Access Tutorial 14: Data Access Objects

14.1 Introduction: What is the DAO hierarchy?

The core of Microsoft Access and an important part of Visual Basic (the stand-alone application development environment) is the Microsoft Jet database engine. The relational DBMS functionality of Access comes from the Jet engine; Access itself merely provides a convenient interface to the database engine.

Because the application environment and the database engine are implemented as separate components, it is possible to upgrade or improve Jet without altering the interface aspects of Access, and vice-versa.

Microsoft takes this component-based approach further in that the interface to the Jet engine consists of a hierarchy of components (or "objects") called Data Access Objects (DAO). The advantage of DAO is that its modularity supports easier development and maintenance of applications.

The disadvantage is that is you have to understand a large part of the hierarchy before you can write your first line of useful code. This makes using VBA difficult for beginners (even for those with considerable experience writing programs in BASIC or other 3GLs*).

14.1.1 DAO basics

Although you probably do not know it, you already have some familiarity with the DAO hierarchy. For example, you know that a **Database** object (such as `univ0_vx.mdb`) contains other objects such as tables (**TableDef** objects) and queries (**QueryDef** objects). Moving down the hierarchy, you know that TableDef objects contain **Field** objects.

* Third-generation programming languages.

Last update: 25-Aug-1997

Unfortunately, the DAO hierarchy is somewhat more complex than this. However, at this level, it is sufficient to recognize three things about DAO:

1. Each object that you create is an **instance** of a **class** of similar objects (e.g., `univ0_vx` is a particular instance of the class of Database objects).
2. Each object may contain one or more **Collections** of objects. Collections simply keep all objects of a similar type or function under one umbrella. For example, Field objects such as `DeptCode` and `CrsNum` are accessible through a Collection called **Fields**).
3. Objects have **properties** and **methods** (see below).

14.1.2 Properties and methods

You should already be familiar with the concept of object properties from the tutorial on form design (Tutorial 6). The idea is much the same in DAO: every object has a number of properties that can be either observed (read-only properties) or set (read/write properties). For example, each TableDef (table definition) object has a read-only property called *DateCreated* and a read/write property called *Name*. To access an object's properties in VBA, you normally use the `<object name>.<property name>` syntax, e.g., `Employees.DateCreated`.

To avoid confusion between a property called `DateCreated` and a field (defined by you) called `DateCreated`, Access version 7.0 and above require that you use a bang (!) instead of a period to indicate a field name or some other object created by you as a developer. For example: `Employees!DateCreated.Value` identifies the *Value* property of the `DateCre-`

ated field (assuming one exists) in the Employees table.

Methods are actions or behaviors that can be applied to objects of a particular class. In a sense, they are like predefined functions that only work in the context of one type of object. For example, all Field objects have a method called FieldSize that returns the size of the field. To invoke a object's methods, you use the <object name>.<method> [parameter$_1$, ..., parameter$_n$] syntax, e.g.,: DeptCode.FieldSize.

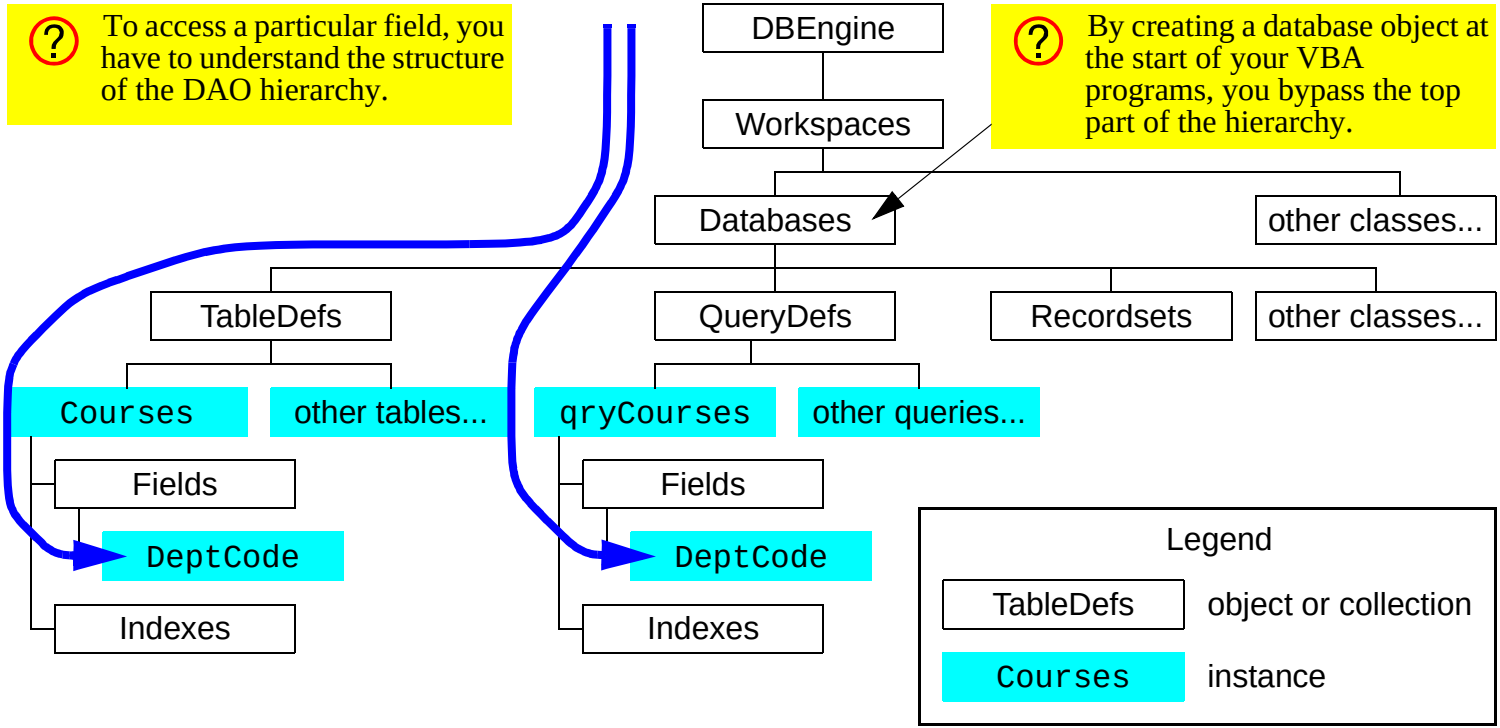
A reasonable question at this point might be: Isn't FieldSize a property of a field, not a method? The answer to this is that the implementation of DAO is somewhat inconsistent in this respect. The best policy is to look at the object summaries in the on-line help if you are unsure.

A more obvious example of a method is the CreateField method of TableDef objects, e.g.: Employees.CreateField("Phone", dbText, 25)
This creates a field called Phone, of type dbText (a constant used to represent text), with a length of 25 characters.

14.1.3 Engines, workspaces, etc.

A confusing aspect of the DAO hierarchy is that you cannot simply refer to objects and their properties as done in the examples above. As Figure 14.1 illustrates, you must include the entire path through the hierarchy in order to avoid any ambiguity between, say, the DeptCode field in the Courses TableDef object and the DeptCode field in the qryCourses QueryDef object.

FIGURE 14.1: Navigating the DAO hierarchy.



Working down through the hierarchy is especially confusing since the first two levels (**DBEngine** and **Workspaces**) are essentially abstractions that have no physical manifestations in the Access environment. The easiest way around this is to create a Database object that refers to the currently open database (e.g., `univ0_vx.mdb`) and start from the database level when working down the hierarchy. Section 14.3.1 illustrates this process for version 2.0.

14.2 Learning objectives

- ❒ What is the DAO hierarchy?
- ❒ What are objects? What are properties and methods?
- ❒ How do I create a reference to the current database object? Why is this important?
- ❒ What is a recordset object?
- ❒ How do I search a recordset?

14.3 Tutorial exercises

14.3.1 Setting up a database object

In this section you will write VBA code that creates a pointer to the currently open database.

- Create a new module called `basDAOTest` (see Section 12.3.3 for information on creating a new module).
- Create a new subroutine called `PrintRecords`.
- Define the subroutine as follows:

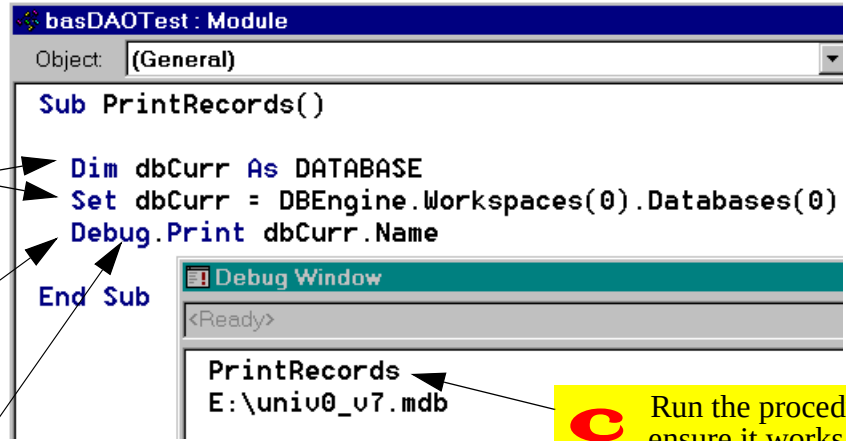
```
Dim dbCurr As DATABASE
Set dbCurr =
  DBEngine.Workspaces(0).Databases(0)
Debug.Print dbCurr.Name
```

- Run the procedure, as shown in Figure 14.2.

Let us examine these three statements one by one.

1. `Dim dbCurr As DATABASE`
   This statement declares the variable `dbCurr` as an object of type Database. For complex objects

FIGURE 14.2: Create a pointer to the current database.



a Declare and set the pointer (`dbCurr`) to the current database.

b Add a line to print the name of the database.

? Although you can use the `Print` statement by itself in the debug window, you must invoke the `Print` method of the Debug object from a module—hence the `Debug.Print` syntax.

c Run the procedure to ensure it works.

? Version 7.0 and above support a less cumbersome way referring to the current database—the `CurrentDb` function:
`Set dbCurr = CurrentDb`

(in contrast to simple data types like integer, string, etc.) Access does not allocate memory space for a whole database object. Instead, it allocates space for a **pointer** to a database object. Once the pointer is created, you must set it to point to an object of the declared type (the object may exist already or you may have to create it).

2. `Set dbCurr = DBEngine.Workspaces(0).Databases(0)`
   (Note: this should be typed on one line). In this statement, the variable `dbCurr` (a pointer to a Database object) is set to point to the first Database in the first Workspace of the only Database Engine. Since the numbering of objects within a collection starts at zero, `Databases(0)` indicates the first Database object. Note that the first Database object in the Databases collection is always the currently open one.

Do not worry if you are not completely sure what is going on at this point. As long as you understand that you can type the above two lines to create a pointer to your database, then you are in good shape.

3. `Debug.Print dbCurr.Name`
   This statement prints the name of the object to which `dbCurr` refers.

14.3.2 Creating a Recordset object

As its name implies, a TableDef object does not contain any data; instead, it merely defines the structure of a table. When you view a table in design mode, you are seeing the elements of the TableDef object. When you view a table in datasheet mode, in contrast, you are seeing the contents of **Recordset** object associated with the table.

To access the data in a table using VBA, you have to invoke the `OpenRecordset` method of the Database object. Since most of the processing you do in VBA involves data access, familiarity with Recordset objects is essential. In this section, you will create a Recordset object based on the `Courses` table.

- Delete the `Debug.Print dbCurr.Name` line from your program.
- Add the following:

```
Dim rsCourses As Recordset
Set rsCourses =
  dbCurr.OpenRecordset("Courses")
```

The first line declares a pointer (`rsCourses`) to a Recordset object. The second line does two things:

1. Invokes the `OpenRecordset` method of `dbCurr` to create a Recordset object based on the table named `"Courses"`. (i.e., the name of the table is a parameter for the `OpenRecordset` method).
2. Sets `rsCourses` to point to the newly created recordset.

Note that this `Set` statement is different than the previous one since the `OpenRecordset` method results in a new object being created (`dbCurr` points to an existing database—the one you opened when you started Access).

14.3.3 Using a Recordset object

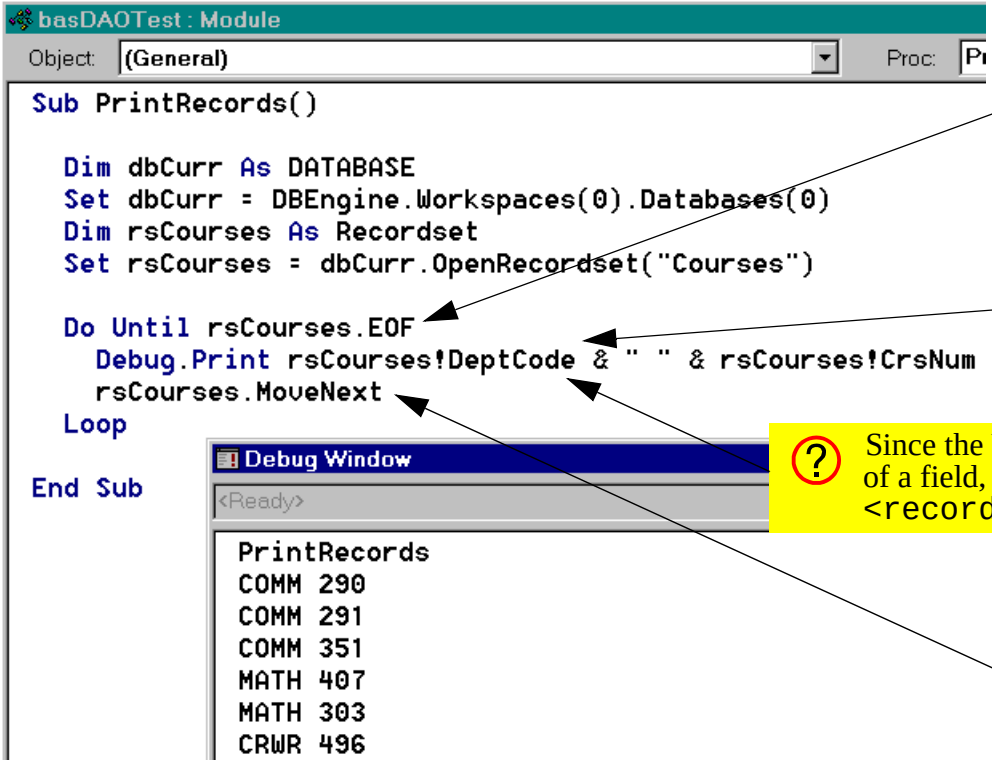
In this section, you will use some of the properties and methods of a Recordset object to print its contents.

- Add the following to `PrintRecords`:

```
Do Until rsCourses.EOF
Debug.Print rsCourses!DeptCode & " "
  & rsCourses!CrsNum
rsCourses.MoveNext
Loop
```

- This code is explained in Figure 14.3.

FIGURE 14.3: Create a program to loop through the records in a Recordset object.



`EOF` is a property of the recordset. It is true if the record counter has reached the "end of file" (EOF) marker and false otherwise.

The exclamation mark (!) indicates that `DeptCode` is a user-defined field (rather than a method or property) of the recordset object.

Since the Value property is the default property of a field, you do not have to use the `<recordset>!<field>.Value` syntax.

The `MoveNext` method moves the record counter to the next record in the recordset.

14.3.4 Using the `FindFirst` method

In this section, you will use the `FindFirst` method of Recordset objects to lookup a specific value in a table.

- Create a new function called `MyLookUp()` using the following declaration:

```
Function MyLookUp(strField As
  String, strTable As String,
  strWhere As String) As String
```

An example of how you would use this function is to return the `Title` of a course from the `Courses` table with a particular `DeptCode` and `CrsNum`. In other words, `MyLookUp()` is essentially an SQL statement without the `SELECT`, `FROM` and `WHERE` clauses.

The parameters of the function are used to specify the name of the table (a string), the name of the field (a string) from which you want the value, and a `WHERE` condition (a string) that ensures that only one record is found.

For example, to get the `Title` of COMM 351 from the `Courses` table, you would provide `MyLookUp()` with the following parameters:

1. "`Title`" — a string containing the name of the field from which we want to return a value;
2. "`Course`" — a string containing the name of the source table; and,
3. "`DeptCode = 'COMM' AND CrsNum = '335'`" — a string that contains the entire WHERE clause for the search.

Note that both single and double quotation marks must be used to signify a string within a string. The use of quotation marks in this manner is consistent with standard practice in English. For example, the sentence: "`He shouted, 'Wait for me.'`" illus-

trates the use of single quotes within double quotes.

- Define the `MyLookUp()` function as follows:

```
Dim dbCurr As DATABASE
Set dbCurr = CurrentDb
```

If you are using version 2.0, you cannot use the `CurrentDb` method to return a pointer to the current database. You must use long form (i.e., `Set dbCurr = DBEngine…`)

```
Dim rsRecords As Recordset
Set rsRecords =
 dbCurr.OpenRecordset(strTable,
 dbOpenDynaset)
```

In version 2.0, the name of some of the predefined constants are different. As such, you must use `DB_OPEN_DYNASET` rather than `dbOpenDynaset` to specify the type of Recordset object to be opened (the `FindFirst` method only works with "dynaset" type recordsets, hence the need to include the additional parameter in this segment of code).

```
rsRecords.FindFirst strWhere
```

VBA uses a rather unique convention to determine whether to enclose the arguments of a function, subroutine, or method in parentheses: if the procedure returns a value, enclose the parameters in parentheses; otherwise, use no parentheses. For example, in the line above, `strWhere` is a parameter of the `FindFirst` method (which does not return a value).

```
If Not rsRecords.NoMatch() Then
MyLookUp =
 rsRecords.Fields(strField).Value
```

```
Else
MyLookUp = ""
End If
```

- Execute the function with the following statement (see Figure 14.4):

```
? MyLookUp("Title", "Courses",
 "DeptCode = 'COMM' AND CrsNum =
 '351'")
```

As it turns out, what you have implemented exists already in Access in the form of a predefined function called `DLookUp()`.

- Execute the `DLookUp()` function by calling it in the same manner in which you called `MyLookUp()`.

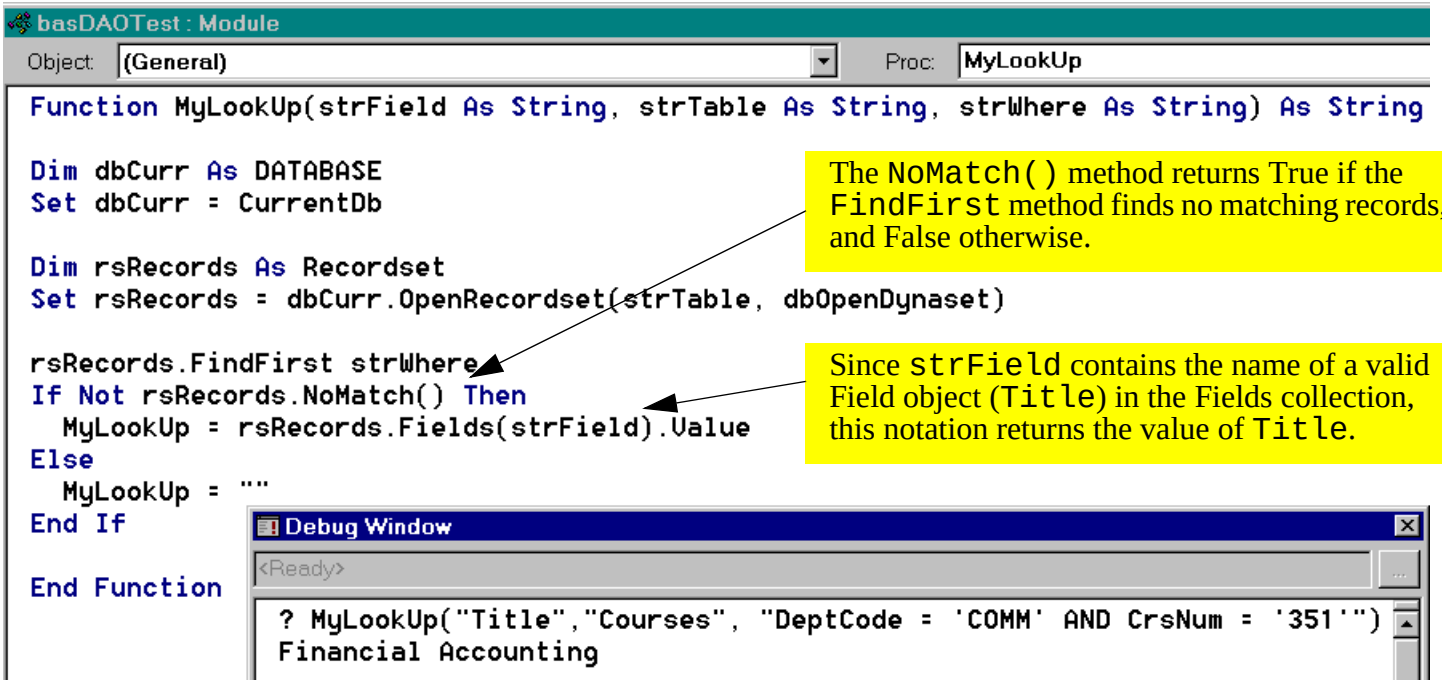
14.3.5 The `DLookUp()` function

The `DLookUp()` function is the "tool of last resort" in Access. Although you normally use queries and recordsets to provide you with the information you need in your application, it is occasionally necessary to perform a stand-alone query—that is, to use the `DLookUp()` function to retrieve a value from a table or query.

When using `DLookUp()` for the first few times, the syntax of the function calls may seem intimidating. But all you have to remember is the meaning of a handful of constructs that you have already used. These constructs are summarized below:

- **Functions** — `DLookUp()` is a function that returns a value. It can be used in the exact same manner as other functions, e.g.,
  `x = DLookUp(…)` is similar to
  `x = cos(2*pi)`.
- **Round brackets** `()` — In Access, round brackets have their usual meaning when grouping together operations, e.g., `3*(5+1)`. Round brackets are also used to enclose the arguments of function calls, e.g., `x = cos(2*pi)`.

FIGURE 14.4: `MyLookUp()`**: A function to find a value in a table.**



- **Square brackets [ ]** — Square brackets are not a universally defined programming construct like round brackets. As such, square brackets have a particular meaning in Access/VBA and this meaning is specific to Microsoft products. Simply put, square brackets are used to signify the name of a field, table, or other object in the DAO hierarchy—they have no other meaning. Square brackets are mandatory when the object names contain spaces, but optional otherwise. For example, `[Forms]![frmCourses]![DeptCode]` is identical to `Forms!frmCourses!DeptCode`.
- **Quotation marks** " " — Double quotation marks are used to distinguish literal strings from names of variables, fields, etc. For example, `x = "COMM"` means that the variable x is equal to the string of characters *COMM*. In contrast, `x = COMM` means that the variable x is equal to the value of the variable `COMM`.
- **Single quotation marks** ' ' — Single quotation marks have only one purpose: to replace normal quotation marks when two sets of quotation marks are nested. For example, the expression `x = "[ProductID] = '123'"` means that the variable x is equal to the string *ProductID = "123"*. In other words, when the expression is evaluated, the single quotes are replaced with double quotes. If you attempt to nest two sets of double quotation marks (e.g., `x = "[ProductID] = "123""`) the meaning is ambiguous and Access returns an error.
- **The Ampersand** & — The ampersand is the concatenation operator in Access/VBA and is unique to Microsoft products. The concatenation operator joins two strings of text together into one string of text. For example,

`x = "one" & "_two"` means that the variable `x` is equal to the string *one_two*.

If you understand these constructs at this point, then understanding the `DLookUp()` function is just a matter of putting the pieces together one by one.

14.3.5.1 Using `DLookUp()` in queries

The `DLookUp()` function is extremely useful for performing lookups when no relationship exists between the tables of interest. In this section, you are going to use the `DLookUp()` function to lookup the course name associated with each section in the `Sections` table. Although this can be done much easier using a join query, this exercise illustrates the use of variables in function calls.

- Create a new query called `qryLookUpTest` based on the `Sections` table.
- Project the `DeptCode`, `CrsNum`, and `Section` fields.
- Create a calculated field called `Title` using the following expression (see Figure 14.5):

```
Title: DLookUp("Title", "Courses",
 "DeptCode = '"& [DeptCode] & "' AND
 CrsNum = '" & [CrsNum] & "'")
```

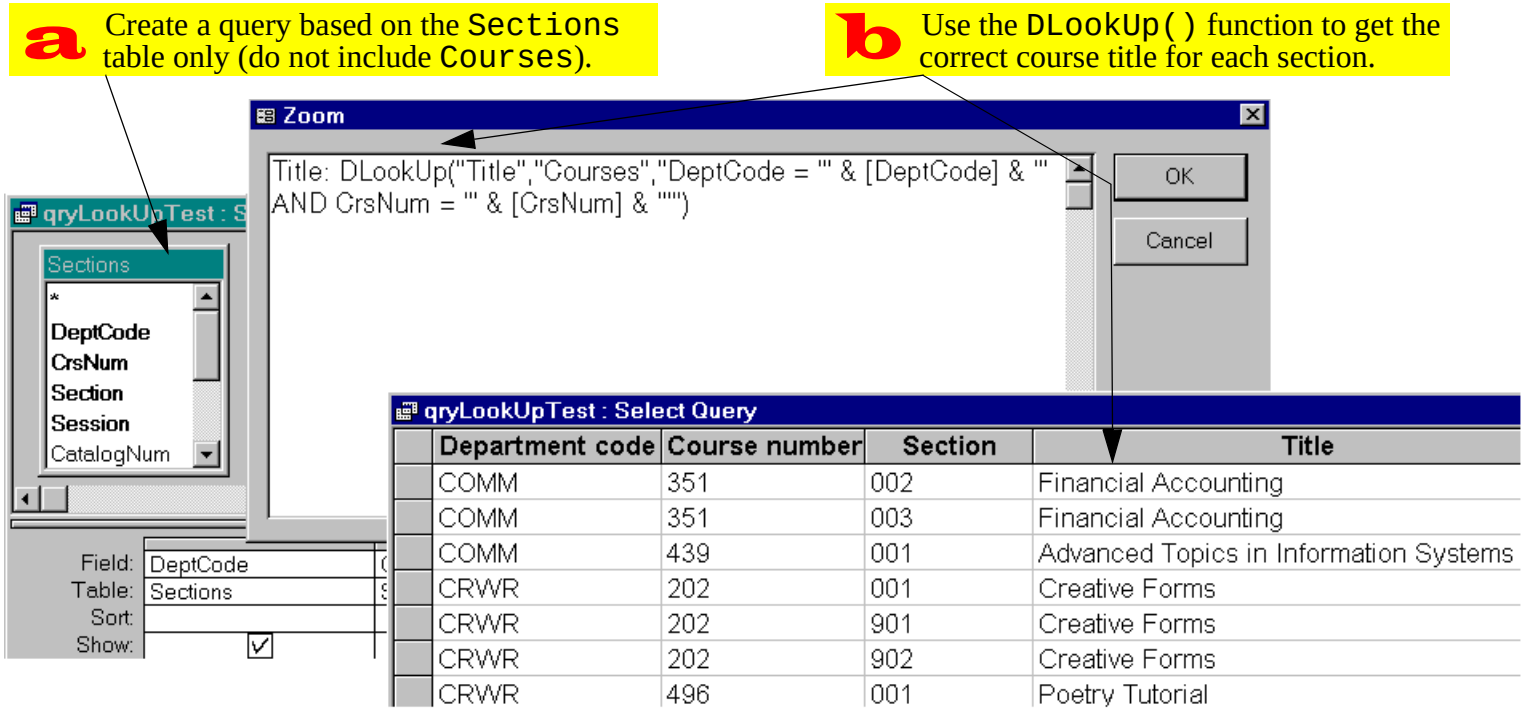
14.3.5.2 Understanding the WHERE clause

The first two parameters of the `DLookUp()` are straightforward: they give the name of the field and the table containing the information of interest. However, the third argument (i.e., the `WHERE` clause) is more complex and requires closer examination.

At its core, this `WHERE` clause is similar to the one you created in Section 5.3.2 in that it contains two criteria. However, there are two important differences:

1. Since it is a `DLookUp()` parameter, the entire clause must be enclosed within quotation marks. This means single and double quotes-within-quotes must be used.

FIGURE 14.5: Create a query that uses `DLookUp()`.



2. It contains variable (as opposed to literal) criteria. For example, `[DeptCode]` is used instead of `"COMM"`. This makes the value returned by the function call dependent on the current value of the `DeptCode` field.

In order to get a better feel for syntax of the function call, do the following exercises (see Figure 14.6):

Switch to the debug window and define two string variables (see Section 12.3.1 for more information on using the debug window):

```
strDeptCode = "COMM"
strCrsNum = "351"
```

These two variables will take the place the field values while you are in the debug window.

- Write the `WHERE` clause you require without the variables first. This provides you with a template for inserting the variables.
- Assign the `WHERE` clause to a string variable called `strWhere` (this makes it easier to test).
- Use `strWhere` in a `DLookUp()` call.

14.4 Discussion

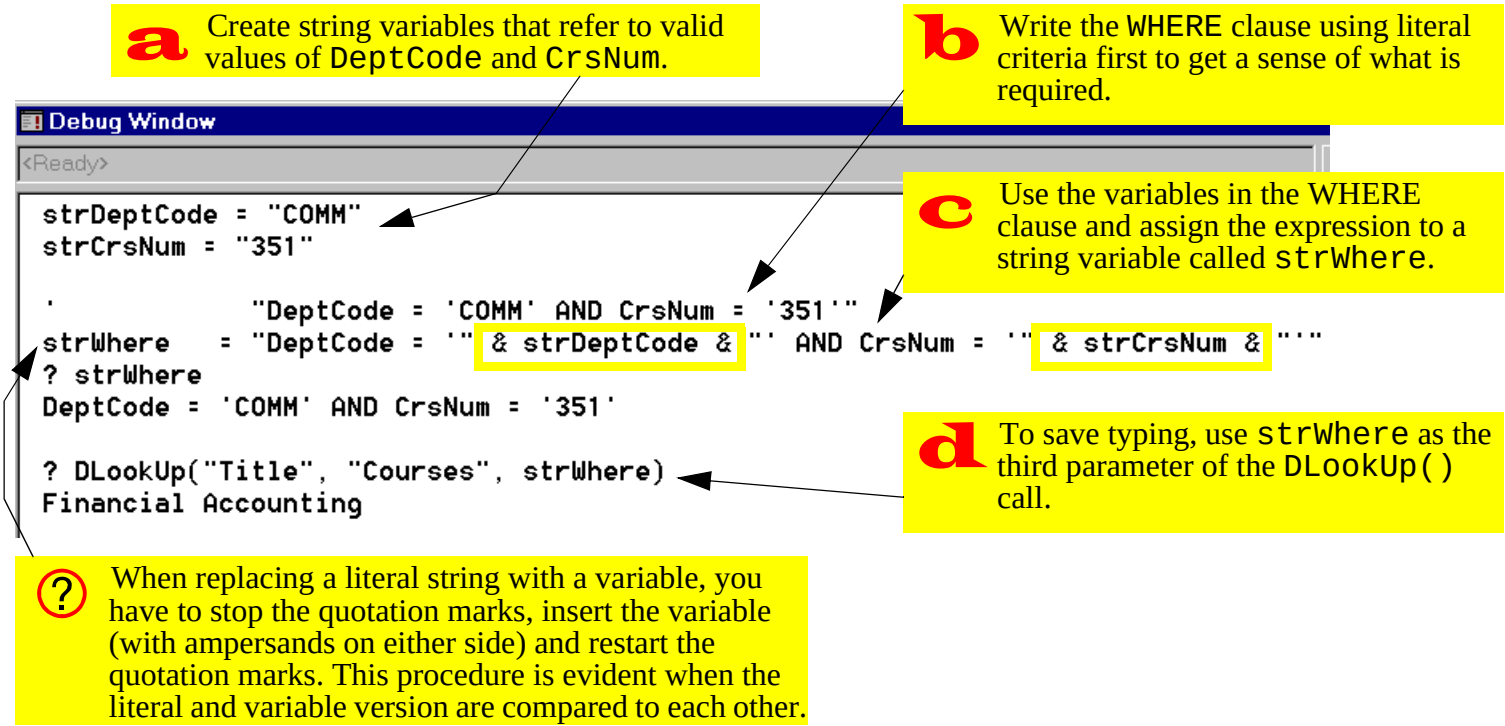
14.4.1 VBA versus SQL

The `PrintRecords` procedure you created in Section 14.3.3 is interesting since it does essentially the same thing as a select query: it displays a set of records.

You could extend the functionality of the `Print-Records` subroutine by adding an argument and an `IF-THEN` condition. For example:

```
Sub PrintRecords(strDeptCode as
 String)
Do Until rsCourses.EOF
If rsCourses!DeptCode = strDeptCode
 Then
Debug.Print rsCourses!DeptCode & " "
 & rsCourses!CrsNum
```

FIGURE 14.6: Examine the syntax of the WHERE clause.



```
End If
rsCourses.MoveNext
Loop
rsCourses.Close
End Sub
```

This subroutine takes a value for `DeptCode` as an argument and only prints the courses in that particular department. It is equivalent to the following SQL command:

```
SELECT DeptCode, CourseNum FROM
 Courses WHERE DeptCode =
 strDeptCode
```

14.4.2 Procedural versus Declarative

The difference between extracting records with a query language and extracting records with a programming language is that the former approach is **declarative** while the latter is **procedural**.

SQL and QBE are declarative languages because you (as a programmer) need only tell the computer *what* you want done, not *how* to do it. In contrast, VBA is a procedural language since you must tell the computer exactly how to extract the records of interest.

Although procedural languages are, in general, more flexible than their declarative counterparts, they rely a great deal on knowledge of the underlying structure of the data. As a result, procedural languages tend to be inappropriate for end-user development (hence the ubiquity of declarative languages such as SQL in business environments).

14.5 Application to the assignment

14.5.1 Using a separate table to store system parameters

When you calculated the tax for the order in Section 9.5, you "hard-coded" the tax rate into the form. If the tax rate changes, you have to go through all the forms that contain a tax calculation, find the hard-coded value, and change it. Obviously, a better approach is to store the tax rate information in a table and use the value from the table in all form-based calculations.

Strictly speaking, the tax rate for each product is a property of the product and should be stored in the `Products` table. However, in the wholesaling environment used for the assignment, the assumption is made that all products are taxed at the same rate.

As a result, it is possible to cheat a little bit and create a stand-alone table (e.g., `SystemVariables`) that contains a single record:

VariableName	Value
GST	0.07

Of course, other system-wide variables could be contained in this table, but one is enough for our purposes. The important thing about the `SystemVariables` table is that it has absolutely no relationship with any other table. As such, you must use a `DLookUp()` to access this information.

- Create a table that contains information about the tax rate.
- Replace the hard-coded tax rate information in your application with references to the value in the table (i.e., use a `DLookUp()` in your tax calculations). Although the `SystemVariables` table only contains one record at this point, you

should use an appropriate `WHERE` clause to ensure that the value for GST is returned (if no `WHERE` clause is provided, `DLookUp()` returns the first value in the table).

The use of a table such as `SystemVariables` contradicts the principles of relational database design (we are creating an attribute without an entity). However, trade-offs between theoretical elegance and practicality are common in any development project.

14.5.2 Determining outstanding backorders

An good example in your assignment of a situation requiring use of the `DLookUp()` is determining the backordered quantity of a particular item for a particular customer. You need this quantity in order to calculate the number of each item to ship.

The reason you must use a `DLookUp()` to get this information is that there is no relationship between the `OrderDetails` and `BackOrders` tables.

Any relationship that you manage to create between `OrderDetails` and `BackOrders` will be nonsensical and result in a non-updatable recordset.

- In the query underlying your `OrderDetails` subform, create a calculated field called `QtyOnBackOrder` to determine the number of items on backorder for each item added to the order. This calculated field will use the `DLookUp()` function.

There are two differences between this `DLookUp()` and the one you did in Section 14.3.5.1

1. Both of the variables used in the function (e.g., `CustID` and `ProductID`) are not in the query. As such, you will have to use a join to bring the

missing information into the query.

2. `ProductID` is a text field and the criteria of text fields must be enclosed in quotation marks, e.g.:
   `ProductID = "123"`
   However, `CustID` is a numeric field and the criteria for numeric fields is not enclosed in quotations marks, e.g.:
   `CustID = 4`.

Not every combination of `CustID` and `ProductID` will have an outstanding backorder. When a matching records is not found, the `DLookUp()` function returns a special value: `Null`. The important thing to remember is that `Null` plus or minus anything equals `Null`. This has implications for your "quantity to ship" calculation.

- Create a second calculated field in your query to convert any `Null`s in the first calculated field to zero. To do this, use the `iif()` and `IsNull()` functions, e.g.:

```
QtyOnBackOrderNoNull:
  iif(IsNull([QtyOnBackOrder]),0,[Qty
  OnBackOrder])
```

- Use this "clean" version in your calculations and on your form.

It is possible to combine these two calculated fields into a one-step calculation, e.g.:
`iif(IsNull(DLookUp(…)),0,`
`DLookUp(…))`.
The problem with this approach is that the `DLookUp()` function is called twice: once to test the conditional part of the immediate if statement and a second time to provide the "false" part of the statement. If the `BackOrders` table is very large, this can result in an unacceptable delay when displaying data in the form.