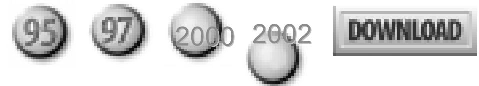
Excel: Data Entry for Access?

Scott McManus



For your users who feel more comfortable inputting data in a spreadsheet environment, Scott McManus shows you how to take an Excel spreadsheet and turn it into a data entry form for Access—complete with data verification and error checking. By combining Automation, Excel templates, and Excel's data validation option, you can have the same control over users' input as you would in Access using lookup tables, relationships, and rules.

GIVEN the option of using Excel as their data input interface of choice, I can hear a lot of Access developers respond with, "Why would you?" A well-designed Microsoft Access user interface provides data verification through three mechanisms: validation rules, lookup tables, and relationships. These are very difficult to replicate in Excel.

I used to agree with that point of view, but recently I was involved in four projects where the staff entering the data into databases weren't dedicated data entry people. Instead, they were professional people who needed to enter a small amount of data each day. Even though the four projects were in different industries, all of the staff in these projects were familiar with spreadsheets and, in fact, used them daily in many complex and varied ways to assist in scientific research or financial applications.

Unfortunately, the lovingly made forms of my Access user interface were rejected by the staff. In all cases they wanted to use the familiar copy, paste, and fill functionality of Excel to input their data. They had great trouble entering data row by row. They preferred to enter data in rows or in part of a column as the mood, or brain process, struck them. As these users were also senior management or technical professionals, we had to give the users the interface they wanted. But we also needed to ensure that the data was valid and verified, without taking too much valuable time from our "power users."

This led us on to two separate problem areas:

- How to import the spreadsheet data into Access.
- How to make sure the data is valid and verified.

Our final answer incorporated solutions in both areas into a single design.

Importing data

With whatever method that you choose to import spreadsheet data into Microsoft Access, a carefully designed Excel template will make life easier for you and support a more "automatic" process. Without a template, you allow a free-flowing kind of data input that will require more complex testing to cover all eventualities. In most cases, the end result will be that you'll have to import the data manually. For your users, a template provides a safe and friendly data entry environment. You need to take the time to design the spreadsheet user interface and have it accepted by your users before continuing on with the import and validation processes.

Import into Access manually

A common, but messy, way to import data is to use the Access tools to import the Excel sheet as a table. The manual import gives you two choices: Either import the spreadsheet into an existing table after deleting previous records (only possible if the spreadsheet's structure and column names don't change), or import into a temporary table and then use an update/append query to transfer the data (required if the column names do change between imports).

The process for the manual import is:

1. Select File | Get External Data | Import.
2. Select the spreadsheet file using the File Open dialog.
3. Select the worksheet or named range (see Figure 1).
4. Select whether the first row contains field names.
5. Either import into a new table or import into an existing table where the Field Names are *exactly* the same.
6. If you imported into a new table, run some code to copy the records from the temporary table to its final destination.

I have several reservations about this method. The first is that constant importing and deleting into temporary tables will cause the database to bloat, and force more frequent database compacts. The second problem is determining the table name to use in your code. If it's a temporary table name, you have to ensure

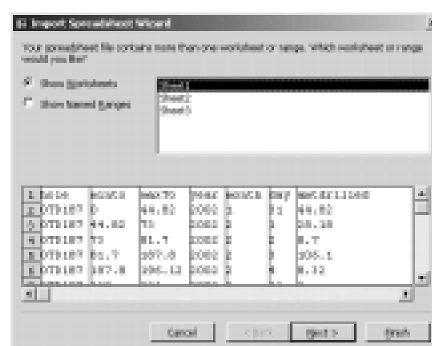


Figure 1. The familiar Access import wizard, importing an Excel spreadsheet. You may select a named range within a worksheet or a complete worksheet.

that the name of the new import table is passed to the routine that merges or appends the data to the final table or tables.

The third problem is that the Access import tool assigns the data type (integer, number, text, date, and so forth) based on the first non-header row that contains data in the worksheet. Figure 2 shows a situation where an Integer field in Excel has been assigned the Double Data Type. This can cause problems with mixed data type fields (such as columns where there can be numbers, alpha characters, or a combination of both). If the numeric data is in the first row, then the Access import tool will set the field type to numeric. When a row with alpha characters is found, errors occur and your import fails. Unfortunately, the import wizard doesn't let you change the auto selected data type. To get around this problem, you need to export your data (another step) to a text-based format such as *.CSV or *.TXT.

Finally, unless you use ADO or DAO in the final step that moves the data from the temporary table, you give up control of the data validation/verification process.

Linking to Excel

In this method you also use the External Data Wizard, but this time you link to the spreadsheet the same way that you'd link a software/application MDB to an MDB containing data. The process is:

1. Select File | Get External Data | Link Tables.
2. Select the Excel spreadsheet using the File Open dialog.
3. Select the worksheet or named range.
4. Let the Wizard know if the first row contains the header.
5. Provide a name for the new linked table (see Figure 3).

A linked spreadsheet can be treated like any other Access table—as long as you have all of the data in the spreadsheet in rows. You can view the data and include it in both select and update queries. You can use ADO or DAO with SQL in code to update the data in the linked table.

Problems with this method are few. The main problem is that if you re-link using a different table name each time, you have to keep on creating queries (or modifying them) and modifying your code to refer to the latest name. A secondary problem is that, as with the Manual Import Wizard, the data type of the fields is automatically set by Access.

One last thing that you should look out for when using this method is if the same sheet is continually used for data input. When do you know to run the import routine? How can you be sure earlier imports have succeeded and have or have not been processed successfully? As with the manual import, you'll need to use code to transfer data to its final destination if you want to maintain control over your data validation.

Automating Excel

This is by far my preferred method for getting data into Access because it solves the problem of varying Excel names. I also like it because *everything* is code-driven, so errors in the data can be trapped and dealt with inside routines at import time. The method requires that you use a template and that information be entered into the Excel spreadsheet exactly the same way each time. The previous methods, because they aren't fully automatic, allow for human intervention when columns aren't consistent.

Because this method relies on templates—rather than an exclusive row-by-row access—lookup tables, user messages, and even forms and controls can be used in the template to enhance verification and validation. Controls (like buttons and dropdown lists) may exist in the same row that data does—something that you can't do if you use the Import Wizard to import or link an Excel sheet.

Use the common dialog control to get the location and name of the Excel file that you want to import. You can use an API call, the Office FileOpenDialog box, or the Windows FileOpenDialog box. If you're programming in Visual Basic you can use the Common Dialog Control (which is also part of the toolbox in Visual Studio .NET!).

Other articles have dealt with using the File Open dialogs, so I won't go into that here. However, I will go over instantiating the Excel objects. The first thing that you must do is open a code window (code-behind file or a module) and check off a reference to the Excel object library in the Tools | References list. Figure 4 shows the reference selection box dialog.

With Excel referenced from your project, the next step is to write the code to instantiate an Excel object and open

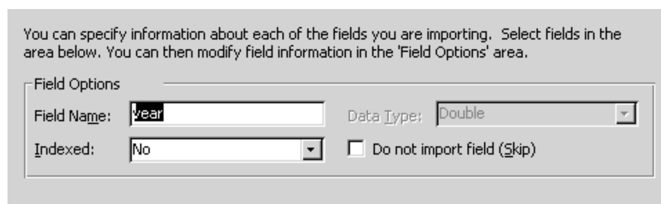


Figure 2. During an Excel import, the Import Wizard has decided to make an Integer Field in Excel a Double data type in Access. You can see that the Data Type combo box has been disabled.

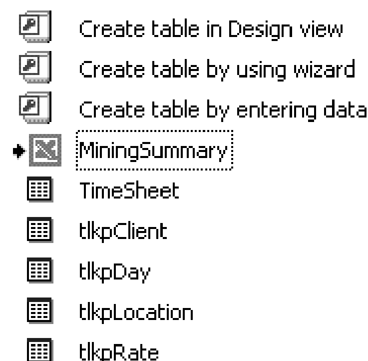


Figure 3. An example of a linked spreadsheet after using the Import Wizard.

the worksheet that you want to process. You can then determine how big the range is that you want to process, assuming that you haven't predefined the set of rows with data in each worksheet. This is the skeleton of the code that I use, which has the full path to the worksheet in the variable strFileToOpen:

```
Dim xlsApp As Object
Dim strFileToOpen as String

 Set xlsApp = CreateObject("Excel.Application")
 Set xlsApp = New Excel.Application
 xlsApp.Workbooks.Open strFileToOpen

 'Run your Import here
 'Check your Data here

 'Close the application
  xlsApp.Workbooks.Close
  set xlsApp = Nothing
```

In my skeleton code, I close the spreadsheet and set the xslApp variable to Nothing when I'm done. If I don't do this, Excel may just hang around in memory longer than I need it to. If I don't know how many rows will need to be imported, I need to find that out by examining the Excel spreadsheet. Using the "Current Region" method in the following code will do this, provided that the rows are always filled in and there aren't many gaps in the data:

```
Dim strSheetName as String
Dim strRowCount as Integer
Dim i, j  as Integer

xlsApp.Worksheets(strSheetName).Cells(i, j).Select
'Select the left top most cell in the work sheet.
'you can use CurrentWorksheet or sheet Index
'or as I have used the actual name.
'cell reference is by cell(row,column)

xlsApp.ActiveCell.CurrentRegion.Select
'this selects the region of data. It is much
'easier to use templates with fixed numbers
'of data entry rows.

strRowCount = xlsApp.Selection.Rows.count
```

The ideal case is to define a specific set of rows

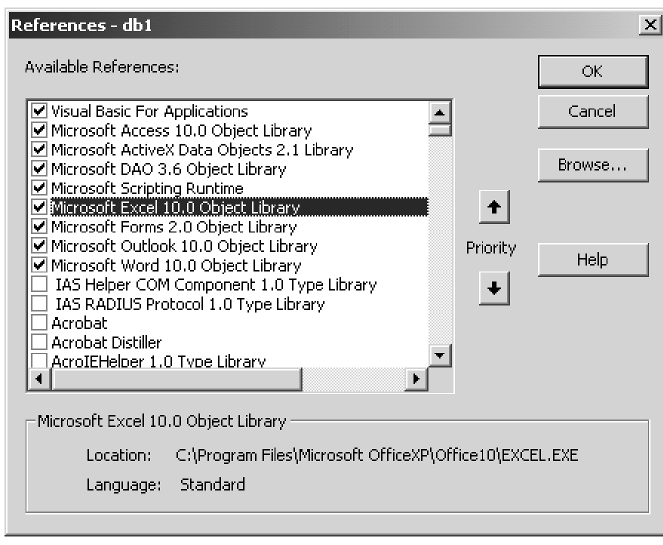


Figure 4. The reference manager in Visual Basic for Applications.

in an Excel template to be imported each time from a spreadsheet.

You could now set up a verification and validation routine that checks each piece of data against a rule or lookup table. If there's a problem with the data, you can set the color of the Excel cell to a predetermined color for a type of error. You could even insert an error message into an adjacent cell.

If there were no errors, I then extract the Excel data line by line and put it into the required table(s). I can input the data using either an ADO or DAO recordset or with a SQL statement. With a recordset I'd use the AddNew method to create new records and update the fields individually. Alternatively, I could create a SQL Insert or Update command and use the RunSQL method of the DoCmd object and avoid ADO and DAO altogether.

The next three code listings in the article provide an example of each method. The ADO and DAO methods are very similar.

Using SQL

The following code is an example of extracting data from each row in the spreadsheet selection and generating a SQL Insert statement that adds a record to a single table. I could, if needed, use two or more SQL statements to update more than one table, depending on the information contained in the spreadsheet.

Each row that I update will generate a message from Access as it begins the SQL update. A second message will display if there's an error. I set the warnings off to allow the process to flow without unnecessary messaging from Access. But you need to be careful! Access won't warn you if a row isn't inserted due to a key violation or data error, but will just skip on to the next row. So you need to make sure before executing your SQL that your data is ready to be inserted. Alternatively, you can devise a way to determine how many rows have been successfully inserted. If that number doesn't match the number of rows in your spreadsheet, then you know something has gone wrong and you must deal with the issue. Personally, I feel checking the data first and making sure the data is clean before doing the update is the neatest way. I'll assume, for now, that the data in the spreadsheet is all new records and clean.

```
Dim sqlStr as string
Dim strHole as string
Dim strFrom, strTo as Single
Dim strMag as Integer
Dim xlsSheetApp as string

xlsSheetApp = xlsApp.Worksheets(strSheetName)
DoCmd.SetWarnings False
For i = 1 To strRowCount
  strHole = xlsSheetApp.Cells(i, 1).Value
  strFrom = xlsSheetApp.Cells(i, 2).Value
  strTo = xlsSheetApp.Cells(i, 3).Value
  strMag = xlsSheetApp.Cells(i, 7).Value

  sqlStr = "INSERT INTO [table1]" &_
  "( Field1, Field2, Field3)" &_
        "VALUES  ( '" & strHole & "'," & _
   strFrom & "," & strTo & "," & strMag & " )"
```

```
    DoCmd.RunSQL sqlStr
Next i
```

While the RunSQL method is easy to implement if you know SQL, it does have some disadvantages. The lack of data error handling during the loop is one problem. Another disadvantage is speed. Using SQL writes from code isn't a problem if you're updating a block of data, but because I'm operating on a single row at a time I'm generating many RunSQL executions, each with its own overhead.

ADO and DAO

In the ADO and DAO methods, you need only one statement to open a connection to a table or tables. You can leave the table open during the loop and write the data row by row without reopening your connection. I've found that the greatest overhead with ADO or DAO is the open statement that connects to a table, so by opening the connection only once, performance is improved. The DAO and ADO statements are very similar.

The next code listing shows the same loop that I used in my RunSQL method, but with ADO used instead of a RunSQL statement:

```
Dim rstLogic As New ADODB.Recordset
Dim cnCurrent As ADODB.Connection

Set cnCurrent = CurrentProject.Connection
logic.Open "logic_faliures", cnthisconnect, _
  adOpenKeyset, adLockOptimistic, adCmdTable

For i = 1 To strRowCount
  strHole = xlsSheetApp.Cells(i, 1).Value
  strFrom = xlsSheetApp.Cells(i, 2).Value
  strTo = xlsSheetApp.Cells(i, 3).Value
  strMag = xlsSheetApp.Cells(i, 7).Value

  rstLogic.AddNew
  rstLogic![Field1] = strHole
  rstLogic![Field2] = strTo
  rstLogic![Field3] = strMag
  rstLogic.Update

Next i
rstLogic.Close
```

A compromise

A third alternative is to use the Execute method of ADO's connection object. This method allows me to submit SQL statements as I did with the RunSQL statement, but with the reduced overhead of the single open:

```
Dim cnCurrent As ADODB.Connection

Set cnCurrent = CurrentProject.Connection

For i = 1 To strRowCount
  strHole = xlsSheetApp.Cells(i, 1).Value
  strFrom = xlsSheetApp.Cells(i, 2).Value
  strTo = xlsSheetApp.Cells(i, 3).Value
  strMag = xlsSheetApp.Cells(i, 7).Value

  sqlStr = "INSERT INTO [table1]" &_
  "( Field1, Field2, Field3)" &_
        "VALUES  ( '" & strHole & "'," & _
   strFrom & "," & strTo & "," & strMag & " )"

  concurrent.Execute sqlStr

Next i
cnCurrent.Close
```

With these last two methods, I don't have to worry about suppressing messages and I can use standard VBA On Error statements to catch errors.

Validation in Excel

As I said earlier, it's important to build an Excel template for your users to enter their data into. A template will ensure that the data is entered the same way each time and provide a familiar layout for the users. Best of all, Excel has built-in validation that you can access from the menu item Data | Validation.

Figure 5 shows some of the options you have with data validation. You can apply the rule that you create to a single cell or to an entire column. Using the Input Message tab you can provide hints or tips to the data entry users that will appear when a user selects a cell that's had validation applied to it. These messages are handy because they're in the same format as an Excel Comment, are unobtrusive, and don't require the user to press a button to remove the message. The message appears to the right of the cell and, when the user moves out of the cell, the message disappears or is replaced by the message that refers to the cell that the user has just moved to.

If you find your users are being overloaded with messages, you can put a list of permanent messages in colored boxes to the side of the template. The Error Alert tab can be used to set up a message box to alert the user that data inconsistent with the input cell has been entered. Because this is a message box and it would appear each time an incorrect value is entered, you may choose to use a routine that checks the whole sheet after the data entry is complete.

While the message boxes are useful, I recommend that you limit their use to critical cases, as the boxes can become annoying to your users. If, for instance, you also use the Input Message option, the user is also prompted

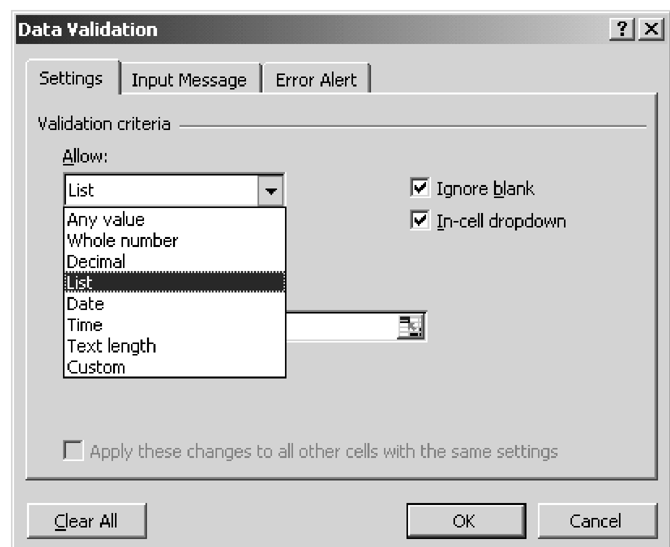


Figure 5. Setting up validation in Excel.

with an example of what should be in the cell. More critically, if the user takes advantage of copy and paste, then the data bypasses your validation rule! From the Error Alert tab, you can establish Information, Warning, or Stop message boxes with a description or example of the type of data that should be entered.

When you set up the validation rule, you have different options depending on the data type you select. For example, you can specify that the data be within a certain range, be greater than some value, be less than some value, and so on.

As I mentioned earlier, you can use many of the standard Windows controls in your Excel spreadsheet—dropdown lists included. Like a lookup table, a dropdown list allows you to control what data your users can put into a cell. If you decide to use a dropdown list for text-based input, you'll need to provide a location in the spreadsheet for the data in the list. In order to keep your users from modifying your data, you'll probably want to hide and/or protect the cells that contain your list(s) because, as the saying goes, "If the users can see the data, they will change the data."

When placing the data that you'll use in a dropdown list, you need to consider what will happen if a user deletes an entire row or column. Depending on where your data is placed, this action may delete information from your lists. One strategy that works is to partition your spreadsheet template into quadrants and then, using protection, "split" or "freeze panes" from the Windows Menu item. Placing the data input location for your users in either the top left quadrant or the bottom right quadrant and the lists in the opposite corner bypasses this row/column deletion problem.

Here are some key points to remember:

- Validation rules can be applied to a whole row or column—this is handy where a variable number of data entry rows will be entered each time.
- Copy and paste ignores the validation rules.
- Range data for lists can be accidentally deleted or changed, so you either need to protect that area of the spreadsheet or place it strategically (in the upper left hand corner and using a template, for instance).

Populating lists

If you're going to use dropdown lists to control data input, simply entering the valid values into your spreadsheet isn't a good choice. As the data in the database changes—including the data in lookup tables—the data in your Excel spreadsheet will get out of date. Rather than copying and pasting the data from Access to Excel (with the premise that you actually remember to do this!), a more elegant solution is to create a routine that executes when the template spreadsheet is loaded by the user. That routine can populate the spreadsheet from lookup tables in your database.

This code shows how to set up a list in Excel for a validation list. I begin by establishing a dummy range. My data will actually go in the area specified in the Formula1 property (in this case, U4 to U27):

```
Columns("G:G").Select
With Selection.Validation
  .Delete
  .Add Type:=xlValidateList, _
   AlertStyle:=xlValidAlertStop, _
   Operator:= xlBetween, _
   Formula1:="=$U$4:$U$27"
  .IgnoreBlank = True
  .InCellDropdown = True
  .InputTitle = ""
  .ErrorTitle = ""
  .InputMessage = ""
  .ErrorMessage = ""
  .ShowInput = True
  .ShowError = True
End With
```

Once a validation is established, you can use the Modify method of the Validation object to reset the range in "Forumla1" like this:

```
expression.Modify(Type, AlertStyle, Operator, _
    Formula1, Formula2)
```

The only required parameter for this method is Type. Expression is the selection.

I'll begin writing my Excel code by opening the VBA IDE in Excel and then opening the module for the Workbook by double-clicking ThisWorkBook in the Project window on the left-hand side of the IDE. Once the

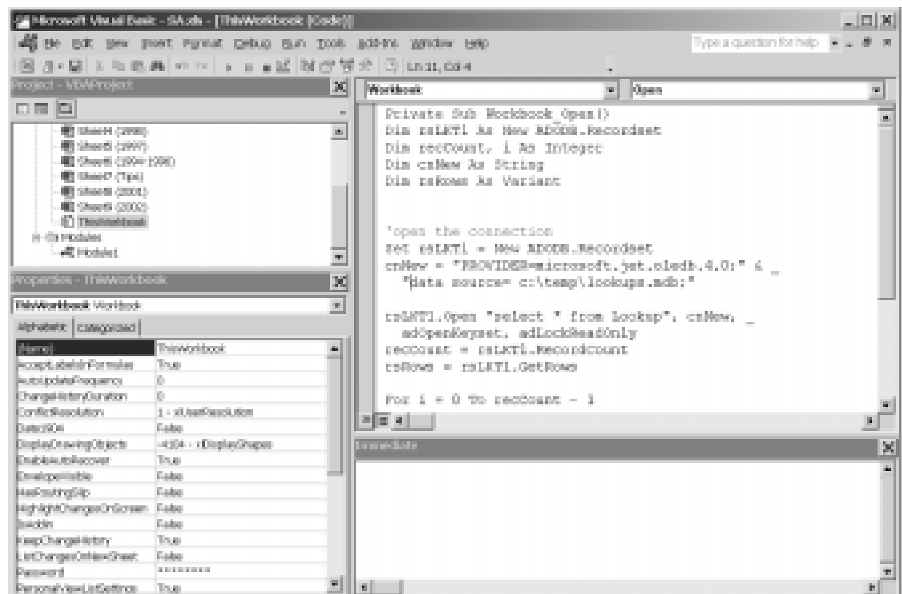


Figure 6. Setting the code to run when a workbook is loaded.

module opens, I'll select WorkBook in the dropdown list in the upper left-hand corner of the IDE and the Open event from the dropdown list on the right-hand side (see Figure 6, on page 14).

To import the data from Access, you could use the intrinsic Excel Data Import Routine that makes use of DSNs. Once again, I prefer to use ADO code to do the import, because using the DSN requires you to set up a DSN on each machine that the spreadsheet will be installed on.

Here's the code that will read my data from my database and insert it into the spreadsheet:

```
Dim rsLKTl As New ADODB.Recordset
Dim recCount, i As Integer
Dim cnNew As String
Dim rsRows As Variant


'open the connection
Set rsLKTl = New ADODB.Recordset
cnNew = "PROVIDER=microsoft.jet.oledb.4.0;" & _
  "data source=c:\temp\lookups.mdb;"

rsLKTl.Open "select * from Lookup", cnNew, _
  adOpenKeyset, adLockReadOnly
recCount = rsLKTl.RecordCount
rsRows = rsLKTl.GetRows

For i = 0 To recCount - 1
 Worksheets(1).Cells(i + 1, 8).Value = _
  rsRows(0, i)

Next i
rsLKTl.Close
Set rsLKTl = Nothing
```

This code creates a recordset and then loops through it, moving each value into a cell in the worksheet. In the Cell reference I'm actually asking Excel to use Cell "H1" and then to move down the column for succeeding rows. After this, I could then call my earlier code to add a new list that uses this data or modify the range of an existing list. When setting the list's range in the Formula1 property, I can adjust the range to match the number of records actually loaded.

Validating spreadsheet data

Dropdown lists won't handle all of your validation issues. To handle the remaining validation tasks you can use Access code, probably in the same routine that imports the spreadsheet. Alternatively, you can put the code in the Excel spreadsheet and run it from buttons in the template that start the import. If you do place buttons to check data in the template, I'd recommend that you only check to make sure that required fields have been entered and that data contained in cells is consistent with the rules that you've made with Excel validation (that is, you're catching errors that have slipped through via copy-and-pasting). Checking data against relationships and data rules is best done from Access.

The code behind your Excel buttons will typically loop through each row and look at particular cells to make sure either that data exists or that it's consistent with data that should be in those cells. I usually change the color of the cell when I find an error. This means that when I rerun my error checking, I must set the cell's color back to the default color so the user doesn't keep trying to fix data that's already been corrected. You may also want to place a comment against the cell with the error to help the user determine what's needed to correct the error.

This code shows an example where data entry is checked row by row, and checks that *(a)* one numeric field is greater than another, and *(b)* the call has data entered into it. The routine also adds a verbose comment to let the user know what the problem is:

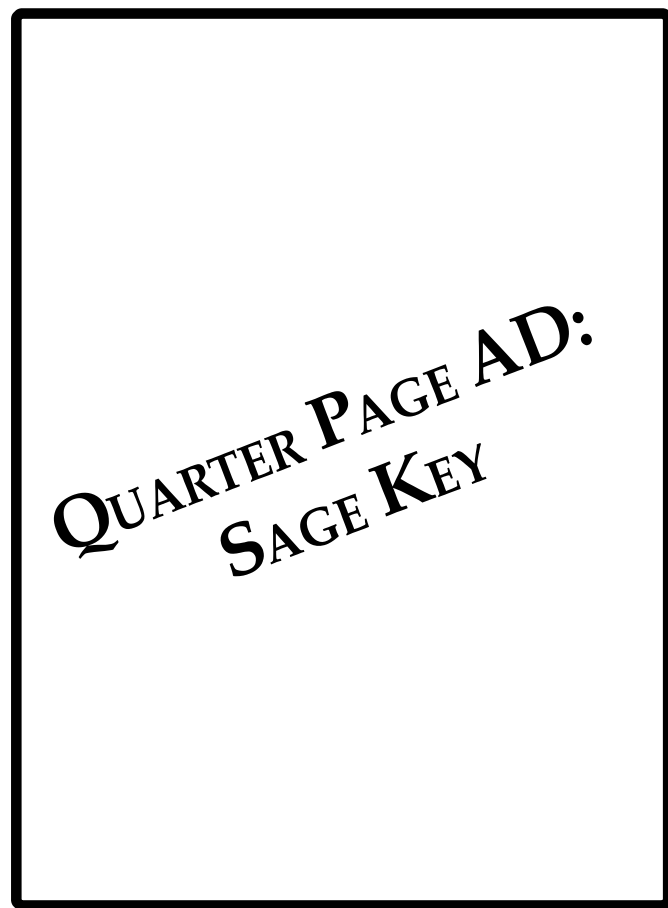
```
Dim strhole
Dim j
Dim i
Dim strfrom
Dim strto

For i = 1 To 100
 J=i+1
 strhole = Worksheets("Sheet1").Cells(j, 7).Value
 strfrom = Worksheets("Sheet1").Cells(j, 8).Value
 strto = Worksheets("Sheet1").Cells(j, 9).Value
 If strhole Like "OT*" And Len(strfrom) < 1 Then
   Worksheets("Sheet1").Cells(j, 8).Select
   Selection.Interior.ColorIndex = 3
   Selection.Interior.Pattern = xlSolid
 End If

 If Len(strhole) < 1 Then
  Worksheets("Sheet1").Cells(j, 7).Select
  Selection.Interior.ColorIndex = 3
  Selection.Interior.Pattern = xlSolid
 End If

 If strfrom > strto Then
  Worksheets("Sheet1").Cells(j, 9).Select
  Selection.Interior.ColorIndex = 5
```

Continues on page 19



Continued from page 15

```
  Selection.Interior.Pattern = xlSolid
  ActiveCell.AddComment
  ActiveCell.Comment.Visible = False
  ActiveCell.Comment.Text = _
          "To should be Greater than From"

Next i
```

Finally, I validate the data in each row of the spreadsheet against relationship rules and lookup tables in the database. You should do this in a batch before importing, rather than have a message box appear every time there's an error during import. If there are no errors, you can start the batch import of the spreadsheet data. If there are errors, coloring the problem data cells will let the user know where the problem is.

This next code listing can be used in a loop of the Excel sheet or within Access. In this case, I'm testing the one-to-many relationship between the table that I'm inserting the record into and a table called Hole. The field Hole is the Primary Key in the Hole table, and I need to make sure that my value exists in the Primary Key table:

```
rstHole.Open "SELECT * FROM HOLE WHERE [Hole] & _
   =  '" & strHole & "'", cnthisconnect, _
   adOpenForwardOnly, adLockReadOnly, adCmdText
strHole1 = rstHole("Hole")
strFrom1 = rstInterval("fFrom")
 If isnull(strHole1) = true or strhole1 = "" Then
    'Error Hole must exist to create one to many
    'relationship!!!
  Else
  If strFrom = strFrom1 then
```

```
    'Error strFrom is part of the key with Hole
    'to form a one to many relationship
    'strFrom & StrHole cannot exist already!!
  End if
End If
```

I've used ADO here, but you can also use DAO (for simple checks, DLookup could be used).

From here it's a simple matter to write error colors and comments back into the Excel spreadsheet and end the session, warning the user. If I don't find any problems, I can continue on to the import phase that I discussed in the first section.

Using Excel for data entry runs a poor second against user interfaces that we can build in Access. But, if you do have to deal with Excel as a data entry front end, then with a little hard work and good design you can provide your users with a user interface that leaves them with a comfortable "warm fuzzy spreadsheet feeling." You can also ensure that your data is valid, which reduces the amount of work required by the database administrator, which is probably you. ▲

Scott McManus is the principal of Skandus. Scott's interests are in object- rich databases, XML, Outlook, and 3D objects in the mining industry. Outside the office, he enjoys archaeology, natural science, music, and the Mid North Coast of NSW.

May 2003 Downloads

- NETREPRT.ZIP—Danny Lesandrini has sent along the code that shows how to export your Access report as XML. Once your Access report is created as an XML file, you can re- create it as an HTML report. With your report in XML, you can integrate Access with .NET to become your .NET Web reporting tool. (Access 2000)
- ORDOUT.ZIP—The sample database in this download processes e-mail messages from a third-party ordering system. The code assumes the format for an e-mail message produced by DigiBuy. You can test the code using a sample e-mail generated by code in the download database. (Access 2000)
- EXCDATA.ZIP—Scott McManus demonstrates how to use Excel as a data entry system for Access complete with data validation. (Access 2000)
- AA0503.ZIP—This month's "Access Answers" column's sample database contains a single query: It calculates the median freight charge for the Northwind OrderDetails table, using a single query. (Access 2000)

For access to all current and archive content and source code, log in at www.vb123.com/kb with your unique subscriber user name and password.
For access to this issue's Downloads only, click on the "Source Code" button, select the file(s)
you want from this issue, and enter the User name and Password at right when prompted.

User name **iguana**

Password **jade**

Smart Access (ISSN 1066-7911)

is published monthly (12 times per year) by:

Pinnacle
A division of Lawrence Ragan Communications, Inc.
316 N. Michigan Ave., Suite 400
Chicago, IL 60601

POSTMASTER: Send address changes to Lawrence Ragan Communications, Inc., 316 N. Michigan Ave., Suite 400, Chicago, IL 60601.

Copyright © 2003 by Lawrence Ragan Communications, Inc. All rights reserved. No part of this periodical may be used or reproduced in any fashion whatsoever (except in the case of brief quotations embodied in critical articles and reviews) without the prior written consent of Lawrence Ragan Communications, Inc. Printed in the United States of America.

Brand and product names are trademarks or registered trademarks of their respective holders. Microsoft is a registered trademark of Microsoft Corporation. Microsoft Access is a registered trademark of Microsoft Corporation. *Smart Access* is an independent publication not affiliated with Microsoft Corporation. Microsoft Corporation is not responsible in any way for the editorial policy or other contents of the publication.

This publication is intended as a general guide. It covers a highly technical and complex subject and should not be used for making decisions concerning specific products or applications. This publication is sold as is, without warranty of any kind, either express or implied, respecting the contents of this publication, including but not limited to implied warranties for the publication, quality, performance, merchantability, or fitness for any particular purpose. Lawrence Ragan Communications, Inc, shall not be liable to the purchaser or any other person or entity with respect to any liability, loss, or damage caused or alleged to be caused directly or indirectly by this publication. Articles published in *Smart Access* do not necessarily reflect the viewpoint of Lawrence Ragan Communications, Inc. Inclusion of advertising inserts does not constitute an endorsement by Lawrence Ragan Communications, Inc., or *Smart Access*.