
This.value     = ""
This.InputMask = "999999"
This.MaxLength = 6

This.addproperty(cOldCode,"")
```

When event

```
This.cOldCode = str(val(invoice.itemcode),6)
return .t.
```

Before entering the item code column, we read the value stored in the underlying cursor

Valid event

Change the code as follows:

```
Local cCode
cCode = str(val(this.value),6)

If cCode = this.cOldCode   && we've been here before

   Return 1         && so we do nothing, just get out
Else
    select items
    if seek( cCode , "items", "codeid")

       this.value = cCode


       with thisform
         .nPrice   = items.pesos
         .cDescrip = items.descrip
       endwith

       select invoice
       Replace invoice.detail with thisform.cDescrip ,;
               invoice.unitprice with thisform.nPrice   ,;
               invoice.linetotal with (invoice.quantity * invoice.unitprice)

       thisform.inv_totals()

      return 1

   else

      this.value = ""
      messagebox("INVALID CODE",48,"ATTENTION")
      return 0
   endif
```

```
endif
```

(The call to the inv_totals() method is made to ensure invoice totals are recalculated every time we change the item, from a previously entered one. See further down for an explanation of this method.)

The unit price column

Even though the unit price is in the ITEMS table, it is possible that it is not up to date, or that the user wants to use a different unit price for a particular invoice.

For that reason, we allow entry of another price into this column, with the following code in the INIT event.

```
this.addproperty("nLastPrice",0.00)
this.value = 0.00
```

Why do we initialize here the nLastPrice property? For the same reason that when we were navigating in any of the four directions, if we pressed ENTER in the items column, the VALID event would fire and brought forward the table price, instead of respecting the price we just entered in the unit price column. We have to make sure this does not happen. For that reason, before we enter this column, we read the previous value, by using the following code in the WHEN event:

```
this.nLastPrice = this.value
return .t.
```

and input is validated in the VALID event with:

```
if not inlist( lastkey() , 19 , 4 , 5 , 24 )  && left,right,up,down

   if this.value <> this.nLastPrice

      Replace invoice.unitprice with this.value   ,;
         invoice.linetotal with (invoice.quantity * invoice.unitprice)
   endif
endif

thisform.inv_totals()
```

What we are doing here is ultra - redundant: we first make sure that, if any of the direction arrows are pressed, only the totals at the bottom of the invoice are calculated. But, even if none of the arrow keys was pressed, only the price in the underlying cursor is changed and the extended total for that record is calculated, but only if the value we have in that column is different from the one that was there before we entered it.

In this manner, the problem mentioned is avoided. The price will not be changed inadvertently.

The rightmost column

The totals column textbox should be read-only, because it will just show the value calculated by the inv_totals method.

However, we should allow entry into this column, so the user presses the enter key. Doing this, she will cause a new record to be added to the INVOICE cursor, and focus will be in the first column, in preparation for a new entry.

How is this accomplished?

When we established the grid's properties, the AllowAddNew property was set to .f.

I fiddled with this property for some time, but soon discarded it because, in my opinion, it is useless in an invoicing environment.

This property, when set to true, causes a new record to be added when the user presses the down arrow key. Because we navigate the grid in all four directions, we would be adding unwanted records every time we went down. Therefore, as far as we are concerned, this property remains false.

We use the KEYPRESS method once again to add a new record and establish focus on the first column of the next line, with the following code:

```
LPARAMETERS nKeyCode, nShiftAltCtrl

local N

if nKeyCode = 13   && enter key pressed

   replace invoice.linetotal with (invoice.quantity * invoice.unitprice)
   go top in invoice
   count to N for not deleted()

   if N < thisform.nMaxLinesAllowed

      go bottom in invoice

      if not empty( invoice.itemcode )
          append blank
          go bottom in invoice
          keyboard '{dnarrow}'
      endif
   else
       delete next 1

       messagebox( 'MAXIMUM ALLOWED LINES'+CHR(13)+ 'FOR THE INVOICE FORM ARE: '+
           str(thisform.nMaxLinesAllowed),16,'ERROR')
   endif
endif

This.Parent.Parent.refresh      && grid is refreshed
```

Let us analyze the above code in detail:

When the user presses the enter key, this routine counts the number of non deleted records in the cursor and compares it to the maximum allowed lines an invoice can have. This is important if we use preprinted invoice forms, that have a limited space for invoice details. If the maximum allowed is exceeded, an error message warns the user and the newly added record is erased.

If the limit was not reached, a new record is added to the invoice cursor. We force focus on the last line of the grid with the combination of the GO BOTTOM and KEYBOARD '{DNARROW}' commands.

Notice the additional test: if there is no item code in the first column, then a new record is added, otherwise, nothing happens. This prevents useless empty lines to be added to the grid.

The inv_totals method

Every time we add a new line, or erase an existing one, change a price, replace an item by another or change quantities, our invoice bottom line will change.

The bottom line, or bottom lines if you will, is made up of the invoice subtotal (the sum total obtained by multiplying quantities times unit prices in every line), and any applicable taxes, global discounts or additional items (freight, interest, whatever). The very last line should be the total the customer has to pay.

Now, depending on how much room you have on your screen, all these values could be placed in just one line at the bottom or in a columnar way, one on top of the other. It is really the programmer's choice, and will not affect the calculations in any way.

The results of the calculations will be shown in either labels or textboxes, at the programmer's will.

In my own case I prefer to use labels. For example, if the invoice subtotal were 2,000.00 dollars and the label to display it would have the very imaginative name of lblSubTotal, we would show it with the following command line:

```
Thisform.lblSubTotal.caption = transform(thisform.nSubTotal,"9,999.99")
```

To have all totals in this labels align right, you should make the labels alignment property = 1

Having said the above, let us now see how the INV_TOTALS method operates

```
With thisform

   Store 0.00 to .nSubTotal,.nTotal,.nTax1,.nTax2,.nST2 , .nDiscount

   Select invoice
   Go top

   sum (invoice.quantity * invoice.unitprice) to .nSubTotal for not deleted()

   .nDiscount = .nSubTotal * -.nPercent / 100
   .nST2      = .nSubTotal + .nDiscount
   .nTax1  = .nST2 * .nFedTax / 100
   .nTax2  = .nST2 * .nProvTax / 100
   .nTotal = .nST2 + .nTax1 + .nTax2

   .show_labels()
   .refresh
endwith
```

Assuming we had a global discount property called nDiscount and a global discount percentage property called nPercent and that the user entered a 5 % global discount, we would now calculate the discount amount in this manner:

```
Thisform.nDiscount = thisform.nSubTotal* -thisform.nPercent / 100
```

(The nPercent form property is the controlsource of the textbox used to enter the desired 5% discount)

The above would render a negative discount value. Then we recalculate the invoice subtotal, to show the discounted value like so:

```
Thisform.nST2 = thisform.nSubTotal + thisform.nDiscount
```

(Note: we have to subtract the global discount from the subtotal, but since it is already negative, we use the add sign).

And now we are ready to calculate whatever taxes apply in your country.

Let us assume that you have a Federal Sales Tax of 7% and a Provincial Sales Tax of 8%. Then you would need to apply these percentages to the net subtotal as above.

```
Thisform.nTax1 = Thisform.nST2 * thisform.nFedTax / 100
Thisform.nTax2 = thisform.nST2 * thisform.nProvTax / 100
```

Finally, we can show the invoice total, by adding the net subtotal plus both tax amounts:

```
Thisform.nTotal = thisform.nST2 + thisform.Tax1 + thisform.Tax2
```

And we complete the method with a call to the method that will show all these calculated values in their respective labels at the bottom of the invoice.

```
Thisform.show_labels()
```

The show_labels method

As anticipated in the above section, we would use a series of TRANSFORM functions to place the calculated values in

their respective labels at the bottom of the invoice form.

```
local cPic
cPic = "99,999.99"

with thisform

   .lblST1.caption    =  transform( .nSubTotal , cPic )
   .lblDisc.caption   =  transform( .nDiscount , cPic )
   .lblST2.caption    =  transform( .nST2      , cPic )
   .lblFT.caption     =  transform( .nTax1     , cPic )
   .lblPT.caption     =  transform( .nTax2     , cPic )
   .lblTotal.caption  =  transform( .nTotal    , cPic )
endwith
```

The save method

I am not going to show here how to save our invoice using transactions, because it would take a lot of space and, again, it would go beyond the scope of this article. However, saving consists in using a scan - endscan loop on the INVOICE cursor, to save all its lines to the appropriate tables.

Conclusion

I have shown the way to make invoices and other "complicated" data entry screens, using grids.

In my real world applications I add a lot more functionality than just the methodology shown: call a lookup table for searching items, by pressing a function key when focus is on the item code column (you do this by programming the keypress event of the textbox), or show the customer account if the sale is on credit, by right clicking on the form and calling the accounts form, or enter a new customer while invoicing, or consulting a discount table, or showing a picture of the item being invoiced, etc, etc.

Grids provide us with the ability to greatly enhance our data entry screens. With a little bit of work and a lot of faith, we can do it. So good luck in your quest.