Article: How To Write Sql Statements In Vba
Date:
Posted by: HeelpBook Staff
Source: Link
Permalink: Link

ACCESS – VBA – HOW TO WRITE SQL STATEMENTS IN VBA

This tutorial introduces some **SQL** basics, those essential rules you really need to know when working with **SQL** in your Access databases. This series of tutorials concentrates on combining **SQL** with **VBA** to help build more powerful, flexible and user-friendly databases. Here's what this tutorial contains:

- A 5-minute Course in SQL;
- Getting VBA to Speak SQL;
- How to Write SQL in VBA;
- Working with Variables;
- Debugging Your SQL Code;

A 5-minute Course in SQL

Some Technical Terms

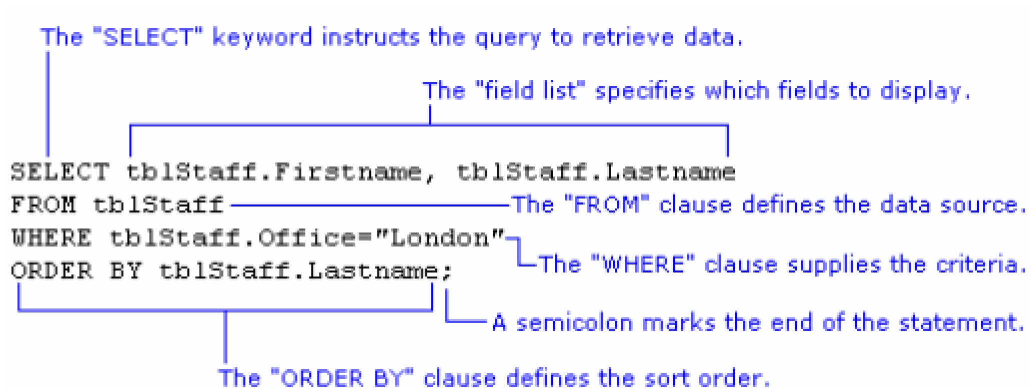
- SQL contains a number of **keywords** such as ***SELECT, DELETE, UPDATE, FROM, WHERE, OR, AND, DISTINCT*** and many others.

- SQL keywords are usually combined with **arguments** in the form of table names, field names, criteria etc. to form an **SQL statement**.

- An SQL statement may contain one or more **clauses** such as a ***WHERE*** clause (***containing the criteria of a query***) or an ***ORDER BY*** clause (determining the order in which a query's data is displayed).

The Structure of an SQL Statement

This illustration shows the principal parts of a typical **SQL** query statement:



For clarity, the different clauses are shown on separate lines. This is how the SQL view of the Access query design window displays its SQL.

What About the Brackets?

Access uses parentheses (round brackets) to enclose the various parts of the WHERE clause but these can be left out if you find this easier (I do!).

```
WHERE (((tblStaff.Office)="London”))
```

is the same as:

```
WHERE tblStaff.Office="London”
```

Don't Skimp on Information

Whenever a field is specified you have the option to append the table name, separating the two with a dot.

```
SELECT Firstname, Lastname FROM tblStaff ORDER BY Lastname
```

is the same as:

```
SELECT tblStaff.Firstname, tblStaff.Lastname FROM tblStaff ORDER BY
tblStaff.Lastname
```

providing that the fields belong to the data source specified in the **FROM** clause. If your query refers to more than one table you must include the table name along with the field name.

Data Type Qualifiers

When supplying values to an **SQL** statement, for example as query criteria, their data type must be correctly defined by a "***qualifier***". This is done by enclosing the value between a pair of appropriate characters.

Text must be enclosed in either single quotes (') or double quotes ("), for example:

```
WHERE tblStaff.Department = "Marketing”
```

or

```
WHERE tblStaff.Department = 'Marketing’
```

A **Date** should be enclosed in hash marks (**#**) also called pound or number signs, for example:

```
WHERE tblStaff.BirthDate = #09/27/1950#
```

A number, of any sort, needs no qualifier and can be entered as it is, for example:

```
WHERE tblInvoices.InvoiceNumber > 1500
```

Get the Date Format Right

Dates in SQL must be written in the ***US date format*** (**month/day/year**). This is imperative regardless of the default date format settings on your computer. The **Access** query design window accepts dates in your local default format but it converts the date you type to the correct format when it builds the **SQL** statement.

Remember the Semicolon!

An SQL statement must finish with a semicolon (**;**). If you omit the semicolon when writing an SQL statement in the SQL view of the Access query design window your query will still work because Access corrects the error for you! You must remember to include it when writing SQL in VBA and elsewhere.

Getting VBA to Speak SQL

VBA and **SQL** are different things. **VBA** is a ***full-blown programming language*** that you can use to get **Access** (and other **Microsoft Office** programs) to do just about anything you want. **SQL** is a language used exclusively to manipulate the data and structure of a database.

VBA calls on a vast range of objects, properties, methods, functions and constants to construct the sometimes complex statements used to control the program. **SQL** uses a limited range of "keywords" combined with information you supply, such as table names, field names and criteria, to construct essentially simple (***although sometimes long***) statements detailing what you want the program to do with your data.

SQL can often be used alone, for example when setting the *RecordSource* property of a form or report in design view, or when working in the *SQL View* of the **Access** query design window. But this tutorial is about how to combine **SQL** with **VBA**. Its aim is explain the rules of **SQL** and to encourage you to use good code-writing practice and to avoid the pitfalls and problems that can occur. The way you write your code is a very personal thing so I'm going to show you how I do things, using some techniques I have learnt from others and some I've figured out myself.

You don't have to do it my way, but it works for me so it's what I teach!

VBA is a very flexible language and there are often many different ways in which **VBA** can achieve the same task. **SQL** on the other hand is a very precise and inflexible language. Everything has to be just so, but it has the advantage of also being very simple.

How to Write SQL in VBA

Different developers have their own ways of doing things and this is reflected in their coding style. What you will see here is the way I like to write my code.

Use a String Variable

Whenever you write SQL into your VBA code it is important to remember that the SQL is always in the form of a text string. SQL statements can also be quite long, and for that reason they are usually assigned to text variables so that they are easier to handle.

When working with SQL in a VBA procedure I usually assign the SQL to a text variable and I usually name my variable *strSQL*. Something like this...

```
Dim strSQL As String
strSQL = "... THE SQL STATEMENT GOES HERE ..."
DoCmd.RunSQL strSQL
```

Of course you could do away with the variable and apply the **SQL** directly, like this:

```
DoCmd.RunSQL "... THE SQL STATEMENT GOES HERE ..."
```

But, as you will see in the later tutorials building an SQL statement might involve several stages and many lines of code so I usually opt to store it in a variable.

Write SQL Keywords in Upper Case

In case you haven't already noticed, I always write the SQL keywords in capitals (upper case). Access doesn't care if you do this or not but you will find your code much easier to read and understand if you do. Compare these two statements:

```
Select tblStaff.Firstname, tblStaff.Lastname from tblStaff where
tblStaff.Office="Paris";

SELECT tblStaff.Firstname, tblStaff.Lastname FROM tblStaff WHERE
tblStaff.Office="Paris";
```

I am using simple examples here but imagine trying to read an SQL statement that was ten times as long as these (**NOTE**: the maximum length of an SQL statement in VBA is **32,768 characters**!).

Enclose Field Names in Square Brackets

Okay, I'm getting picky here, but I always put square brackets around field names. **Access** only demands that you do this when your field names contain spaces. For example, FirstName is OK but in your code First Name must be written [First Name]. The brackets tell Access that all the words in the field name belong together. They also tell Access that "this is a field name" and so allows you to use otherwise reserved words for the names of fields (such as **[Date]** which is also the name of a function) without causing conflicts.

But I do this for another reason too. I know that if I see some text in square brackets I know it's a field name, whether it has spaces in it or not...

```
SELECT tblStaff.[Firstname], tblStaff.[Lastname] FROM tblStaff WHERE
tblStaff.[Office]="Paris";
```

Write Each Clause on a Separate Line

Unless my **SQL** statement is very short, for example:

```
SELECT tblStaff.* FROM tblStaff;
```

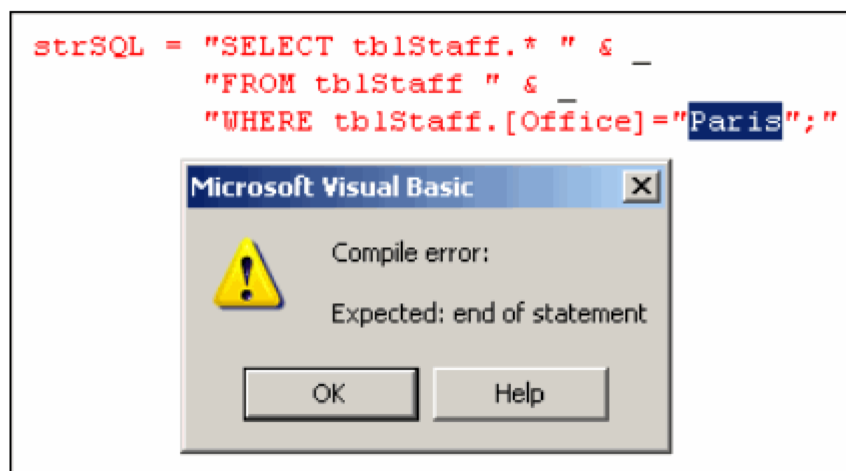
I like to write each clause of the SQL statement on a separate line. This makes long statements much easier to read and edit...

```
SELECT tblStaff.[Firstname], tblStaff.[Lastname]
FROM tblStaff
WHERE tblStaff.[Office]="Paris”;
```

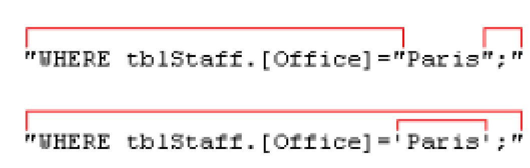
When you do this in a **VBA** procedure you must remember that you are dealing with a single string of text, so you must use the VBA line break character (a space followed by an underscore) and concatenate the lines using an ampersand (&). Don't forget that each line must have both opening and closing quotes.

Alternate Single and Double Quotes

It's easy to get your quotes confused when you are constructing an SQL statement in VBA. The statement itself needs to be enclosed in quotes because it is a VBA text string. But you might also have some text in the **SQL** as criteria in the **WHERE** clause. If you use the same type of quote mark for each Access will get confused. Look at this example...



There is a problem with nested quotes in the **WHERE** clause. Compare the two examples below. In the first example the **VBA** sees two text strings enclosed by double quote marks, and between them a word it doesn't know (Paris) so it generates an error.

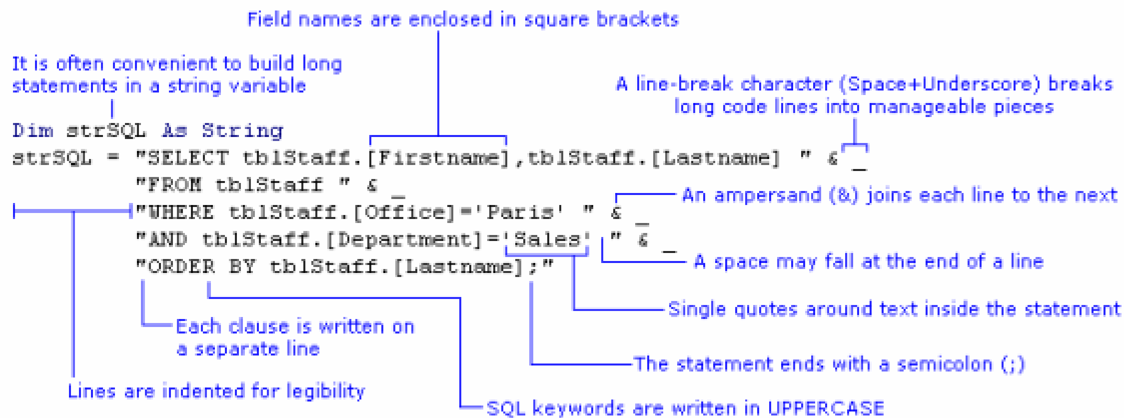


But when the quote marks are alternated as shown in the second example, the problem doesn't arise. The VBA sees a text string enclosed by double quotes, inside which is some more text enclosed in single quotes.

I working with multiple sets of quotes gets confusing, you can always use the **ASCII** character code for the double quote mark - **Chr(34)** - instead. There is an example of this in the next section.

Putting It All Together

Some of these rules are essential, others are just my way of doing things (**and that of many other database developers**). Follow them and you will write good code that is easy to read and to debug. The illustration below shows a completed **SQL** statement written the way I suggest **[click the thumbnail to see a full-sized image]**:



Working with Variables

In the examples I have shown so far, the criteria in the **WHERE** clause have been "**hard-coded**", written directly into the **SQL** statement. But this won't often be the case. One of the main reasons for working with SQL in your VBA procedures is that you can make changes to things.

You might be changing the criteria, fields or even data sources specified in your **SQL** statements each time the code is run. The information that the **SQL** statement needs is often obtained from the user through their choices in a dialog box or from the values in fields on a form. Forthcoming tutorials in this series will show how this can be done.

Consider this simple example where the criteria value is "**hard-coded**" into the **WHERE** clause of the **SQL** statement:

```
Dim strSQL As String
strSQL = "SELECT tblStaff.[Firstname],tblStaff.[Lastname] " & _
         "FROM tblStaff " & _
         "WHERE tblStaff.[Office]='Paris';"
```

You want to allow the user to choose a value for the **Office** criteria each time the query is run, so you build a dialog box in which there is a combo box containing a list of **Offices**. The combo box is named **cboOffice**. You can insert a reference to the value of the combo box directly into the **SQL** statement:

```
Dim strSQL As String
strSQL = "SELECT tblStaff.[Firstname],tblStaff.[Lastname] " & _
         "FROM tblStaff " & _
         "WHERE tblStaff.[Office]='" & Me.cboOffice.Value & "';"
```

Alternatively, you can place the value of the **combo box** into a variable and then insert the variable into the **SQL** statement:

```
Dim strSQL As String
Dim strOffice As String
strOffice = Me.cboOffice.Value
strSQL = "SELECT tblStaff.[Firstname],tblStaff.[Lastname] " & _
         "FROM tblStaff " & _
         "WHERE tblStaff.[Office]='" & strOffice & "';"
```

Using a variable can make the **SQL** statement code easier to read and understand, especially when there are several variable criteria to consider. It is sometimes essential to use this method when the value has to be examined or manipulated in some way before it is passed to the **SQL**.

Whatever method you choose, you must remember to include any necessary data type qualifiers in the **SQL** string. In the illustration below, a single quote mark is included in the **SQL** string either side of the text variable (**marked with red arrows**):

```
"WHERE tblStaff.[Office]='" & strOffice & "';"
```

Alternatively, the **ASCII** character code for the double quote mark (**Chr(34)**) can be inserted either side of the text variable, but outside the SQL string. This method requires more typing but avoids conflicts and confusion arising from nesting quotes.

```
"WHERE tblStaff.[Office]=" & Chr(34) & strOffice & Chr(34) & ";"
```

Remember that as with "**hard-coded**" criteria, variables require the correct qualifiers for their data type: quotes for text, hash-marks (pound signs) for dates, and nothing for numbers.

Debugging Your SQL Code

As with any other sort of programming, you **SQL** code can fail to work properly. It might be because you made a **logic error**, or got the SQL syntax wrong, or perhaps you just made a typo. If this results in workable code it might not produce the result you were expecting. Most often though the result is code that won't run and Access throws up an error. Your job is to figure out what went wrong and put it right.

If you are running an **SQL** statement from within a **VBA** procedure you will see the regular Visual Basic error message. Sometimes these are quite difficult to interpret.

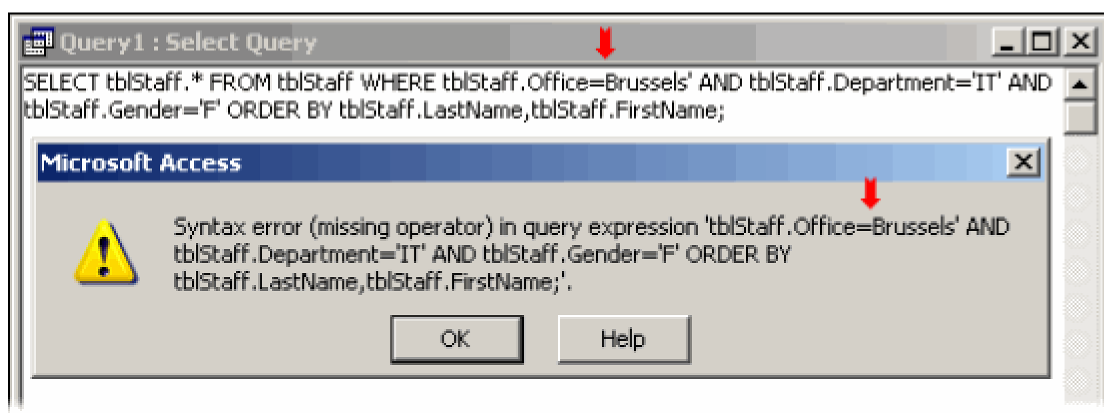
A trick used by many Access developers is to test their **SQL** in the **SQL View** of the **Access** query design window. If there is a problem **Access** will display an error message. The error messages you get from the **SQL** View of the Access query tool are often far more helpful and descriptive that those you get from **VBA**!

SQL Error Messages

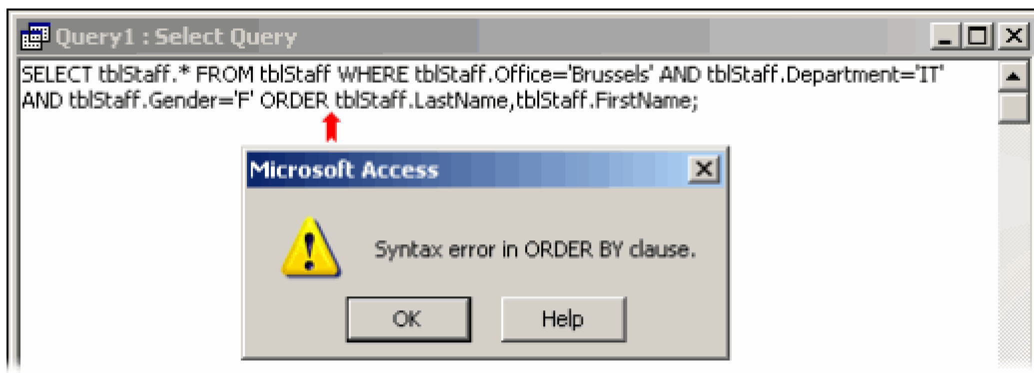
It pays to familiarise yourself with the different sorts of message so that you can quickly trace the source of code errors. Most SQL errors are syntax errors. The Jet database engine can't interpret the **SQL** statement because it doesn't make sense. These often arise from simple typographical errors or omissions so it is important, as with most computer programming, to take care when typing! Here are some examples (**NOTE**: **the red arrows are mine - the messages aren't that helpful!**):

The most common type of error is the "**missing operator**". You would normally understand the term "**operator**" to mean a mathematical symbol such as =,> or < but when an **Access SQL** error message uses the expression "**missing operator**" you should look for something left out of the statement.

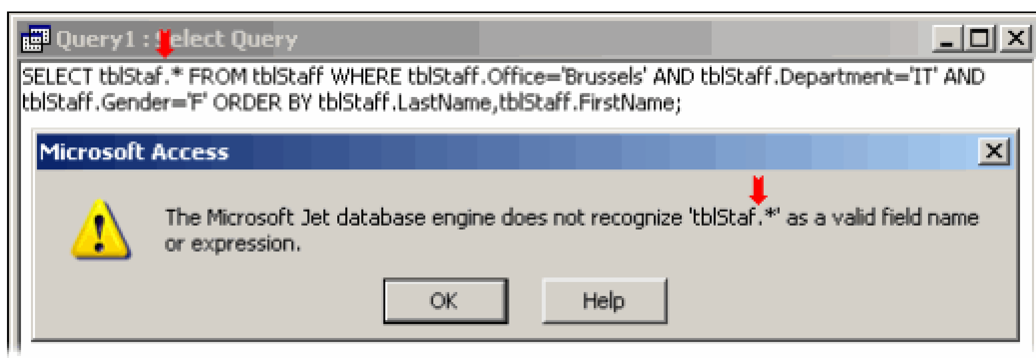
It could be a mathematical operator but it might also be a word like **AND** or **OR**, or perhaps a quote mark. Here the quote mark before the word **Brussels** is missing:



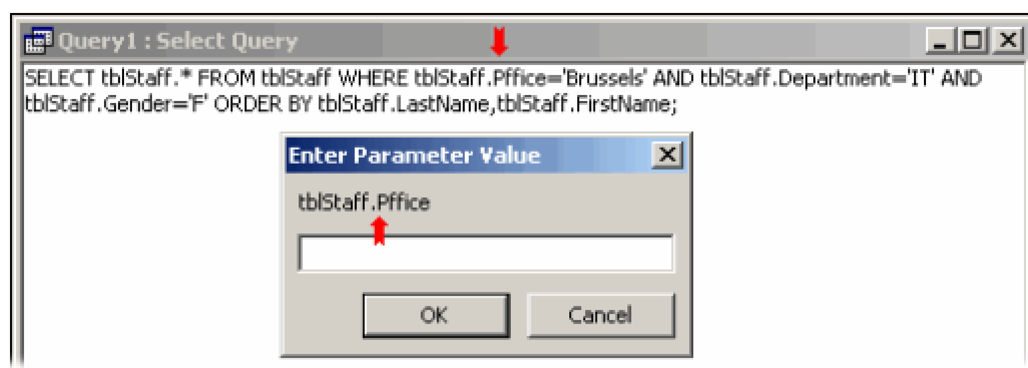
Sometimes error messages are more specific and point you directly to the clause that is causing the problem. Here the word "**BY**" has been omitted from the **ORDER BY** clause of the **SQL** statement:



If you misspell the name of a database object such as a table, or refer to one that doesn't exist, the **Jet** database engine will not recognise it. These errors are usually quite easy to trace:

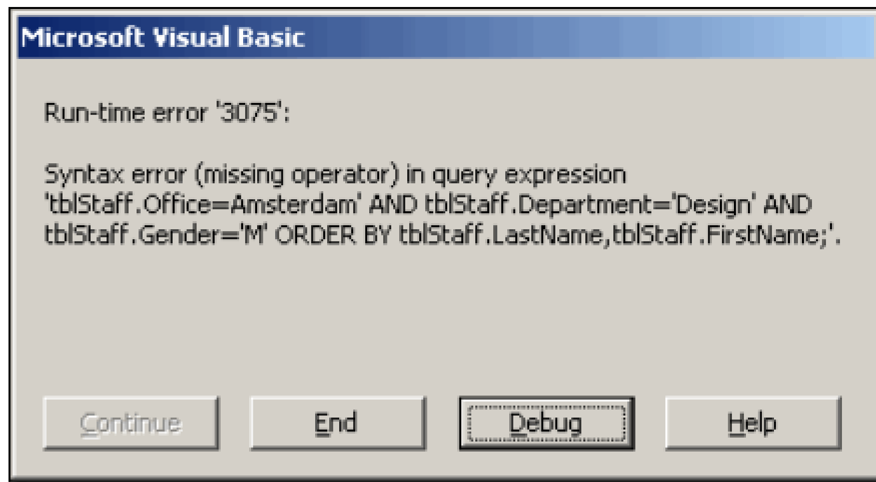


Not really an error message, but a response to an error in the **SQL** statement. If you type the name of a field incorrectly, the ***Jet database engine*** sometimes interprets your **SQL** as a parameter query and displays the familiar parameter dialog. Although this might seem confusing at first, it is usually quite easy to diagnose and correct because the offending name is displayed on the dialog. You just need to find it in your code and correct it:



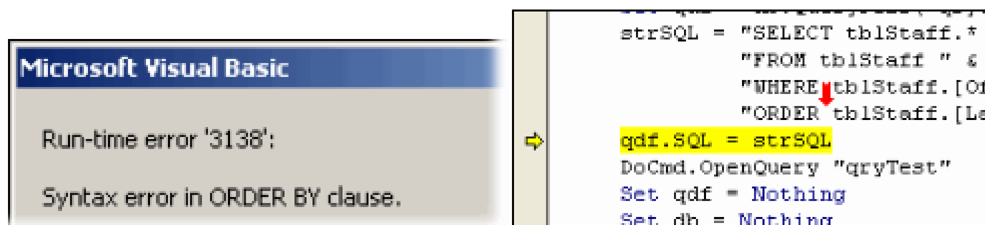
VBA Error Messages

When you try to run a faulty **SQL** statement from **VBA** it will usually generate a **run-time error** resulting in a error VBA message like this one:



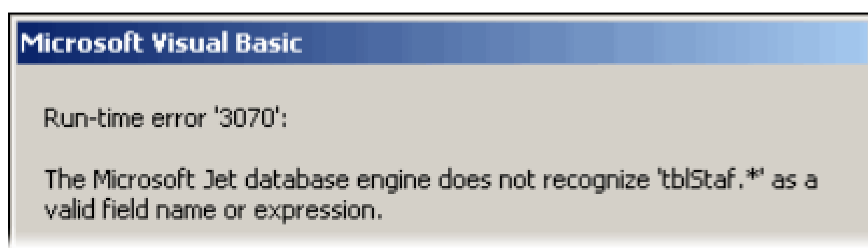
The error messages are delivered in the usual way using the **Visual Basic** error message dialog and don't explicitly state that the problem with your code lies in your **SQL** statement, although this is usually evident from the message itself.

Pressing the **Debug** button will take you to the offending line of code but this may not be the **SQL** statement itself. Here are some examples:



In the example above the code crashed when it tried to apply the **SQL** statement to a stored query. **Access** checked the **SQL** statement before applying it to the query and found a syntax error ("**ORDER**" should be "**ORDER BY**") and highlighted the code line that it could not execute, rather than the one that contained the error.

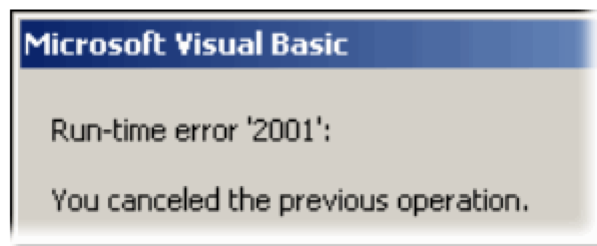
In the next example, it is clear that there is a spelling error somewhere but it might take a while to find:



Here, the error lies in the misspelling of a table name in the **SELECT** clause of the **SQL** statement but the error did not arise until the code tried to run the query and the specified table could not be found.

```
strSQL = "SELECT tblStaf.* "
         "FROM tblStaff " &
         "WHERE tblStaff.[Of
         "ORDER BY tblStaff.
qdf.SQL = strSQL
DoCmd.OpenQuery "qryTest"
Set qdf = Nothing
```

The next example must get the prize for the most confusing error message that **Access** has to offer!



I have seen it in two different circumstances. It is displayed if an error occurs because, from a **VBA** procedure you ran a parameter query - or a query that thought it was a parameter query and so displayed the parameter dialog (**see the example above where the field name was spelled incorrectly**) - and you clicked the parameter dialog's **Cancel** button.

In normal circumstances that would be **OK** but if as a result your code crashed, the "*You canceled the previous operation.*" message is displayed and, I suppose, it makes sense.

But this message also appears when there is a different kind of error, when you attempt to use an **SQL** statement from **VBA** in which the data type of a **WHERE** clause's criteria does not match the data type of the corresponding field.

For example you might have a date field and supply a text data type:

WHERE tblStaff.BirthDate='Tuesday'

instead of matching the data types correctly:

WHERE tblStaff.BirthDate=#09/27/1950#.

Include an Error Handling Routine

Finally, remember that a faulty **SQL** statement can crash your **VBA** code so remember to test your procedures thoroughly and always include an error handling routine.