Access Tutorial 12: An Introduction to Visual Basic

12.1 Introduction: Learning the basics of programming

Programming can be an enormously complex and difficult activity. Or it can be quite straightforward. In either case, the basic programming concepts remain the same. This tutorial is an introduction to a handful of programming constructs that apply to any "third generation" language, not only Visual Basic for Applications (VBA).

Strictly speaking, the language that is included with Access is not Visual Basic—it is a subset of the full, stand-alone Visual Basic language (which Microsoft sells separately). In Access version 2.0, the subset is called "Access Basic". In version 7.0, it is slightly enlarged subset called "Visual Basic for Applications" (VBA). However, in the context of the simple programs we are writing here, these terms are interchangeable.

12.1.1 Interacting with the interpreter

Access provides two ways of interacting with the VBA language. The most useful of these is through saved modules that contain VBA procedures. These procedures (subroutines and functions) can be run to do interesting things like process transactions against master tables, provide sophisticated error checking, and so on.

The second way to interact with VBA is directly through the interpreter. Interpreted languages are easier to experiment with since you can invoke the interpreter at any time, type in a command, and watch it execute. In the first part of this tutorial, you are going to invoke Access' VBA interpreter and execute some very simple statements.

Last update: 25-Aug-1997

In the second part of the tutorial, you are going to create a couple of VBA modules to explore looping, conditional branching, and parameter passing.

12.2 Learning objectives

- What is the debug/immediate window? How do I invoke it?
- What are statements, variables, the assignment operator, and predefined functions?
- How do I create a module containing VBA code?
- What are looping and conditional branching? What language constructs can I use to implement them?
- How do I use the debugger in Access?
- What is the difference between an interpreted and compiled programming language?

12.3 Tutorial exercises

12.3.1 Invoking the interpreter

- Click on the module tab in the database window and press *New*.

This opens the module window which we will use in Section 12.3.3. You have to have a module window open in order for the debug window to be available from the menu.

- Select *View > Debug Window* from the main menu. Note that *Control-G* can be used in version 7.0 and above as a shortcut to bring up the debug window.

In version 2.0, the "debug" window is called the "immediate" window. As such, you have to use *View > Immediate Window*. The term debug window will be used throughout this tutorial.

12.3.2 Basic programming constructs

In this section, we are going to use the debug window to explore some basic programming constructs.

12.3.2.1 Statements

Statements are special keywords in a programming language that do something when executed. For example, the `Print` statement in VBA prints an expression on the screen.

- In the debug window, type the following:
  `Print “Hello world!”↵`

(the ↵ symbol at the end of a line means “press the *Return* or *Enter* key”).

In VBA (as in all dialects of BASIC), the question mark (?) is typically used as shorthand for the `Print` statement. As such, the statement: `? “Hello world!”↵` is identical to the statement above.

12.3.2.2 Variables and assignment

A variable is space in memory to which you assign a name. When you use the variable name in expressions, the programming language replaces the variable name with the contents of the space in memory at that particular instant.

- Type the following:
  ```
  s = “Hello”↵
  ? s & “ world”↵
  ? “s” & “ world”↵
  ```

In the first statement, a variable `s` is created and the string `Hello` is assigned to it. Recall the function of the concatenation operator (&) from Section 4.4.2.

Contrary to the practice in languages like C and Pascal, the equals sign (=) is used to **assign** values to variables. It is also used as the equivalence operator (e.g., does `x = y`?).

When the second statement is executed, VBA recognizes that `s` is a variable, not a string (since it is not in quotations marks). The interpreter replaces `s` with its value (`Hello`) before executing the `Print` command. In the final statement, `s` is in quotation marks so it is interpreted as a **literal string**.

Within the debug window, any string of characters in quotations marks (e.g., `“COMM”`) is interpreted as a literal string. Any string without quotation marks (e.g., `COMM`) is interpreted as a variable (or a field name, if appropriate). Note, however, that this convention is not universally true within different parts of Access.

12.3.2.3 Predefined functions

In computer programming, a function is a small program that takes one or more **arguments** (or **parameters**) as input, does some processing, and returns a value as output. A *predefined* (or *built-in*) function is a function that is provided as part of the programming environment.

For example, `cos(x)` is a predefined function in many computer languages—it takes some number `x` as an argument, does some processing to find its cosine, and returns the answer. Note that since this function is predefined, you do not have to know anything about the algorithm used to find the cosine, you just have to know the following:

1. what to supply as inputs (e.g., a valid numeric expression representing an angle in radians),
2. what to expect as output (e.g., a real number between -1.0 and 1.0).

The on-line help system provides these two pieces of information (plus a usage example and some additional remarks) for all VBA predefined functions.

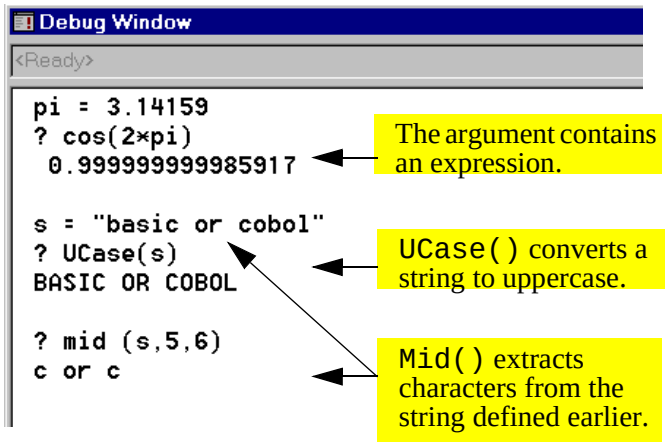
In this section, we are going to explore some basic predefined functions for working with numbers and text. The results of these exercises are shown in Figure 12.1.

- Print the cosine of 2π radians:
  ```
  pi = 3.14159↵
  ? cos(2*pi)↵
  ```
- Convert a string of characters to uppercase:
  ```
  s = “basic or cobol”↵
  ? UCase(s)↵
  ```
- Extract the middle six characters from a string starting at the fifth character:
  ```
  ? mid (s,5,6)↵
  ```

12.3.2.4 Remark statements

When creating large programs, it is considered good programming practice to include adequate internal documentation—that is, to include comments to explain what the program is doing.

FIGURE 12.1: Interacting with the Visual Basic interpreter.



Comment lines are ignored by the interpreter when the program is run. To designate a comment in VBA, use an apostrophe to start the comment, e.g.:

```
‘ This is a comment line!
Print “Hello” ‘the comment starts
  here
```

The original REM (remark) statement from BASIC can also be used, but is less common.

```
REM This is also a comment (remark)
```

12.3.3 Creating a module

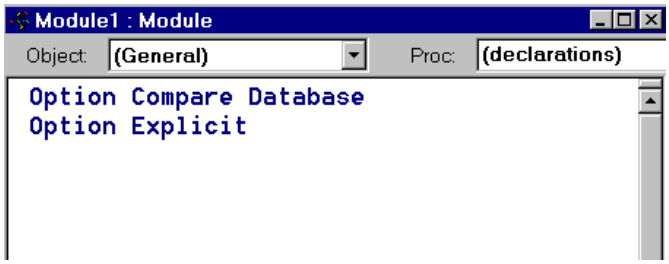
- Close the debug window so that the declaration page of the new module created in Section 12.3.3 is visible (see Figure 12.2).

The two lines:

```
Option Compare Database
Option Explicit
```

are included in the module by default. The `Option Compare` statement specifies the way in which strings are compared (e.g., does uppercase/ lower-case matter?). The `Option Explicit` statement forces you to declare all your variables before using them.

FIGURE 12.2: The declarations page of a Visual Basic module.



In version 2.0, Access does not add the `Option Explicit` statement by default. As such you should add it yourself.

A module contains a declaration page and one or more pages containing **subroutines** or user-defined **functions**. The primary difference between subroutines and functions is that subroutines simply execute whereas functions are expected to return a value (e.g., `cos()`). Since only one subroutine or function shows in the window at a time, you must use the *Page Up* and *Page Down* keys to navigate the module.

The VBA editor in version 8.0 has a number of enhancements over earlier version, including the capability of showing multiple functions and subroutines on the same page.

12.3.4 Creating subroutines with looping and branching

In this section, you will explore two of the most powerful constructs in computer programming: **looping** and **conditional branching**.

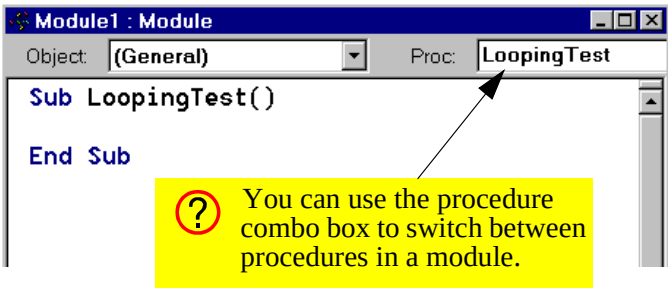
- Create a new subroutine by typing the following anywhere on the declarations page:
  `Sub LoopingTest()↵`

Notice that Access creates a new page in the module for the subroutine, as shown in Figure 12.3.

12.3.4.1 Declaring variables

When you declare a variable, you tell the programming environment to reserve some space in memory for the variable. Since the amount of space that is required is completely dependent on the type of data the variable is going to contain (e.g., string, integer, Boolean, double-precision floating-point, etc.), you

FIGURE 12.3: Create a new subroutine.



have to include data type information in the declaration statement.

In VBA, you use the `Dim` statement to declare variables.

- Type the following into the space between the `Sub... End Sub` pair:

```
Dim i as integer
Dim s as string
```

- Save the module as `basTesting`.

One of the most useful looping constructs is `For <condition>... Next`. All statements between the `For` and `Next` parts are repeated as long as the `<condition>` part is true. The index `i` is automatically incremented after each iteration.

- Enter the remainder of the `LoopingTest` program:

```
s = "Loop number: "
For i = 1 To 10
   Debug.Print s & i
Next i
```

- Save the module.

It is customary in most programming languages to use the *Tab* key to indent the elements within a loop slightly. This makes the program more readable.

Note that the `Print` statement within the subroutine is prefaced by `Debug`. This is due to the object-oriented nature of VBA which will be explored in greater detail in Tutorial 14.

12.3.4.2 Running the subroutine

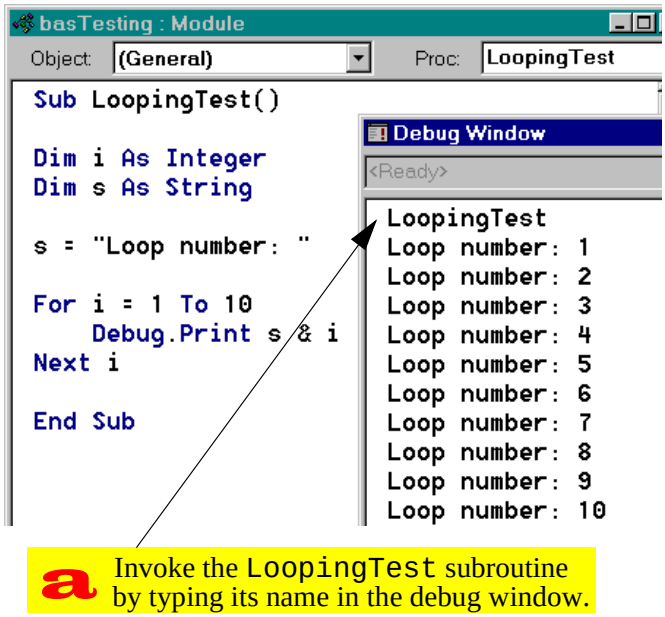
Now that you have created a subroutine, you need to run it to see that it works. To invoke a subroutine, you simply use its name like you would any statement.

- Select *View > Debug Window* from the menu (or press *Control-G* in version 7.0).
- Type: `LoopingTest`↵ in the debug window, as shown in Figure 12.4.

12.3.4.3 Conditional branching

We can use a different looping construct, `Do Until <condition>... Loop`, and the conditional branching construct, `If <condition> Then... Else`, to achieve the same result.

FIGURE 12.4: Run the `LoopingTest` subroutine in the debug window.



- Type the following anywhere under the `End Sub` statement in order to create a new page in the module:

```
Sub BranchingTest↵
```

- Enter the following program:

```
Dim i As Integer
Dim s As String
Dim intDone As Integer
s = "Loop number: "
i = 1
intDone = False
Do Until intDone = True
   If i > 10 Then
      Debug.Print "All done"
      intDone = True
   Else
      Debug.Print s & i
      i = i + 1
   End If
   Loop
```

- Run the program

12.3.5 Using the debugger

Access provides a rudimentary debugger to help you step through your programs and understand how they are executing. The two basic elements of the debugger used here are **breakpoints** and **stepping** (line-by-line execution).

- Move to the `s = "Loop number: "` line in your `BranchingTest` subroutine and select *Run > Toggle Breakpoint* from the menu (you can also press *F9* to toggle the breakpoint on a particular line of code).

Note that the line becomes highlighted, indicating the presence of an active breakpoint. When the program runs, it will suspend execution at this breakpoint and pass control of the program back to you.

- Run the subroutine from the debug window, as shown in Figure 12.5.
- Step through a couple of lines in the program line-by-line by pressing *F8*.

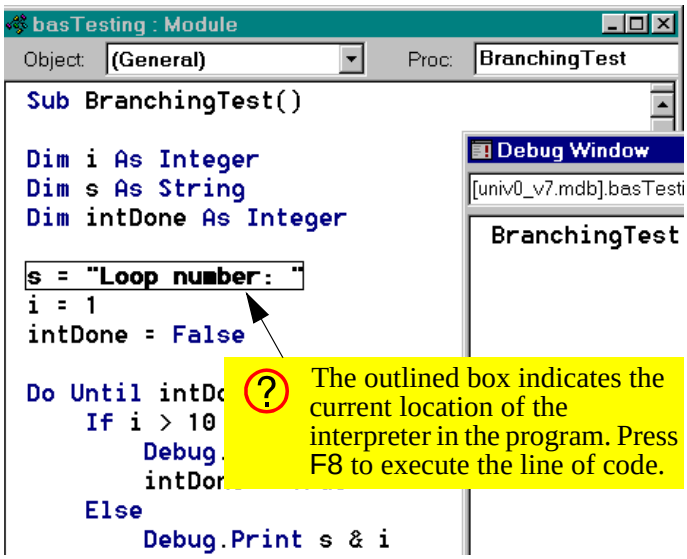
By stepping through a program line by line, you can usually find any program bugs. In addition, you can use the debug window to examine the value of variables while the program's execution is suspended.

- click on the debug window and type
  `? i↵`
  to see the current value of the variable `i`.

12.3.6 Passing parameters

In the `BranchingTest` subroutine, the loop starts at 1 and repeats until the counter `i` reaches 10. It may be preferable, however, to set the start and finish quantities when the subroutine is called from the debug window. To achieve this, we have to pass **parameters** (or **arguments**) to the subroutine.

FIGURE 12.5: Execution of the subroutine is suspended at the breakpoint.



The main difference between passed parameters and other variables in a procedure is that passed parameters are declared in the first line of the subroutine definition. For example, following subroutine declaration

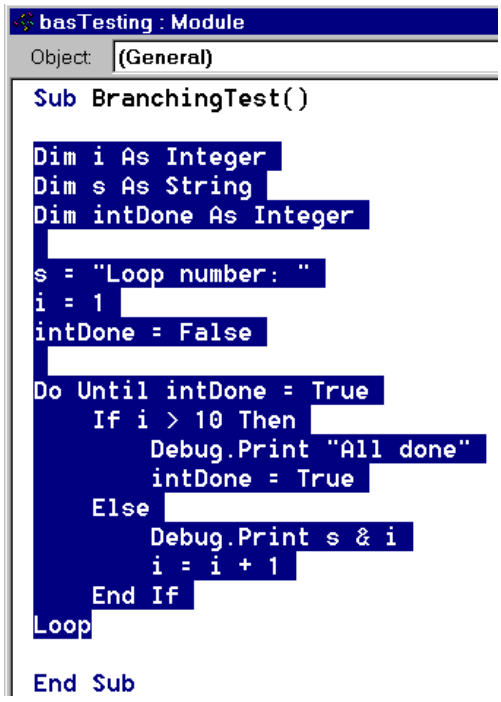
```
Sub BranchingTest(intStart as
  Integer, intStop as Integer)
```

not only declares the variables `intStart` and `intStop` as integers, it also tells the subroutine to expect these two numbers to be passed as parameters.

To see how this works, create a new subroutine called `ParameterTest` based on `BranchingTest`.

- Type the declaration statement above to create the `ParameterTest` subroutine.
- Switch back to `BranchingTest` and highlight all the code except the `Sub` and `End Sub` statements, as shown in Figure 12.6.

FIGURE 12.6: Highlight the code to copy it.



- Copy the highlighted code to the clipboard (*Control-Insert*), switch to `ParameterTest`, and paste the code (*Shift-Insert*) into the `ParameterTest` procedure.

To incorporate the parameters into `ParameterTest`, you will have to make the following modifications to the pasted code:

- Replace `i = 1` with `i = intStart`.
- Replace `i > 10` with `i > intStop`.
- Call the subroutine from the debug window by typing:
  `ParameterTest 4, 12↵`

If you prefer enclosing parameters in brackets, you have to use the `Call <sub name>(parameter`$_1$`, ..., parameter`$_n$`)` syntax. For example:
`Call ParameterTest(4,12)↵`

12.3.7 Creating the `Min()` function

In this section, you are going to create a user-defined function that returns the minimum of two numbers. Although most languages supply such a function, Access does not (the `Min()` and `Max()` function in Access are for use within SQL statements only).

- Create a new module called `basUtilities`.
- Type the following to create a new function:
  `Function MinValue(n1 as Single, n2 as Single) as Single↵`

This defines a function called `MinValue` that returns a single-precision number. The function requires two single-precision numbers as parameters.

Since a function returns a value, the data type of the return value should be specified in the function declaration. As such, the basic syntax of a function declaration is:


```
Function <function name>(parameter1 As <data type>, …, parametern As <data type>) As <data type>
```

The function returns a variable named `<function name>`.

- Type the following as the body of the function:

```
If n1 <= n2 Then
   MinValue = n1
Else
   MinValue = n2
End If
```

- Test the function, as shown in Figure 12.7.

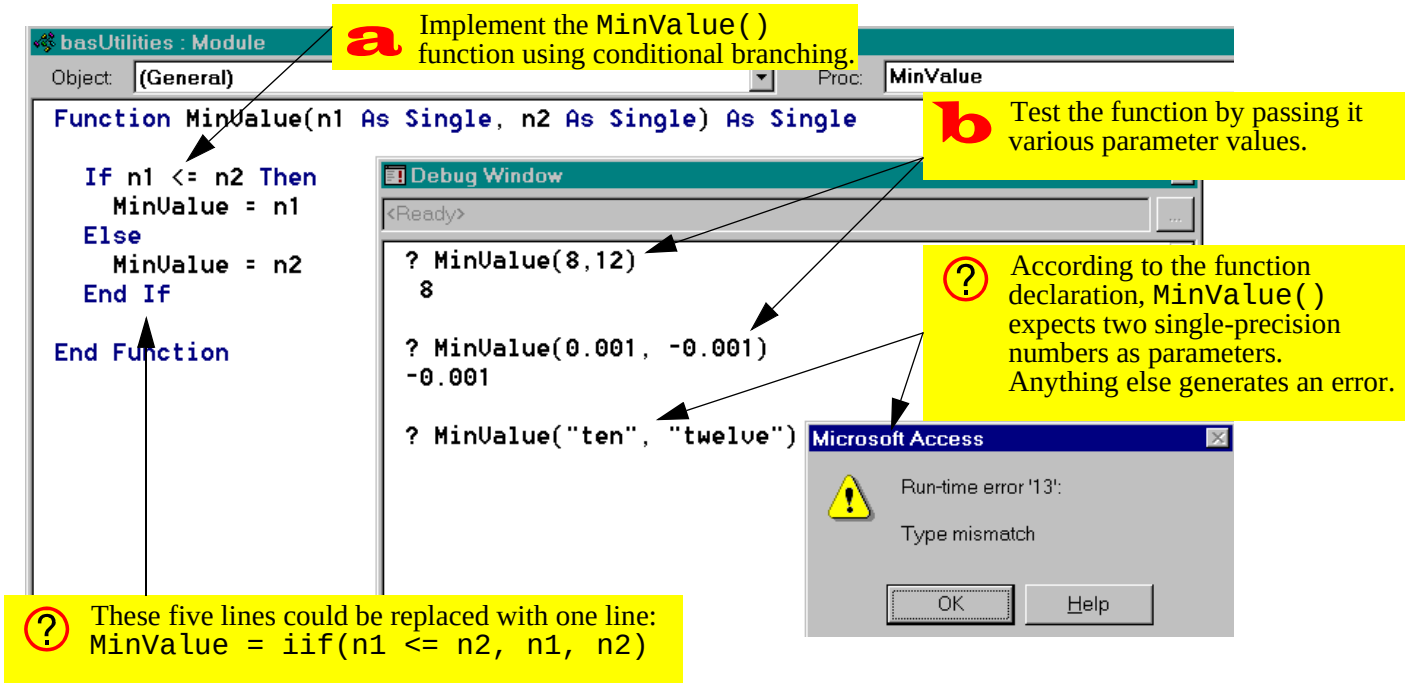
12.4 Discussion

12.4.1 Interpreted and compiled languages

VBA is an **interpreted language**. In interpreted languages, each line of the program is interpreted (converted into machine language) and executed when the program is run. Other languages (such as C, Pascal, FORTRAN, etc.) are **compiled**, meaning that the original (source) program is translated and saved into a file of machine language commands. This executable file is run instead of the source code.

Predictably, compiled languages run much faster then interpreted languages (e.g., compiled C++ is generally ten times faster than interpreted Java). However, interpreted languages are typically easier to learn and experiment with.

FIGURE 12.7: Testing the `MinValue()` function.



12.5 Application to the assignment

You will need a `MinValue()` function later in the assignment when you have to determine the quantity to ship.

- Create a `basUtilities` module in your assignment database and implement a `MinValue()` function.

To ensure that no confusion arises between your user-defined function and the built-in SQL `Min()` function, do not call you function `Min()`.

Access Tutorial 13: Event-Driven Programming Using Macros

13.1 Introduction: What is event-driven programming?

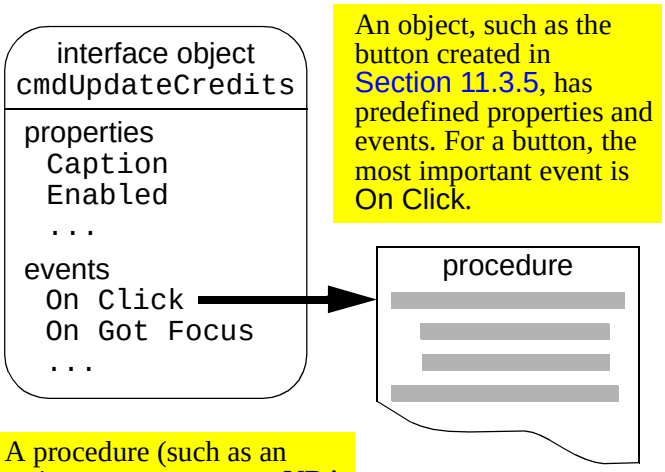
In conventional programming, the sequence of operations for an application is determined by a central controlling program (e.g., a main procedure). In **event-driven** programming, the sequence of operations for an application is determined by the user's interaction with the application's interface (forms, menus, buttons, etc.).

For example, rather than having a main procedure that executes an order entry module followed by a data verification module followed by an inventory update module, an event-driven application remains in the background until certain events happen: when a value in a field is modified, a small data verification program is executed; when the user indicates that the order entry is complete, the inventory update module is executed, and so on.

Event-driven programming, graphical user interfaces (GUIs), and object-orientation are all related since forms (like those created in Tutorial 6) and the graphical interface objects on the forms serve as the skeleton for the entire application. To create an event-driven application, the programmer creates small programs and attaches them to events associated with objects, as shown in Figure 13.1. In this way, the behavior of the application is determined by the interaction of a number of small manageable programs rather than one large program.

Last update: 25-Aug-1997

FIGURE 13.1: In a trigger, a procedure is attached to an event.



13.1.1 Triggers

Since events on forms "trigger" actions, event/procedure combinations are sometimes called **triggers**.

For example, the action query you attached to a button in Section 11.3.5 is an example of a simple, one-action trigger. However, since an action query can only perform one type of action, and since you typically have a number of actions that need to be performed, macros or Visual Basic procedures are typically used to implement a triggers in Access.

13.1.2 The Access macro language

As you discovered in Tutorial 12, writing simple VBA programs is not difficult, but it is tedious and error-prone. Furthermore, as you will see in Tutorial 14, VBA programming becomes much more difficult when you have to refer to objects using the naming conventions of the database object hierarchy. As a consequence, even experienced Access program-

mers often turn to the Access macro language to implement basic triggers.

The macro language itself consists of 40 or so commands. Although it is essentially a procedural language (like VBA), the commands are relatively high level and easy to understand. In addition, the macro editor simplifies the specification of the **action arguments** (parameters).

13.1.3 The trigger design cycle

To create a trigger, you need to answer two questions:

1. What has to happen?
2. When should it happen?

Once you have answered the first question (“what”), you can create a macro (or VBA procedure) to execute the necessary steps. Once you know the answer to the second question (“when”), you can attach the procedure to the correct event of the correct object.

Selecting the correct object and the correct event for a trigger is often the most difficult part of creating an event-driven application. It is best to think about this carefully before you get too caught up in implementing the procedure.

13.2 Learning objectives

- ❒ What is event-driven programming? What is a trigger?
- ❒ How do I design a trigger?
- ❒ How does the macro editor in Access work?
- ❒ How do I attach a macro to an event?
- ❒ What is the SetValue action? How is it used?

- ❒ How do I make the execution of particular macro actions conditional?
- ❒ What is a switchboard and how do I create one for my application?
- ❒ How to I make things happen when the application is opened?
- ❒ What are the advantages and disadvantages of event-driven programming?

13.3 Tutorial exercises

In this tutorial, you will build a number of very simple triggers using Access macros. These triggers, by themselves, are not particularly useful and are intended for illustrative purposes only.

13.3.1 The basics of the macro editor

In this section, you are going to eliminate the warning messages that precede the trigger you created Section 11.3.5.

As such, the answer to the “what” question is the following:

1. Turn off the warnings so the dialog boxes do not pop up when the action query is executed;
2. Run the action query; and,
3. Turn the warnings back on (it is generally good programming practice to return the environment to its original state).

Since a number of things have to happen, you cannot rely on an action query by itself. You can, however, execute a macro that executes several actions including one or more action queries.

- Select the *Macros* tab from the database window and press *New*. This brings up the macro editor shown in Figure 13.2.
- Add the three commands as shown in Figure 13.3. Note that the `OpenQuery` command is used to run the action query.
- Save the macro as `mcrUpdateCredits` and close it.

13.3.2 Attaching the macro to the event

The answer to the "when" question is: When the `cmdUpdateCredits` button is pressed. Since you already created the button in Section 11.3.5, all you need to do is modify its *On Click* property to point the `mcrUpdateCredits` macro.

- Open `frmDepartments` in design mode.
- Bring up the property sheet for the button and scroll down until you find the *On Click* property, as shown in Figure 13.4.

FIGURE 13.4: Bring up the *On Click* property for the button.

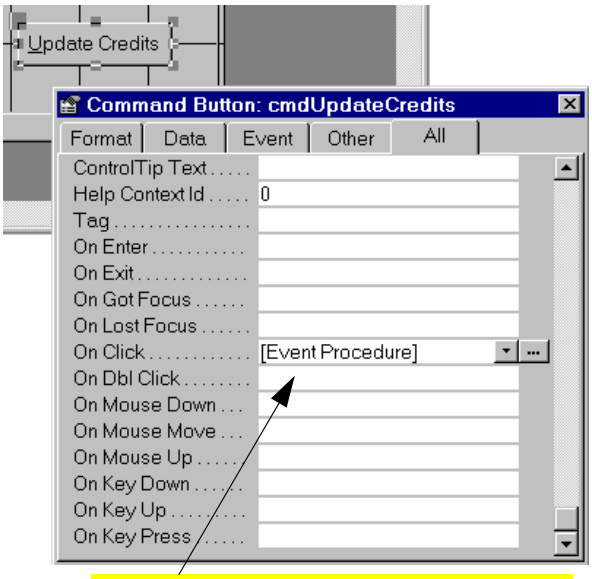


FIGURE 13.2: The macro editor.

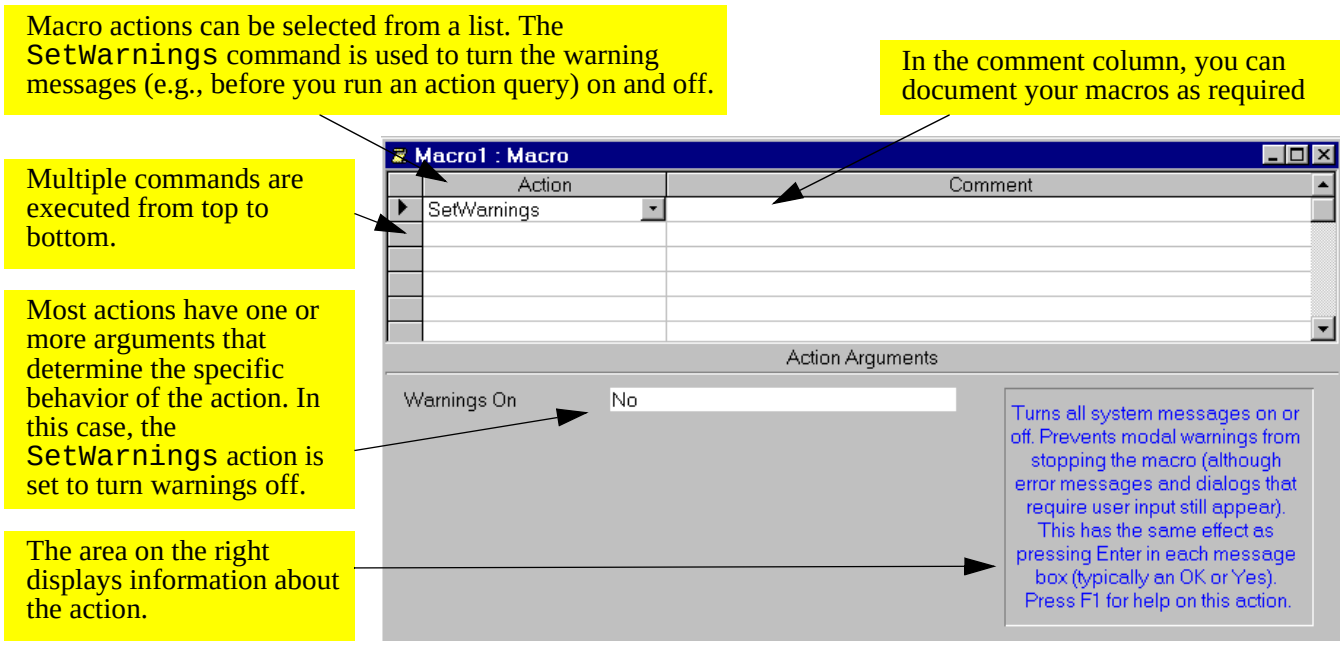
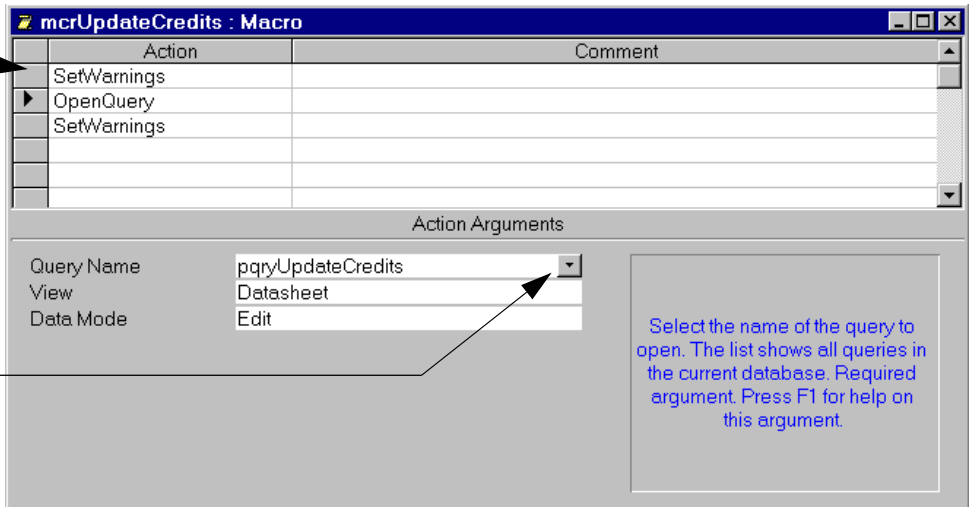


FIGURE 13.3: Create a macro that answers the "what" question.

a Add the three commands to the macro.

b The arguments for the two `SetWarnings` actions are straightforward. For the `OpenQuery` command, you can select the query to open (or run) from a list. Since this is an action query, the second and third arguments are not applicable.

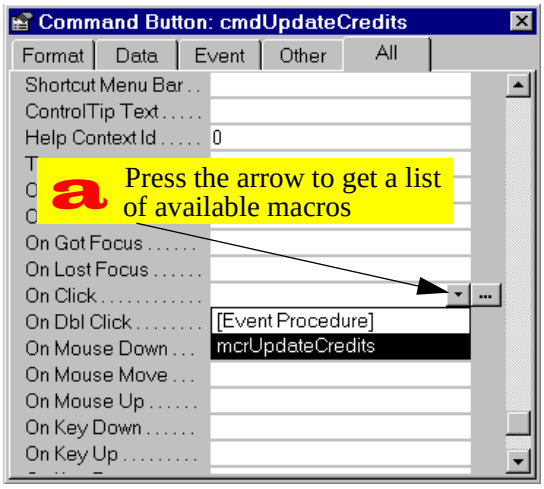


- Press the builder button (…) beside the existing procedure and look at the VBA subroutine created by the button wizard. Most of this code is for error handling.

? Unlike the stand-along VBA modules you created in Tutorial 12, this module (collection of functions and subroutines) is embedded in the `frmDepartments` form.

- Since you are going to replace this code with a macro, you do not want it taking up space in your database file. Highlight the text in the subroutine and delete it. When you close the module window, you will see the reference to the "event procedure" is gone.
- Bring up the list of choice for the *On Click* property as shown in Figure 13.5. Select `mcrUpdateCredits`.

FIGURE 13.5: Select the macro to attach to the *On Click* property.



- Switch to form view and press the button. Since no warnings appear, you may want to press the button a few times (you can always use your roll-back query to reset the credits to their original values).

13.3.3 Creating a check box to display update status information

Since the warning boxes have been disabled for the update credits trigger, it may be useful to keep track of whether courses in a particular department have already been updated.

To do this, you can add a field to the `Departments` table to store this "update status" information.

- Edit the `Departments` table and add a *Yes/No* field called `CrUpdated`.

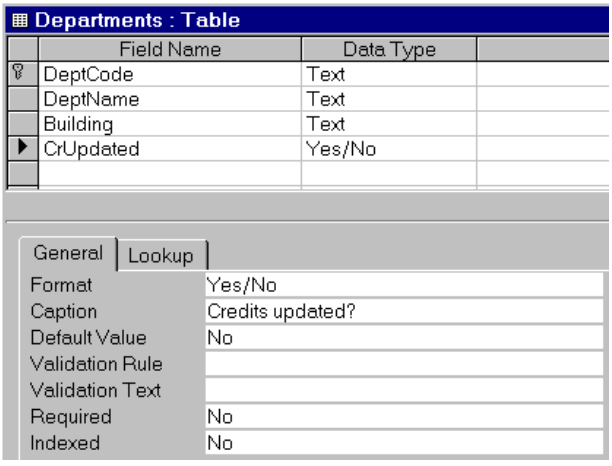
If you have an open query or form based on the `Departments` table, you will not be able to modify the structure of the table until the query or form is closed.

- Set the *Caption* property to `Credits updated?` and the *Default* property to `No` as shown in Figure 13.6.

Changes made to a table do not automatically carry over to forms already based on that table. As such, you must manually add the new field to the departments form.

- Open `frmDepartments` in design mode.
- Make sure the toolbox and field list are visible. Notice that the new field (`CrUpdated`) shows up in the field list.
- Use the same technique for creating combo boxes to create a bound check box control for the yes/no field. This is shown in Figure 13.7.

FIGURE 13.6: Add a field to the `Departments` table to record the status of updates.



13.3.4 The `SetValue` command

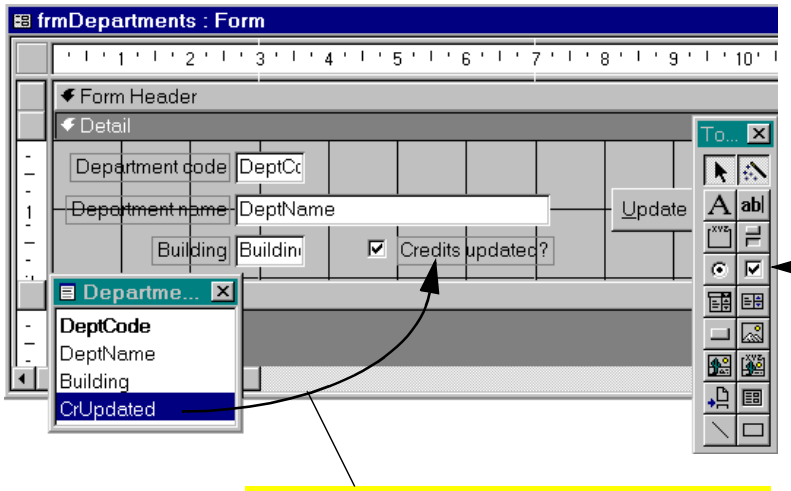
So far, you have used two commands in the Access macro language: `SetWarnings` and `OpenQuery`. In this section, you are going to use one of the most useful commands—`SetValue`—to automatically change the value of the `CrUpdated` check box.

- Open your `mcrUpdateCredits` macro in design mode and add a `SetValue` command to change the `CrUpdated` check box to `Yes` (or `True`, if you prefer). This is shown in Figure 13.8.
- Save the macro and press the button on the form. Notice that the value of the check box changes, reminding you not to update the courses for a particular department more than once.

13.3.5 Creating conditional macros

Rather than relying on the user not to run the update when the check box is checked, you may use a **conditional macro** to *prevent* an update when the check box is checked.

FIGURE 13.7: Add a check box control to keep track of the update status.



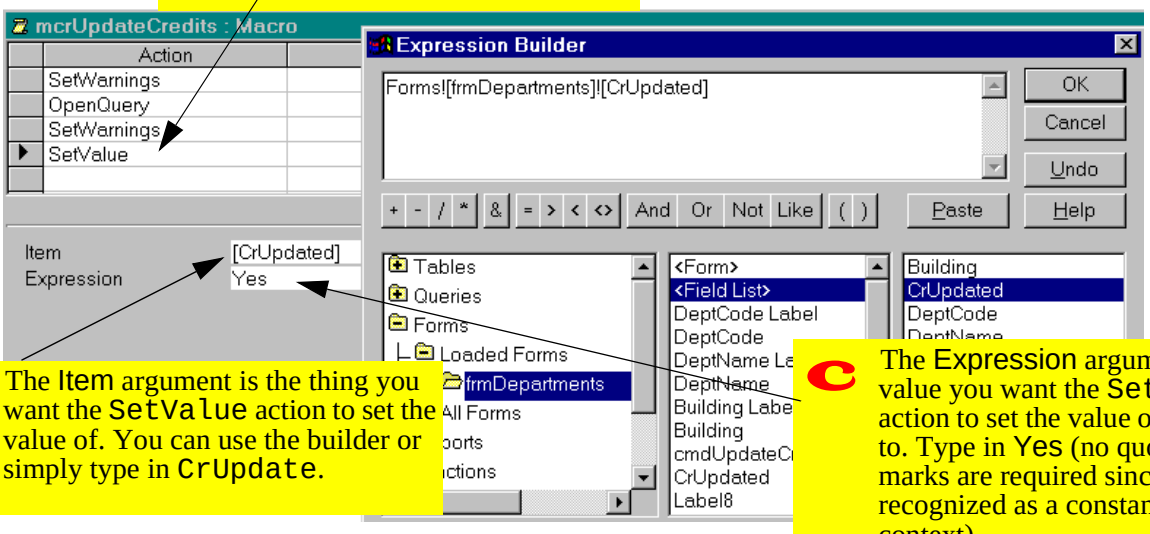
a Select the check box tool from the toolbox.

b Drag the `CrUpdated` field from the field list to the detail section.

? A check box is a control that can be bound to fields of the yes/no data type. When the box is checked, `True` is stored in the table; when the box is unchecked, `False` is stored.

FIGURE 13.8: Add a `SetValue` command to set the value of the update status field when the update is compete.

a Pick the `SetValue` command from the list or simply type it in.

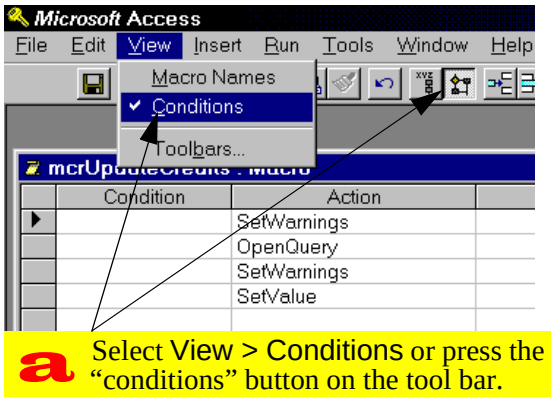


b The Item argument is the thing you want the `SetValue` action to set the value of. You can use the builder or simply type in `CrUpdate`.

c The Expression argument is the value you want the `SetValue` action to set the value of the Item to. Type in `Yes` (no quotation marks are required since `Yes` is recognized as a constant in this context).

- Select *View > Conditions* to display the conditions column in the macro editor as shown in Figure 13.9.

FIGURE 13.9: Display the macro editors condition column



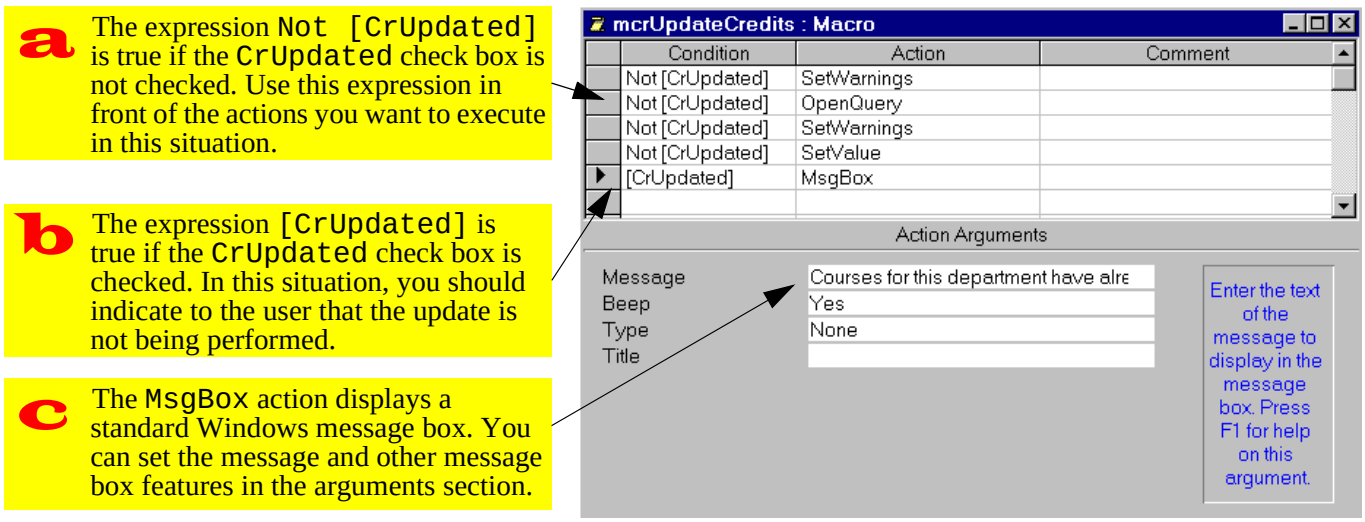
13.3.5.1 The simplest conditional macro

If there is an expression in the condition column of a macro, the action in that row will execute if the condition is true. If the condition is not true, the action will be skipped.

- Fill in the condition column as shown in Figure 13.10. Precede the actions you want to execute if the check box is checked with `[CrUpdated]`. Precede the actions you do not want to execute with `Not [CrUpdated]`.

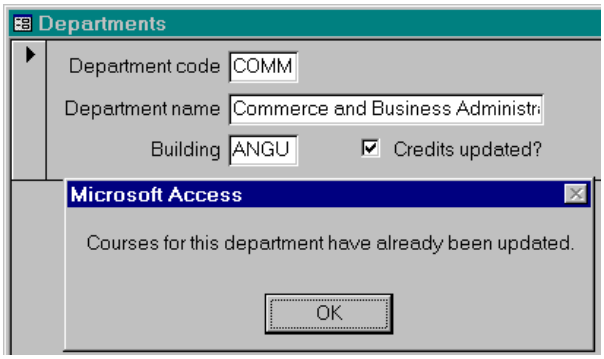
Since `CrUpdated` is a **Boolean** (yes/no) variable, you do not need to write `[CrUpdated] = True` or `[CrUpdated] = False`. The true and false parts are implied. However, if a non-Boolean data type is used in the expression, a comparison operator must be included (e.g., `[DeptCode] = "COMM"`, `[Credits] < 3`, etc.)

FIGURE 13.10: Create a conditional macro to control which actions execute.



- Switch to the form and test the macro by pressing the button. If the `CrUpdated` check box is checked, you should get a message similar to that shown in Figure 13.11.

FIGURE 13.11: The action query is not executed and the message box appears instead.



13.3.5.2 Refining the conditions

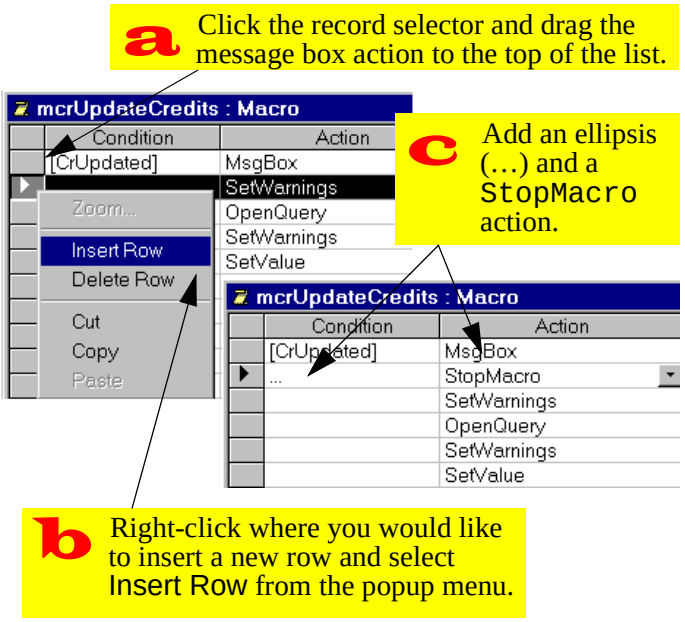
The macro shown in Figure 13.10 can be improved by using an ellipsis (…) instead of repeating the same condition in line after line. In this section, you will simplify your conditional macro slightly.

Move the message box action and condition to the top of the list of actions by dragging its record selector (grey box on the left).

- Insert a new row immediately following the message and add a `StopMacro` action, as shown in Figure 13.12.

The macro in Figure 13.12 executes as follows: If `CrUpdate` is true (i.e., the box is checked), the `MsgBox` action executes. Since the next line has an ellipsis in the condition column, the condition continues to apply. However, that action on the ellipsis line is `StopMacro`, and thus the macro ends without executing the next four lines.

FIGURE 13.12: Rearrange the macro actions and insert a new row.



If the `CrUpdate` box is not checked, the first two lines are ignored (i.e., the lines with the false condition and the ellipsis) and the update proceeds.

13.3.5.3 Creating a group of named macros

It is possible to store a number of related macros together in one macro "module". These **group macros** have two advantages:

1. **Modular macros can be created** — instead of having a large macro with many conditions and branches, you can create a small macro that call other small macros.
2. **Similar macros can be grouped together** — for example, you could keep all you `Departments`-related macros or search-related macros in a macro group.

In this section, we will focus on the first advantage.

- Select *View > Macro Names* to display the macro name column.

- Perform the steps in Figure 13.13 to modularize your macro.
- Change the macro referred to in the *On Click* property of the `cmdUpdateCredits` button from `mcrUpdateCredits` to `mcrUpdateCredits.CheckStatus`.
- Test the operation of the button.

13.3.6 Creating switchboards

One of the simplest (but most useful) triggers is an `OpenForm` command attached to a button on a form consisting exclusively of buttons.

This type of "switchboard" (as shown in Figure 13.14) can provide the user with a means of navigating the application.

- Create an unbound form as shown in Figure 13.15.
- Remove the scroll bars, navigation buttons, and record selectors from the form using the form's property sheet.
- Save the form as `swbMain`.

There are two ways to add button-based triggers to a form:

1. Turn the button wizard off, create the button, and attach an macro containing the appropriate action (or actions).
2. Turn the button wizard on and use the wizard to select from a list of common actions (the wizard writes a VBA procedure for you).

Since the wizard can only attach one action to a button (such as opening a form or running an action query) it is less flexible than a macro. However, once you are more comfortable with VBA, there is nothing to stop you

FIGURE 13.13: Use named macros to modularize the macro.

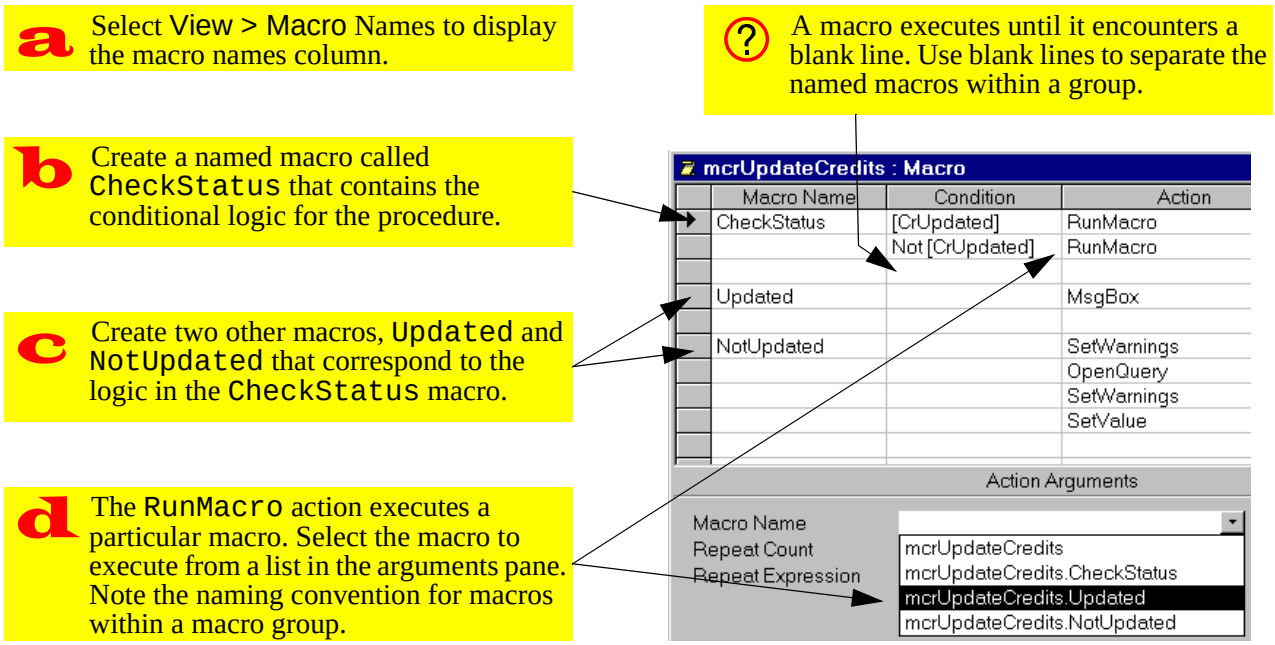
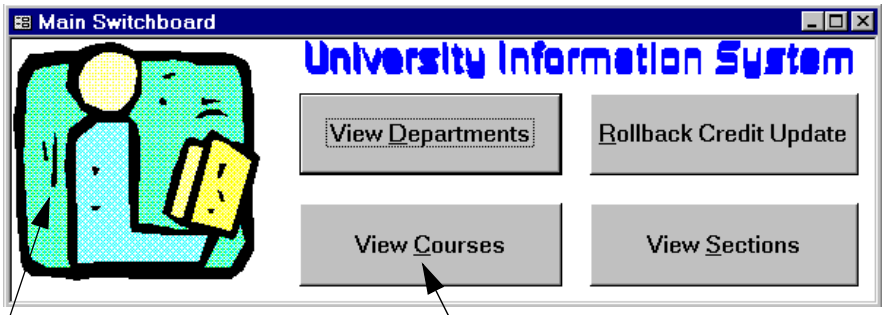


FIGURE 13.14: A switchboard interface to the application.

The command buttons are placed on an unbound form. Note the absence of scroll bars, record selectors, or navigation buttons.

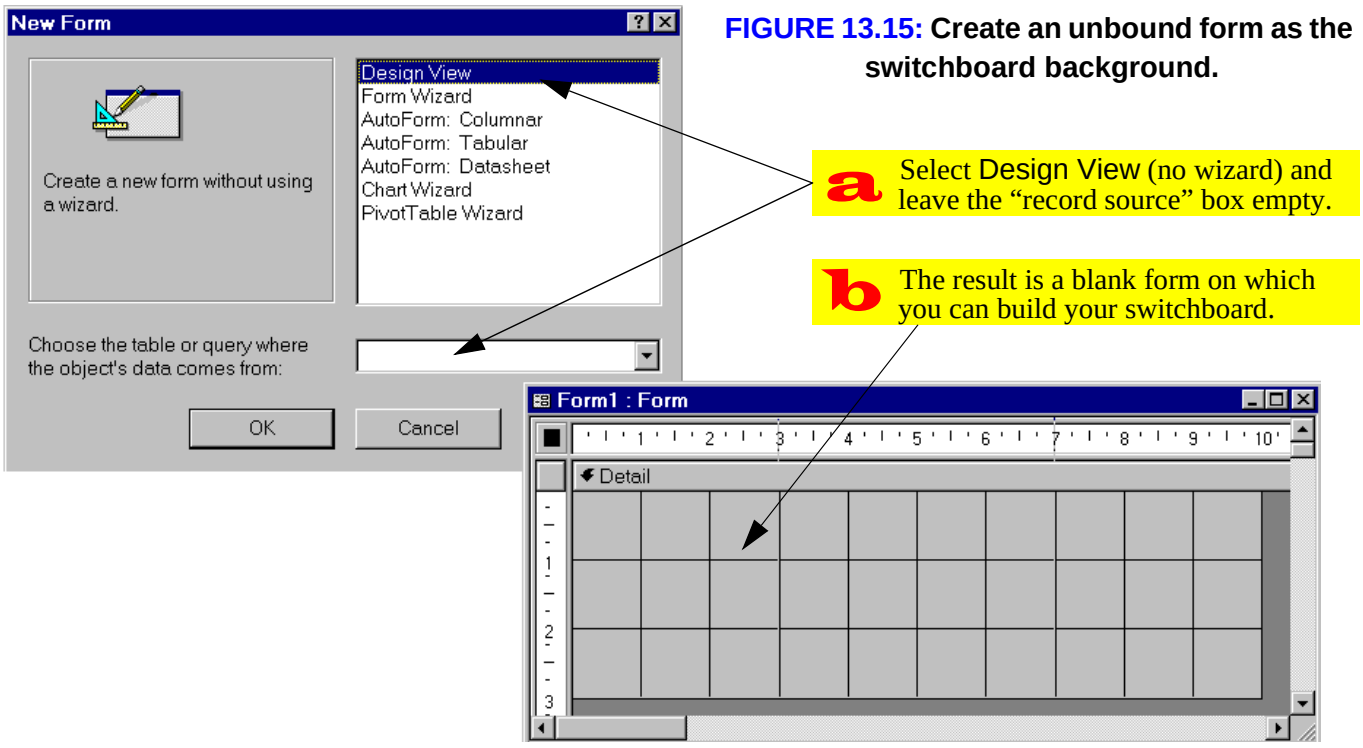
Although it is not shown here, switchboards can call other switchboards, allowing you to add a hierarchical structure to your application.



Gratuitous clip art can be used to clutter your forms and reduce the application's overall performance.

Shortcut keys are include on each button to allow the user to navigate the application with keystrokes.

FIGURE 13.15: Create an unbound form as the switchboard background.



a Select Design View (no wizard) and leave the "record source" box empty.

b The result is a blank form on which you can build your switchboard.

from editing the VBA modules created by the wizard to add additional functionality.

13.3.6.1 Using a macro and manually-created buttons

- Ensure the wizard is turned off and use the button tool to create a button.
- Modify the properties of the button as shown in Figure 13.16.
- Create a macro called `mcrSwitchboard.OpenDept` and use the `OpenForm` command to open the form `frmDepartments`.
- Attach the macro to the *On Click* event of the `cmdDepartments` button.
- Test the button.

13.3.6.2 Using the button wizard

- Turn the button wizard back on and create a new button.
- Follow the directions provided by the wizard to set the action for the button (i.e., open the `frmCourses` form) as shown in Figure 13.17.
- Change the button's font and resize it as required.

You can standardize the size of your form objects by selecting more than one and using *Format > Size > to Tallest* and *to Widest* commands. Similarly, you can select more than one object and use the "multiple selection" property sheet to set the properties all at once.

13.3.7 Using an `autoexec` macro

If you use the name `autoexec` to save a macro (in lieu of the normal `mcr<name>` convention), Access will execute the macro actions when the database is opened. Consequently, auto-execute macros are

FIGURE 13.16: Create a button and modify its appearance.

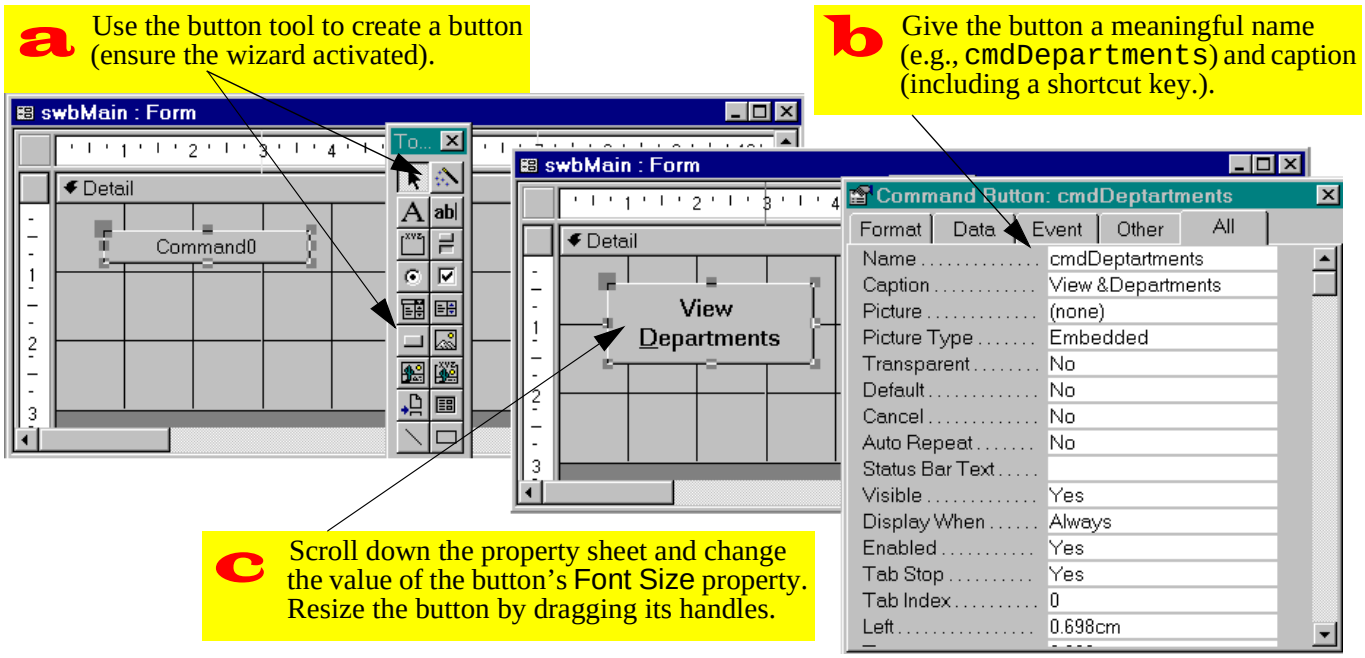
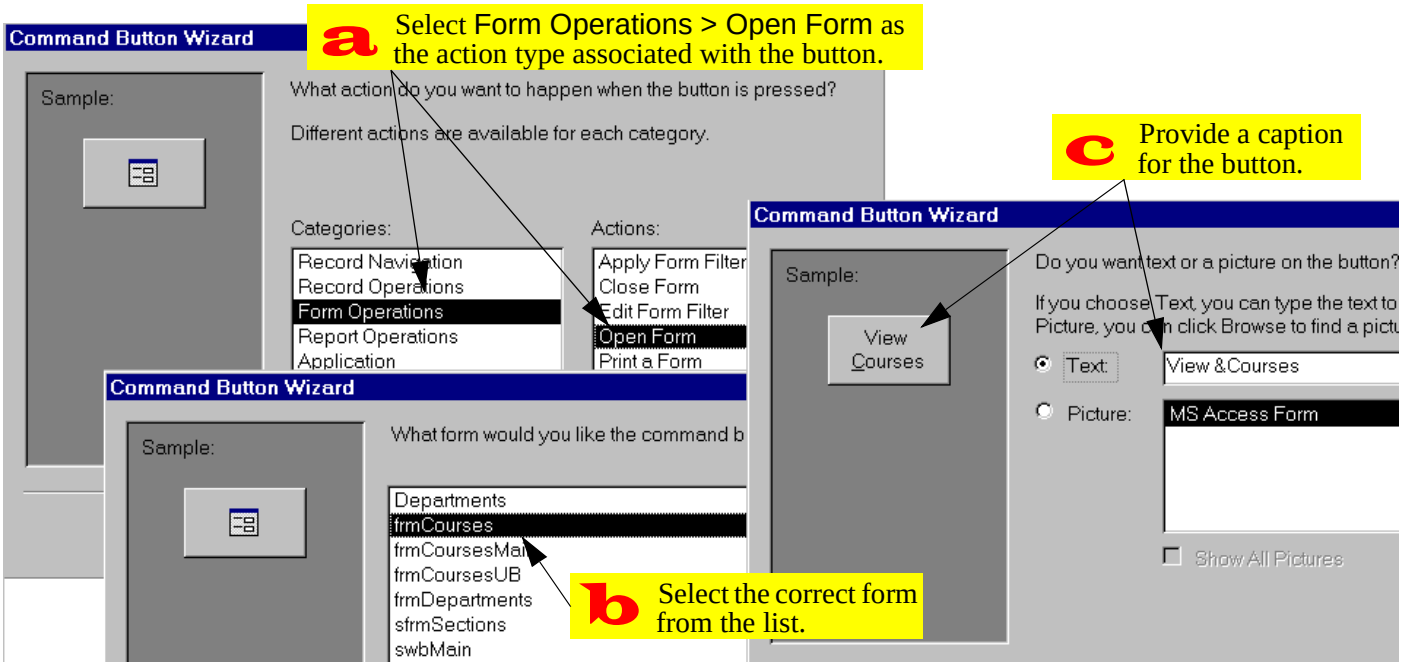


FIGURE 13.17: Use the command button wizard to create a button for the switchboard.



often used to display a switchboard when the user starts the application.

Another typical auto-execute operation is to hide the database window. By doing this, you unclutter the screen and reduce the risk of a user accidentally making a change to the application (by deleting a database object, etc.).

To unhide the database window, select *Window > Unhide* from the main menu or press the database window icon () on the toolbar.

The problem with hiding the database window using a macro is that there is no `HideDatabaseWindow` command in the Access macro language. As such, you have to rely on the rather convoluted `DoMenuItem` action.

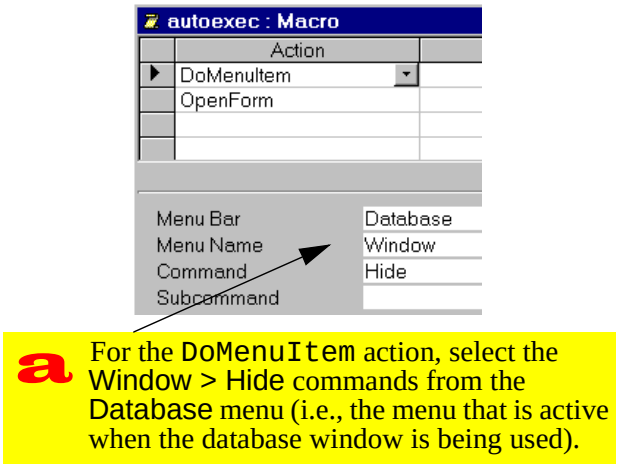
As its name suggests, the `DoMenuItem` action performs an operation just as if it had been selected from the menu system. Consequently, you need to know something about the menu structure of Access before you create your macro.

In version 8.0, the `DoMenuItem` action has been replaced by the slightly more intuitive `RunCommand` action. See on-line help for more information on `RunCommand`.

- Create an auto-execute macro
- Add the `DoMenuItem` and `OpenForm` actions to hide the database window and open the main switchboard, as shown in Figure 13.18.
- Close the database and reopen it after a short delay to test the macro.

In version 7.0 and above, you do not need to use an autoexec macro to hide the database window and open a form. Instead, you can right-click on the database window, select

FIGURE 13.18: Create an auto-execute macro.



a For the `DoMenuItem` action, select the Window > Hide commands from the Database menu (i.e., the menu that is active when the database window is being used).

Startup, and fill in the properties for the application.

13.4 Discussion

13.4.1 Event-driven programming versus conventional programming

The primary advantages of event-driven programming are the following:

1. **Flexibility** — since the flow of the application is controlled by events rather than a sequential program, the user does not have to conform to the programmer's understanding of how tasks should be executed.
2. **Robustness** — Event-driven applications tend to be more robust since they are less sensitive to the order in which users perform activities. In conventional programming, the programmer has to anticipate virtually every sequence of activities the user might perform and define responses to these sequences.

The primary disadvantage of event-driven programs is that it is often difficult to find the source of errors when they do occur. This problem arises from the object-oriented nature of event-driven applications—since events are associated with a particular object you may have to examine a large number of objects before you discover the misbehaving procedure. This is especially true when events cascade (i.e., an event for one object triggers an event for a different object, and so on).

13.5 Application to the assignment

- Add "update status" check boxes to you transaction processing forms (i.e., `Orders` and `Shipments`)
- Create a conditional macro for your `Shipments` form to prevent a particular shipment from being added to inventory more than once.
- Create a main switchboard for you application. It should provide links to all the database objects your user is expected to have access to (i.e., your forms).

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.

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.

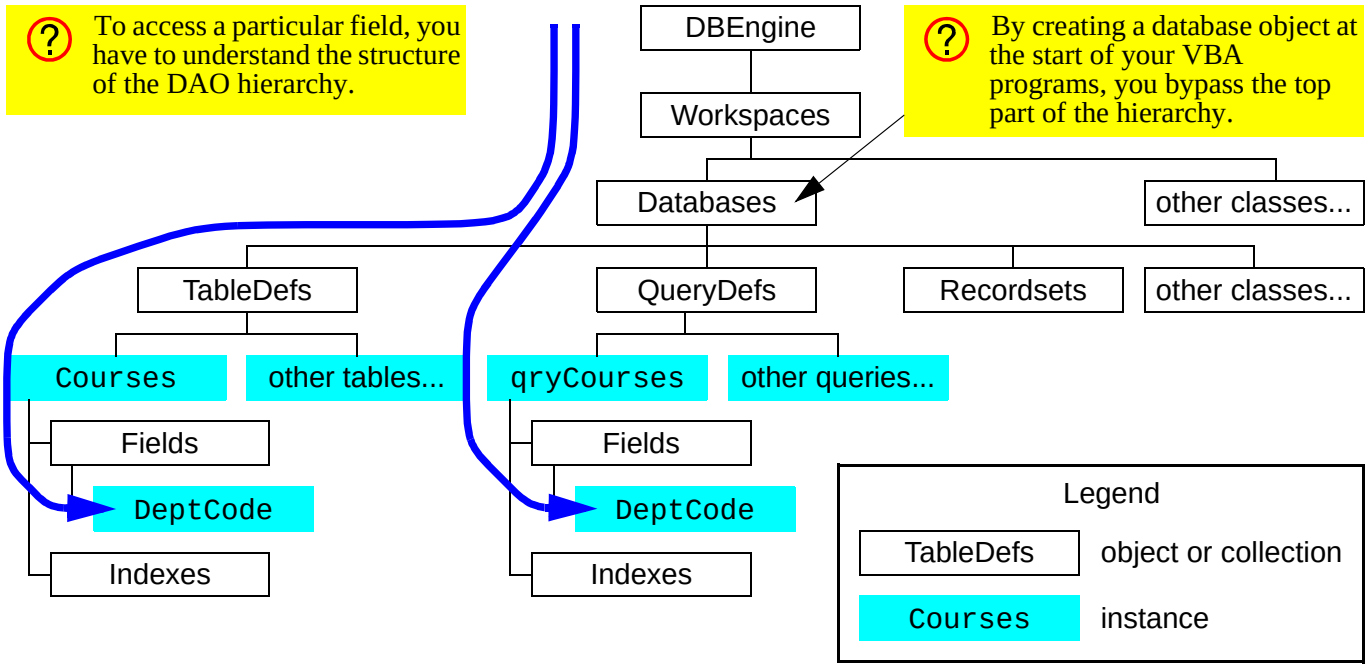
A reasonable question at this point might be: Isn't FieldSize a property of a field, not a method? The answer to this is that the implementation of DAO is somewhat inconsistent in this respect. The best policy is to look at the object summaries in the on-line help if you are unsure.

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:

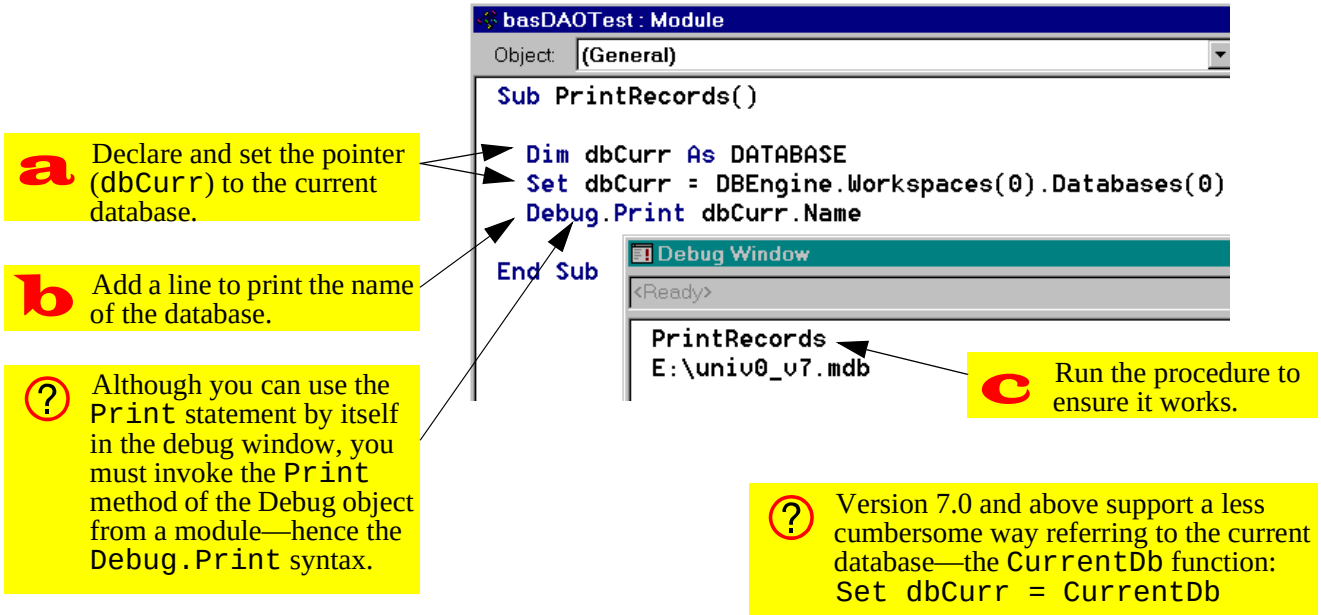
```
Dim dbCurr As DATABASE
Set dbCurr =
  DBEngine.Workspaces(0).Databases(0)
Debug.Print dbCurr.Name
```

- 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.



(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

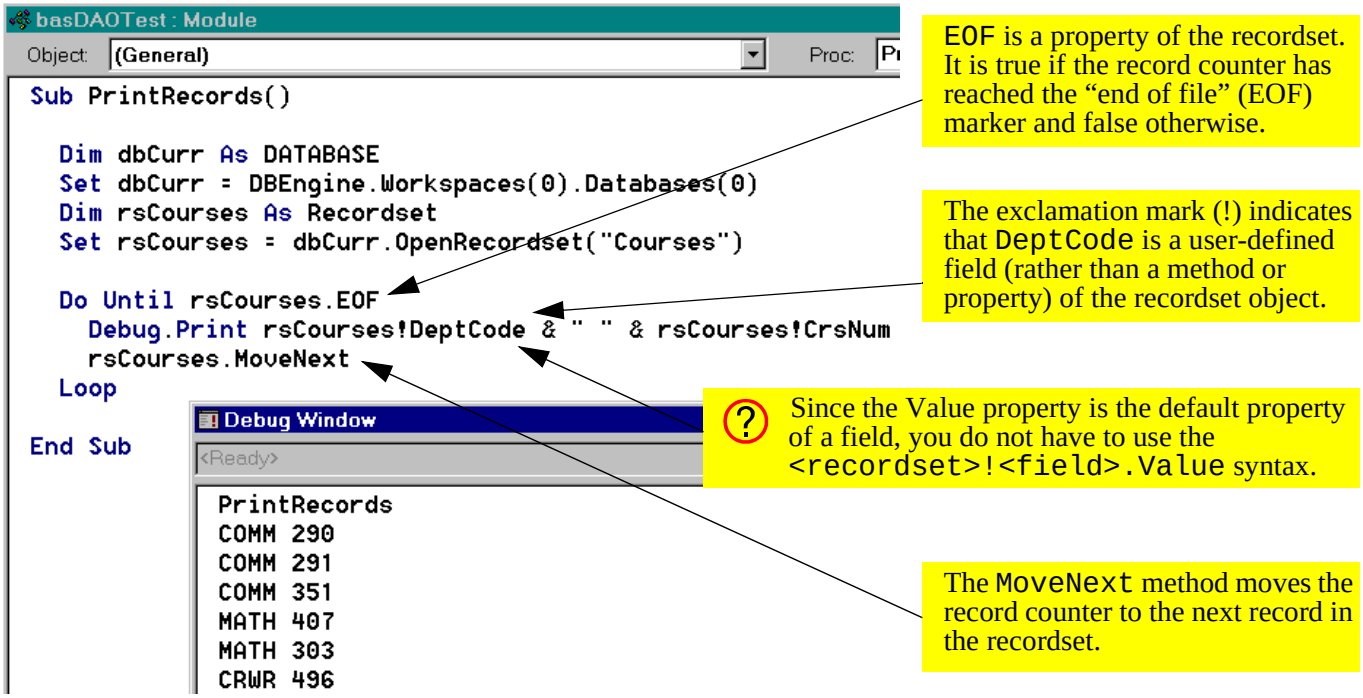
In this section, you will use some of the properties and methods of a Recordset object to print its contents.

- 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.



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()`.

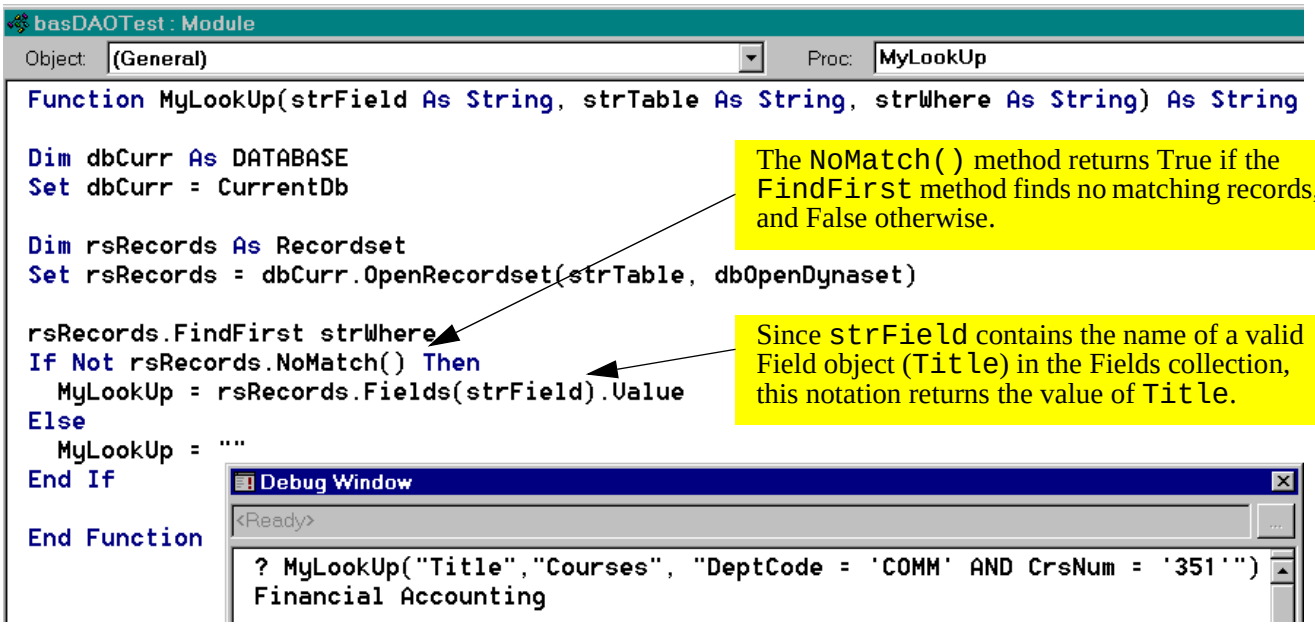
14.3.5 The `DLookUp()` function

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] & "'")
```

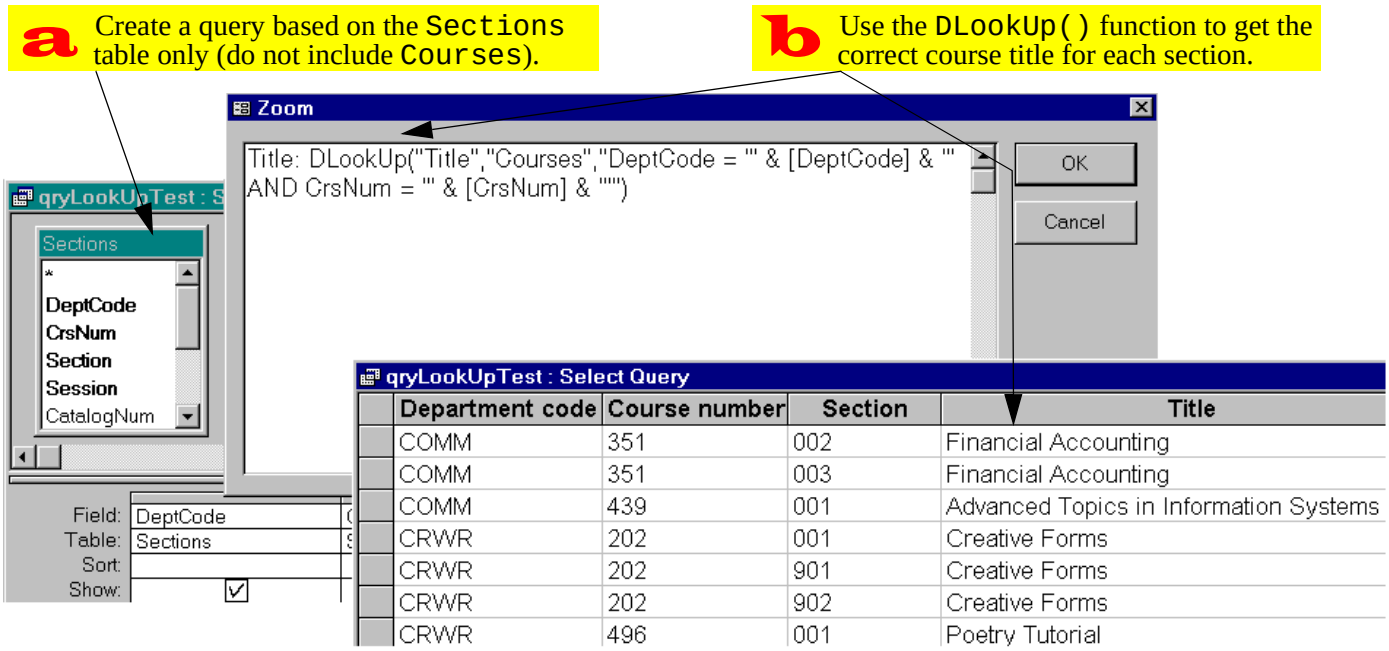
14.3.5.2 Understanding the WHERE clause

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

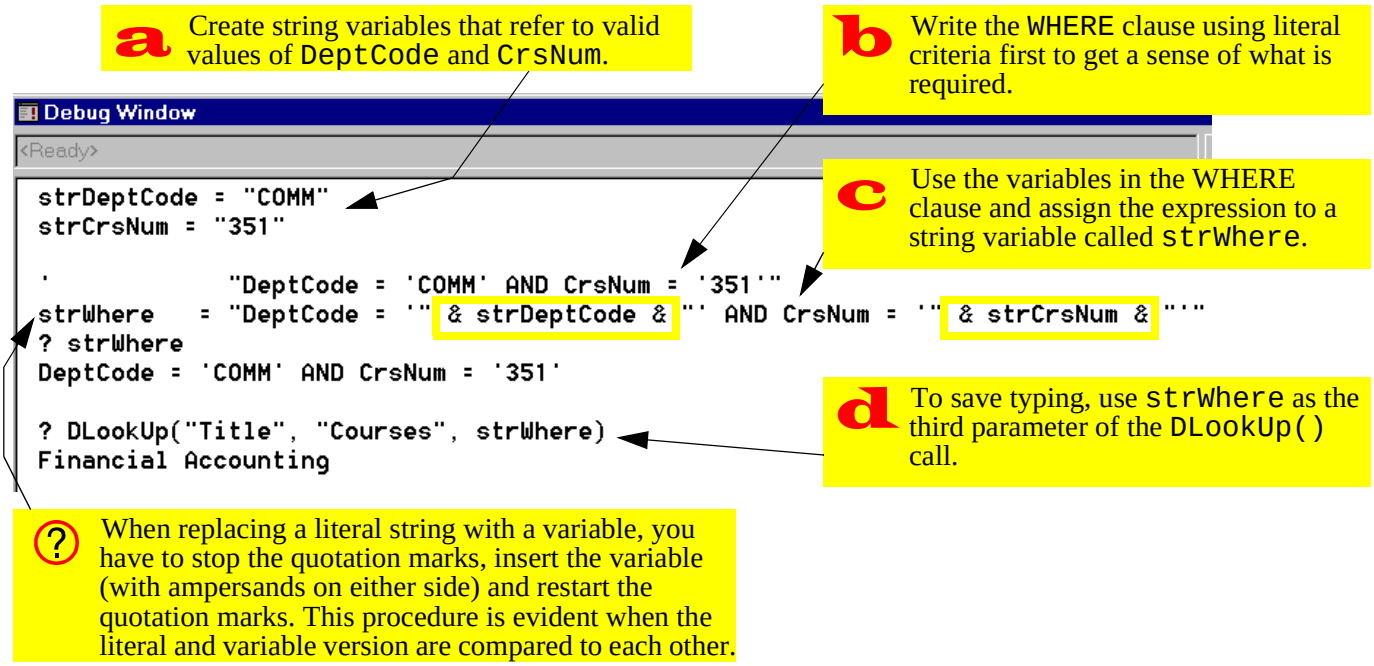
14.4.1 VBA versus SQL

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 `Nulls` 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.

Access Tutorial 15: Advanced Triggers

15.1 Introduction: Pulling it all together

In this tutorial, you will bring together several of the skills you have learned in previous tutorials to implement some sophisticated triggers.

15.2 Learning objectives

- How do I run VBA code using a macro?
- How do I use the value in one field to automatically suggest a value for a different field?
- How do I change the table or query a form is bound to once the form is already created?
- What is the *After Update* event? How is it used?
- How do I provide a search capability for my forms?
- How do I create an unbound combo box?
- Can I implement the search capability using Visual Basic?

15.3 Tutorial exercises

15.3.1 Using a macro to run VBA code

There a some things that cannot be done using the Access macro language. If the feature you wish to implement is critical to your application, then you must implement it using VBA. However, since it is possible to call a VBA function from within a macro, you do not have to abandon the macro language completely.

In this section, you are going to execute the `ParameterTest` subroutine you created in Section 12.3.6 from within a macro. Since the `RunCode` action of the Access macro language can only be used to execute functions (not subroutines) you must do one of two things before you create the macro:

1. Convert `ParameterTest` to a function — you do this simply by changing the Sub at the start of the procedure to `Function`.
2. Create a new function that executes `ParameterTest` and call the function from the macro.

15.3.1.1 Creating a wrapper

Since the second alternative is slightly more interesting, it is the one we will use.

- Open your `basTesting` module from Tutorial 12.
- Create a new function called `ParameterTestWrapper` defined as follows:

```
Function
  ParameterTestWrapper(intStart As
  Integer, intStop As Integer) As
  Integer
  'this function calls the
   ParameterTest subroutine
  ParameterTest intStart, intStop
  ParameterTestWrapper = True
   'return a value
  End Function
```

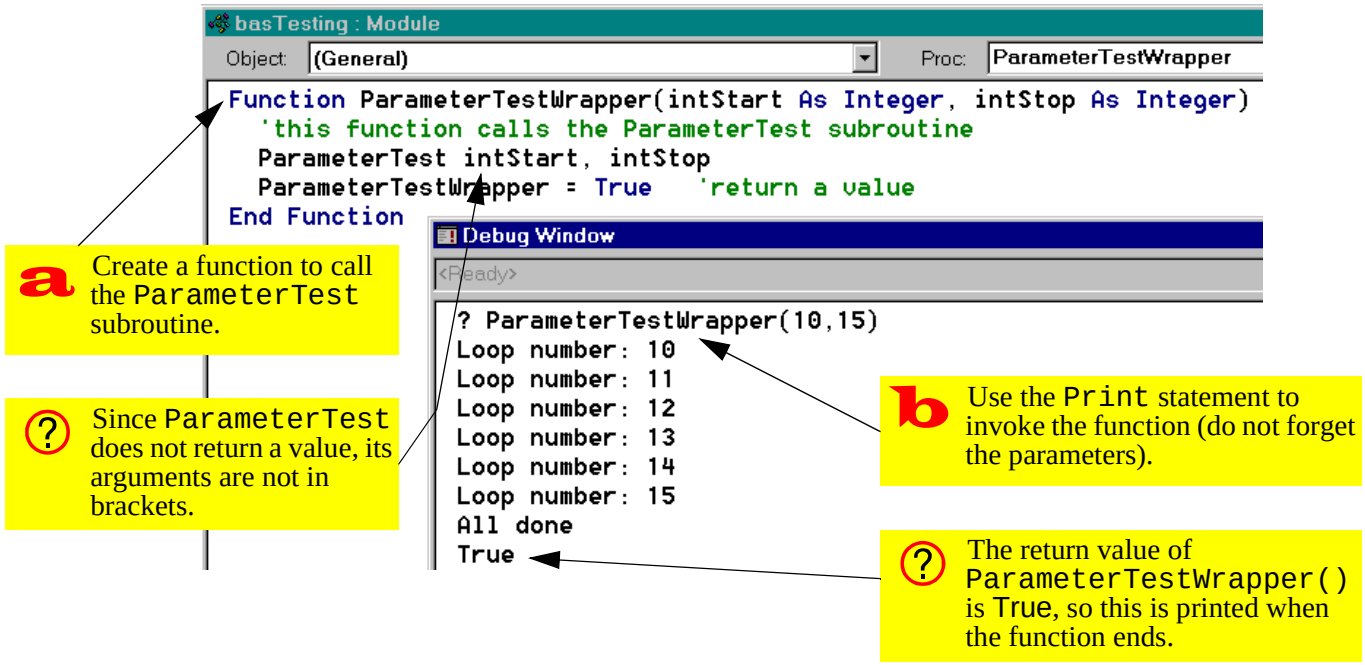
- Call the function, as shown in Figure 15.1.

Note that the return value of the function is declared as an integer, but the actual assignment statement is `ParameterTestWrapper = True`. This is because in Access/VBA, the constants `True` and `False` are defined as integers (-1 and 0 respectively).

15.3.1.2 Using the `RunCode` action

- Leave the module open (you may have to resize and/or move the debug window) and create a new macro called `mcrRunCodeTest`.

FIGURE 15.1: Create a function that calls the `ParameterTest` subroutine.



- Add the `RunCode` action and use the expression builder to select the correct function to execute, as shown in Figure 15.2.

The expression builder includes two parameter place holders (`<<intStart>>` and `<<intStop>>`) in the function name. These are to remind you that you must pass two parameters to the `ParameterTestWrapper()` function. If you leave the place holders where they are, the macro will fail because Access has not idea what `<<intStart>>` and `<<intStop>>` refer to.

- Replace the parameter place holders with two numeric parameters (e.g. 3 and 6). Note that in general, the parameters could be field names or any other references to Access objects containing (in this case) integers.
- Select *Run > Start* to execute the macro as shown in Figure 15.3.

15.3.2 Using activity information to determine the number of credits

In this section, you will create triggers attached to the *After Update* event of bound controls.

15.3.2.1 Scenario

Assume that each type of course activity is generally associated with a specific number of credits, as shown below:

Activity	Credits
lecture	3.0
lab	3.0
tutorial	1.0
seminar	6.0

FIGURE 15.2: Use the expression builder to select the function to execute.

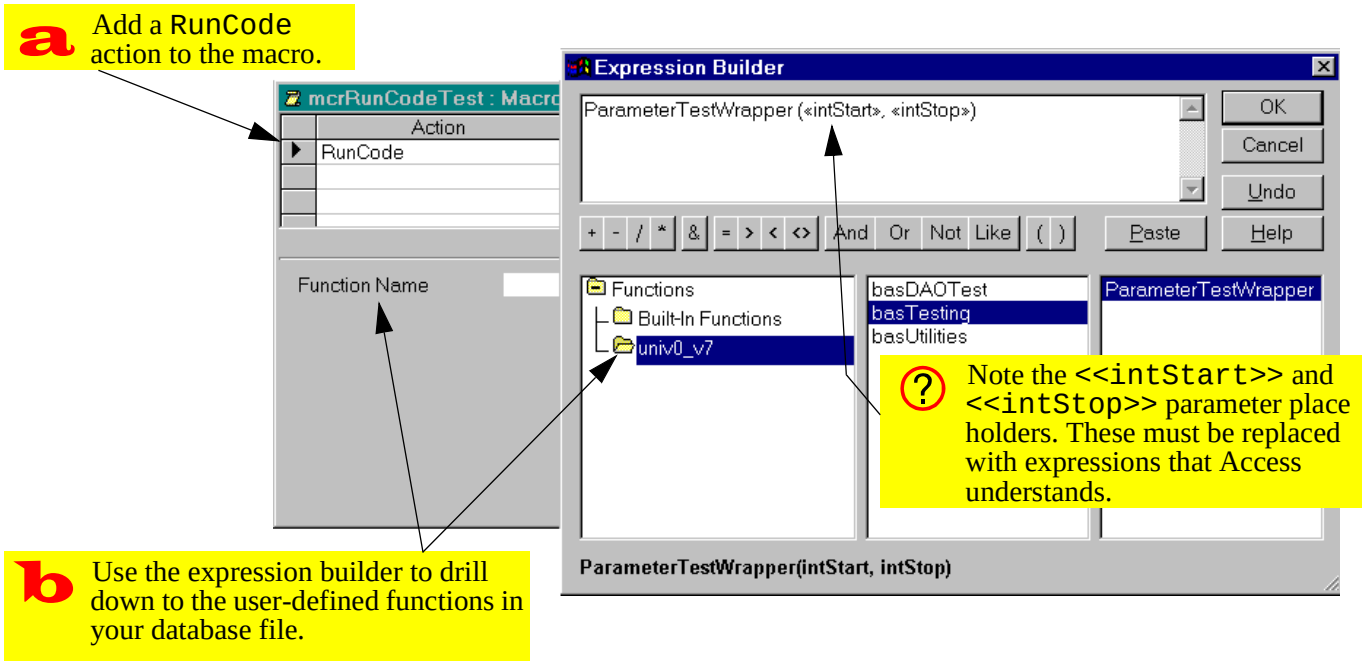
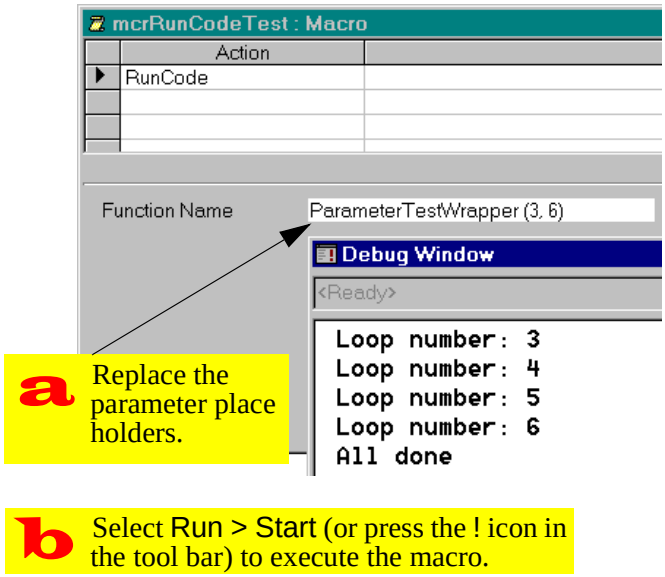


FIGURE 15.3: Execute the `RunCode` macro.



Assume as well that the number of credits for a particular type of course is not cast in stone. As such, the numbers given above are merely "default" values.

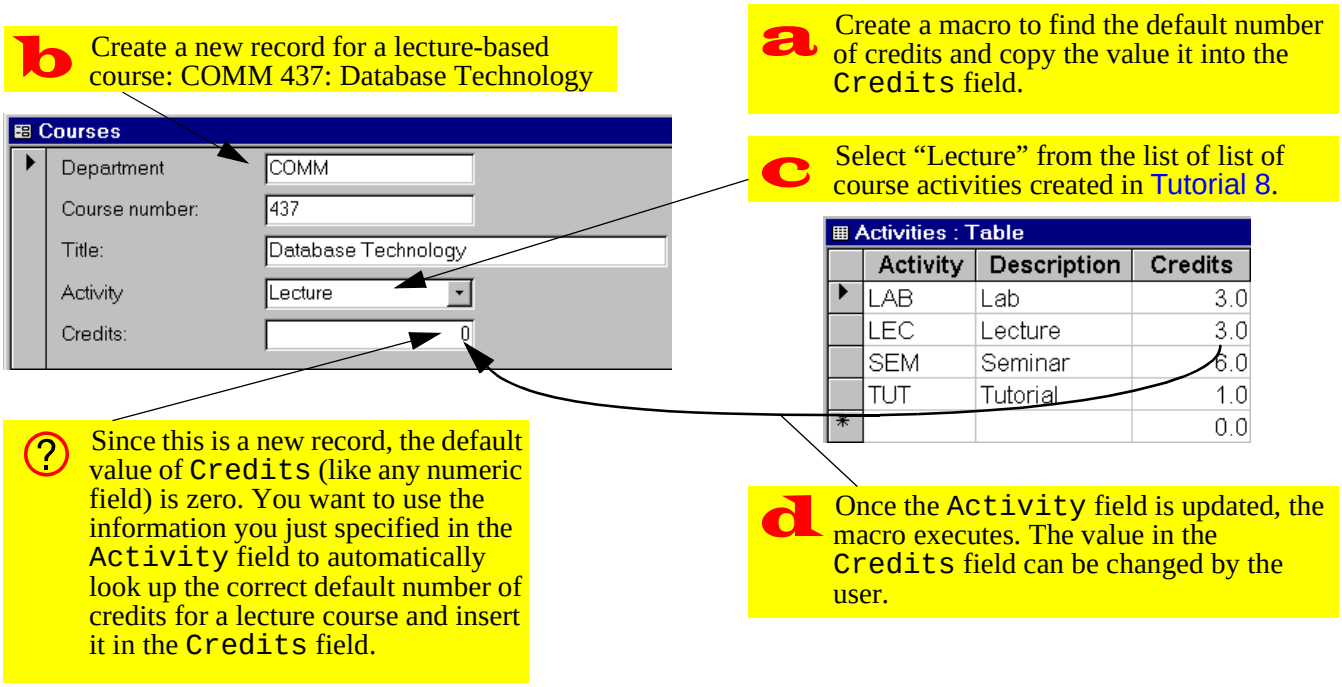
You want to use the default credit values when you create a new course or modify an existing course. However, the user may override this default if necessary for a particular course. The basic requirement is illustrated in Figure 15.4.

15.3.2.2 Designing the trigger

Based on the foregoing, the answer to the "what" question is the following:

1. Look up the default number of credits associated with the course activity showing in the form's `Activity` field.
2. Copy this number into the `Courses.Credits` field.

FIGURE 15.4: Inserting a default value into a new record.



There are several possible answers to the "when" question (although some are better than others). For example:

1. When the user enters the `Credits` field (the *On Enter* event for `Credits`) — The problem with this choice is that the user could modify the course's activity without moving the focus to the `Activity` field. In such a case, the trigger would not execute.
2. When the user changes the `Activity` field (the *After Update* event for `Activity`) — This choice guarantees that whenever the value of `Activity` is changed, the default value will be copied into the `Credits` field. As such, it is a better choice.

15.3.2.3 Preliminary activities

- Modify the `Activities` table to include a single-precision numeric field called `Credits`. Add the values shown in the table in Section 15.3.2.1.
- Ensure that you have a courses form (e.g., `frm-Courses`) and that the form has a combo box for the `Activity` field. You may wish to order the fields such that `Activity` precedes `Credits` in the tab order (as shown in Figure 15.4).

If your move fields around, remember to adjust the tab order accordingly (recall Section 8.3.4).

15.3.2.4 Looking up the default value

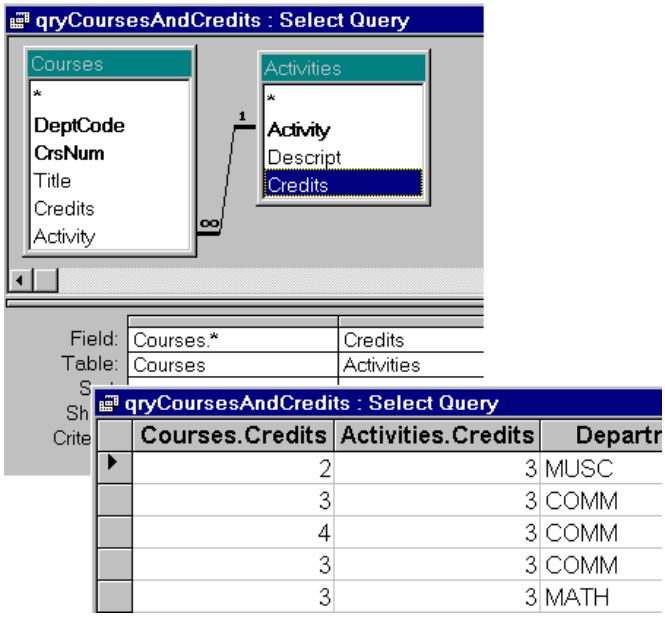
As you discovered in Section 14.3.5, Access has a `DLookUp()` function that allows you to go to the `Activities` table and find the value of `Credits` for a particular value of `Activity`. A different approach is to join the `Activities` table with the `Courses` table in a query so that the default value of credits is always available in the form. This is the approach we will use here.

- Ensure you have a relationship (in the main relationship window) between `Courses.Activity` and `Activities.Activity`.
- Create a new query called `qryCoursesAndCredits` based on the `Courses` and `Activities` tables (see Figure 15.5).

Notice that you have two credits fields: `Courses.Credits` (the actual number of credits for the course) and `Activities.Credits` (the "default" or "suggested" number of credits based on the value of `Activity`). Access uses the `<table name>.<field name>` notation whenever a query contains more than one field with the same name.

Since you already have forms based on the `Courses` table that expect a field called `Credits` (rather than one called `Courses.Credits`), it is a good idea to rename the `Activities.Credits` field in the query. You do this by creating a calculated field.

FIGURE 15.5: Use a join to make the default value available.



- Rename `Activities.Credits` to `DefaultCredits` as shown in Figure 15.6. Note that this eliminates the need for the `<table name>.<field name>` notation.

15.3.2.5 Changing the *Record Source* of the form

Rather than create a new form based on the `qryCoursesAndCredits` query, you can modify the *Record Source* property of the existing `frmCourses` form so it is bound to the query rather than the `Courses` table.

- Bring up the property sheet for the `frmCourses` form and change the *Record Source* property to `qryCoursesAndCredits` as shown in Figure 15.7.

FIGURE 15.6: Rename one of the `Credits` fields.

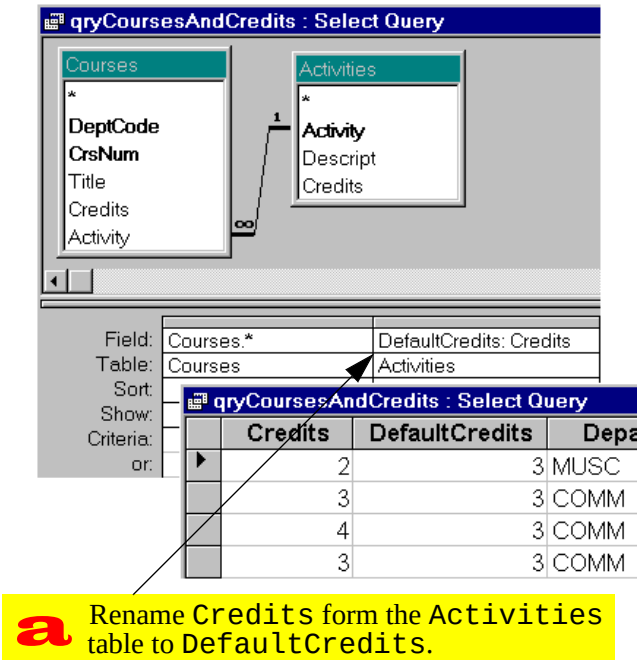
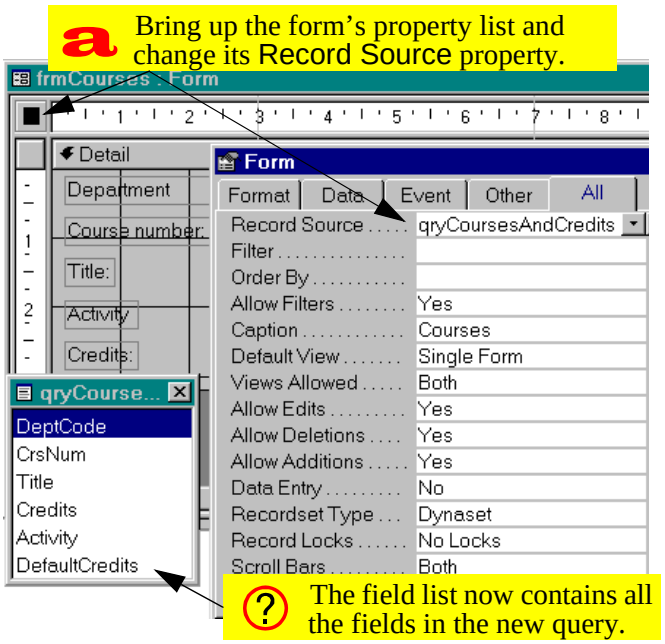


FIGURE 15.7: Change the *Record Source* property of an existing form.



The advantage of using a join query in this manner is that `DefaultCredits` is now available for use within the form and within any macros or VBA modules that run when the form is open.

15.3.2.6 Creating the `SetValue` macro

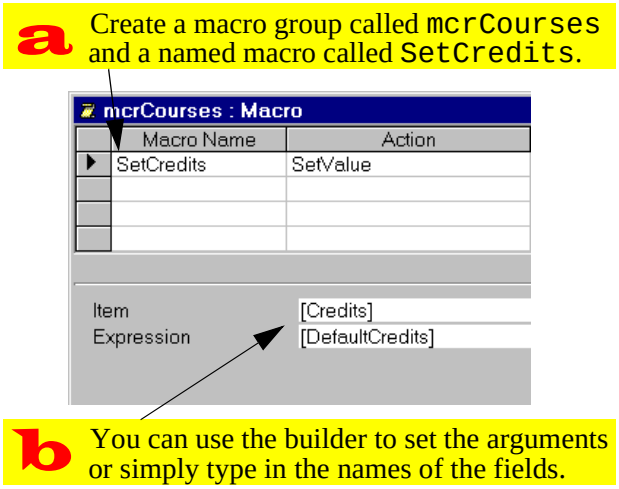
The `SetValue` macro you require here is extremely simple once you have `DefaultCredits` available within the scope of the form.

- Create the `mcrCourses.SetCredits` macro as shown in Figure 15.8.

15.3.2.7 Attaching a procedure to the *After Update* event

The *On Click* event of a button is fairly simple to understand: the event occurs when the button is clicked. The events associated with non-button objects operate in exactly the same way. For example, the *After Update* event for controls (text box, combo box, check box, etc.) occurs when the value

FIGURE 15.8: Create the `SetValue` macro.



of the control is changed by the user. As a result, the *After Update* event is often used to trigger data verification procedures and "auto-fill" procedures like the one you are creating here.

- Attach the `mcrCourses.SetCredits` macro to the *After Update* event of the `Activity` field.
- Verify that the trigger works properly.

15.3.3 Use an unbound combo box to automate search

As mentioned in Tutorial 8, a combo box has no intrinsic search capability. However, the idea of scanning a short list of key values, selecting a value, and having all the information associated with that record pop on to the screen is so basic that in Access version 7.0 and above, this capability is included in the combo box wizard. In this tutorial, we will look at a couple of different means of creating a combo boxes for search from scratch.

15.3.3.1 Manual search in Access

To see how Access searches for records, do the following:

- Open your `frmDepartments` form.

- Move to the field on which you want to search (e.g., `DeptCode`);
- Select *Edit > Find* (or press *Control-F*);
- Fill out the search dialog box as shown in Figure 15.9.

In the dialog box, you specify what to search for (usually a key value) and specify how Access should conduct its search. When you press *Find First*, Access finds the first record that matches your search value and makes it the current record (note that if you are searching on a key field, the *first* matching record is also the *only* matching record).

15.3.3.2 Preliminaries

To make this more interesting, assume that the `frmDepartments` form is for viewing editing existing departmental information (rather than adding new departments). To enforce this limitation, do the following:

- Set the form's *Allow Additions* property to `No`.
- Set the *Enabled* property of `DeptCode` to `No` (the user should never be able to change the key values of existing records).

15.3.3.3 Creating the unbound combo box

The key thing to remember about the combo box used to specify the search criterion is that it has nothing to do with the other fields or the underlying table. As such, it should be unbound.

- Create an unbound combo box in the form header, as shown in Figure 15.10.
- Change the *Name* property of the combo box to `cboDeptCode`.
- The resulting combo box should resemble that shown in Figure 15.11.

When you create an unbound combo box, Access gives it a default name (e.g., `Combo5`). You should do is change this to something more descriptive (e.g., `cboDept-`

FIGURE 15.9: Search for a record using the "find" dialog box.

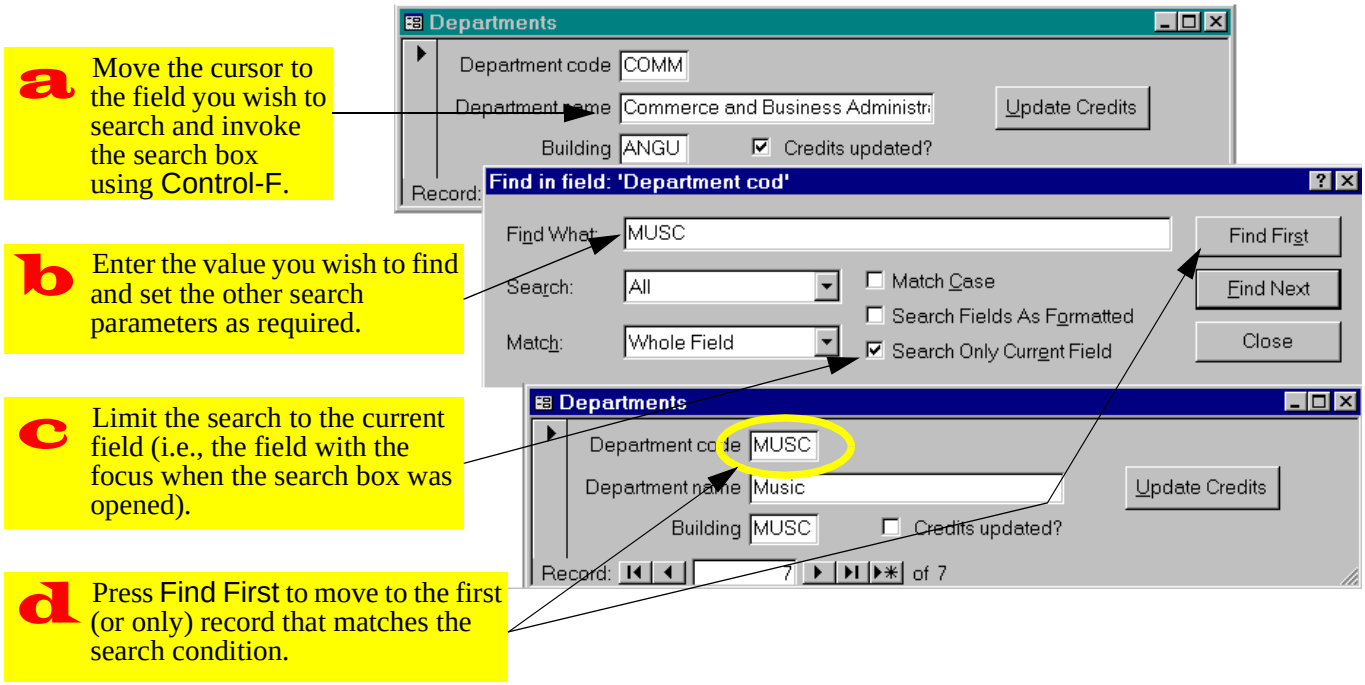


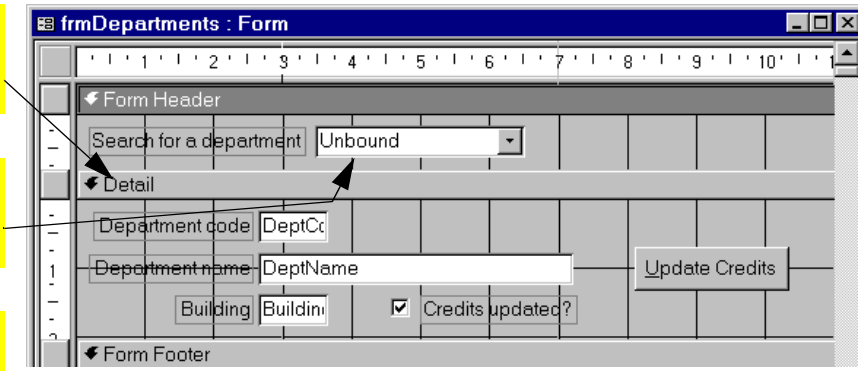
FIGURE 15.10: Create an unbound combo box.

a Drag the separator for the detail down to make room in the form header

b Create an unbound combo box by selecting the combo box tool and clicking in the header area.

c Use the wizard in the usual way to get a list of valid `DeptCode` values and descriptions. The bound column for the combo box should be `DeptCode`.

d Since the combo box is unbound, its value has to be stored for later use rather than stored in a field.



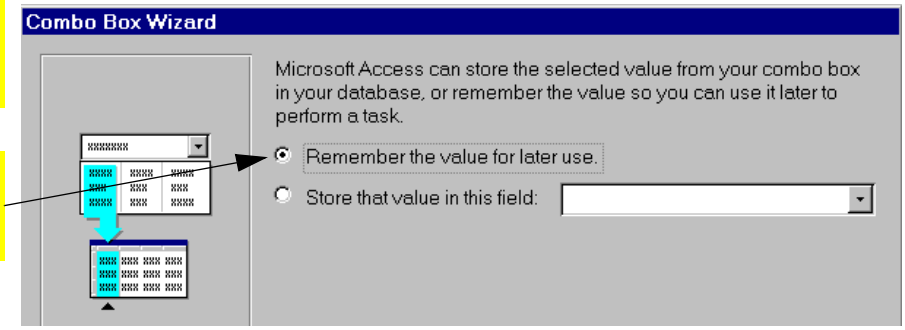
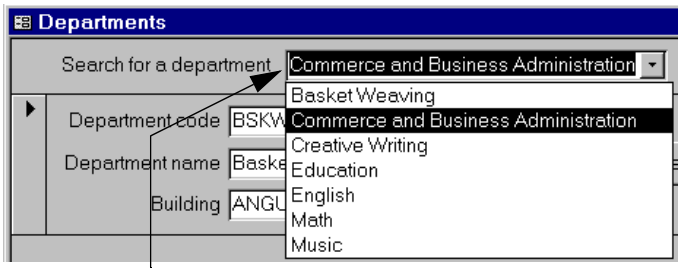


FIGURE 15.11: An unbound combo box.



? Although the `DeptCode` column has been hidden, it is the "bound" column. As a result, the value of the combo box as it appears here is "COMM", not "Commerce and ..."

`Code`). The advantage of the prefix `cbo` is that it allows you to differentiate between the bound field `DeptCode` and the unbound combo box.

15.3.3.4 Automating the search procedure using a macro

When we implement search functionality with a combo box, only two things are different from the manual search in Figure 15.9:

1. the search dialog box does not show up, and
2. the user selects the search value from the combo box rather than typing it in.

The basic sequence of actions, however, remains the same. As a result, the answer to the "what" question is the following:

1. Move the cursor to the `DeptCode` field (this allows the "Search Only Current Field" option to be used, thereby drastically cutting the search time).
2. Invoke the search feature using the current value of `cboDeptCode` as the search value.

3. Move the cursor back to `cboDeptCode` or some other field.

The only problem with this procedure is that the `DeptCode` text box is disabled. As a result, you must include an extra step at the beginning of the macro to set its *Enabled* property to `Yes` and another at the end of the macro to return it to its original state.

- Create a new macro called `mcrSearch.Find-Department`.
- Use the `SetValue` action to set the `Dept-Code.Enabled` property to `Yes`. This can be done using the expression builder, as shown in Figure 15.12.
- Use the `GotoControl` action to move the cursor to the `DeptCode` text box. Note that this action will fail if the destination control is disabled.
- Use the `FindRecord` action to implement the search as shown in Figure 15.13.

FIGURE 15.13: Fill in the arguments for the `FindRecord` action.

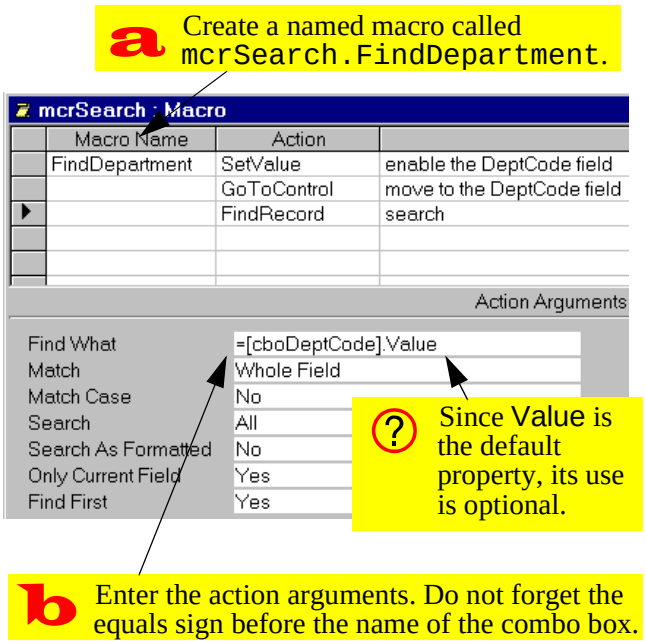
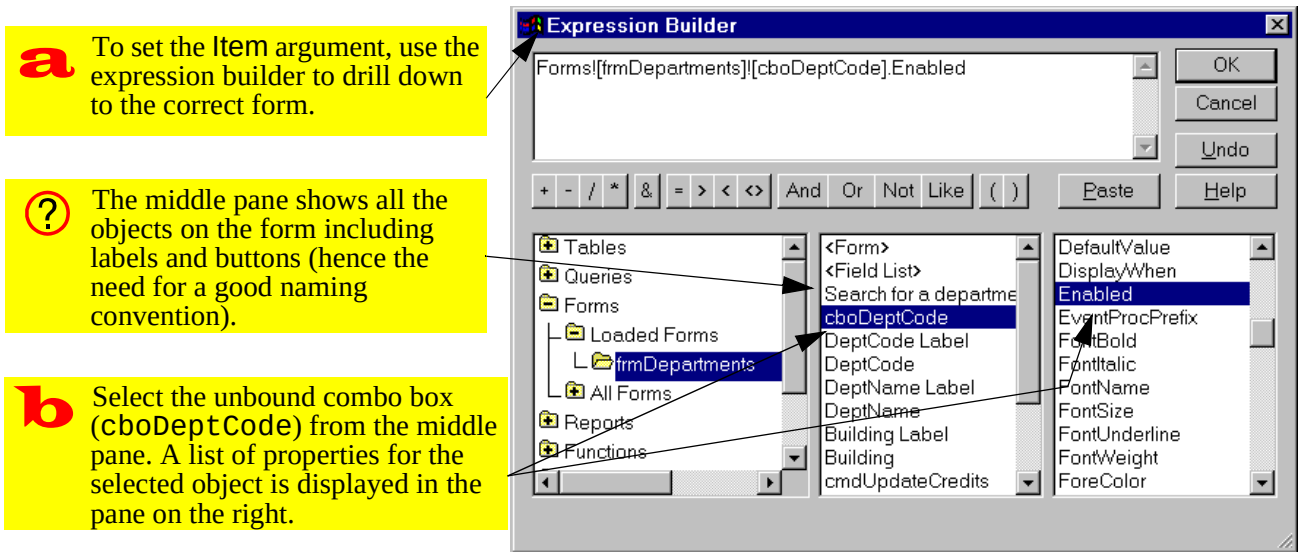


FIGURE 15.12: Use the builder to specify the name of the property to set.



Access interprets any text in the *Find What* argument as a literal string (i.e., quotation marks would not be required to find `COMM`). To use an expression (including the contents of a control) in the *Find What* argument, you must precede it with an equals sign (e.g., `=[cboDeptCode]`.

- You cannot disable a control if it has the focus. Therefore, include another `GotoControl` action to move the cursor to `cboDeptCode` before setting `DeptCode.Enabled = No`.
- Attach the macro `mcrSearch.FindDepartment` to the *After Update* event of the `cboDeptCode` combo box.
- Test the search feature.

15.3.4 Using Visual Basic code instead of a macro

Instead of attaching a macro to the *After Update* event, you can attach a VBA procedure. The VBA procedure is much shorter than its macro counterpart:

1. a copy (clone) of the recordset underlying the form is created,
2. the `FindFirst` method of this recordset is used to find the record of interest.
3. the “bookmark” property of the clone is used to move to the corresponding bookmark for the form.

To create a VBA search procedure, do the following:

- Change the *After Update* event of `cboDeptCode` to “Event Procedure”.
- Press the builder (…) to create a VBA subroutine.

- Enter the two lines of code below, as shown in Figure 15.14.

```
Me.RecordsetClone.FindFirst
  “DeptCode = ‘” & cboDeptCode & “'”
Me.Bookmark =
  Me.RecordsetClone.Bookmark
```

This program consists of a number of interesting elements:

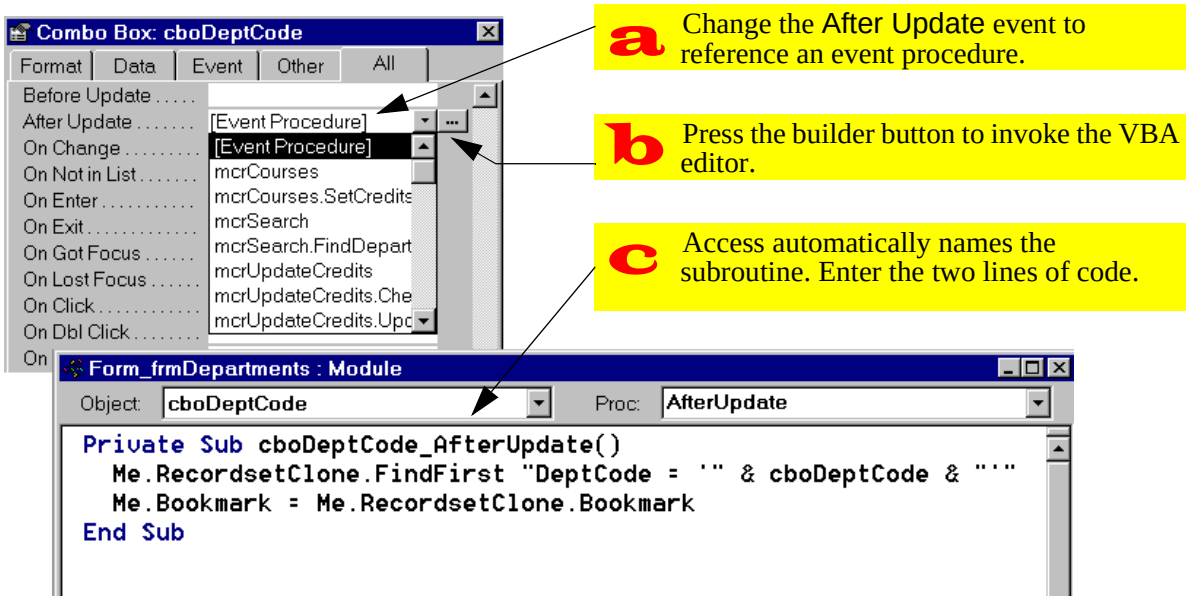
- The property `Me` refers to the current form. You can use the form's actual name, but `Me` is much faster to type.
- A form’s `RecordsetClone` property provides a means of referencing a copy of the form's underlying recordset.
- The `FindFirst` method is straightforward. It acts, in this case, on the clone.
- Every recordset has a bookmark property that uniquely identifies each record. A bookmark is like a “record number”, except that it is stored as a non-human-readable data type and therefore is not of much use unless it is used in the manner shown here. Setting the *Bookmark* property of a record makes the record with that bookmark the current record. In the example above, the bookmark of the records underlying the form is set to equal the bookmark of the clone. Since the clone had its bookmark set by the search procedure, this is equivalent to searching the recordset underlying the form.

15.4 Application to the assignment

15.4.1 Triggers to help the user

- Create a trigger on your order form that sets the actual selling price of a product to its default price. This allows the user to accept the default price or enter a new price for that particular transaction (e.g., the item could be damaged). You will

FIGURE 15.14: Implement the search feature using a short VBA procedure.



have to think carefully about which event to attach this macro to.

- Create a trigger on your order form that calculates a suggested quantity to ship and copies this value into the quantity to ship field. The suggested value must take into account the amount ordered by the customer, any outstanding backorders for that item by that customer, and the current quantity on hand (you cannot ship what you do not have). The user should be able to override this suggested value. (Hint: use the `MinValue()` function you created in Section 12.5.)
- Provide you customer and products forms with search capability.

15.4.2 Updating the `BackOrders` table

Once a sales order is entered into the order form, it is a simple matter to calculate the amount of each product that should be backordered (you did this in Section 10.4). The problem is updating the `BackOrders` table itself because two different situations have to be considered:

1. **A record for the particular customer-product combination exists in the `BackOrders` table** -- If a backorder record exists for a particular customer and a particular product, the quantity field of the record can be added-to or subtracted-from as backorders are created and filled.
2. **A customer-product record does not exist in the `BackOrders` table** -- If the particular customer has never had a backorder for the product in question, then there is no record in the `BackOrders` table to update. If you attempt to update a nonexistent record, you will get an error.

What is required, therefore, is a means of determining whether a record already exists for a particular customer-product combination. If a record does exist, then it has to be updated; if a record does not

exist, then one has to be created. This is simple enough to talk about, but more difficult to implement in VBA. As a result, you are being provided with a shortcut function called `UpdateBackOrders()` that implements this logic.

The requirements for using the `UpdateBackOrders()` function are outlined in the following sections:

15.4.2.1 Create the `pqryItemsToBackOrder` query

If you have not already done so, create the `pqryItemsToBackOrder` query described in Section 10.4. The `UpdateBackOrders()` procedure sets the parameter for the query and then creates a recordset based on the results.

If you did not use the field names `OrderID`, and `ProductID` in your tables, you must use the calculated field syntax to rename them (see Section 15.3.2.4 to review renaming fields in queries).

Note that if the backordered quantity is positive, items are backordered. If the backordered quantity is negative, backorders are being filled. If the backordered quantity is zero, no change is required and these records should no be included in the results of the query.

15.4.2.2 Import the shortcut function

Import the Visual Basic for Applications (VBA) module containing the code for the `UpdateBackOrders()` function. This module is contained in an Access database called `BOSC_Vx.mdb` that you can download from the course home page.

- `BOSC_V2.mdb` is for those running Access version 2.0. To import the module, select *File > Import*, choose `BOSC_V2.mdb`, and select *Module* as the object type to import.
- `BOSC_V7.mdb` is for those running Access version 7.0 or higher. To import the module, select *File > Get External Data > Import*, choose `BOSC_V7.mdb`, and select *Module* as the object type to import.

15.4.2.3 Use the function in your application

The general syntax of the function call is: `UpdateBackOrders(OrderID, CustomerID)`.

The `OrderID` and `CustomerID` are arguments and they both must be of the type Long Integer. If this function is called properly, it will update all the backordered items returned by the parameter query.

15.4.2.4 Modifying the `UpdateBackOrders()` function

The `UpdateBackOrders()` function looks for specific fields in three tables: `BackOrders`, `Customers`, and `Products`. If any of your tables or fields are named differently, an error occurs. To eliminate these errors, you can do one of two of things:

1. Edit the VBA code. Use the search-and-replace feature of the module editor to replace all instances of field names in the supplied procedures with your own field names. This is the recommended approach, although you need an adequate understanding of how the code works in order to know which names to change.
2. Change the field names in your tables (and all queries and forms that reference these field names). This approach is not recommended.

15.4.3 Understanding the `UpdateBackOrders()` function

The flowchart for the `UpdateBackOrders()` function is shown in Figure 15.15. This function repeatedly calls a subroutine, `BackOrderItem`, which

updates or adds the individual items to the `BackOrders` table. The flowchart for the `BackOrderItem` subroutine is shown in Figure 15.16.

There are easier and more efficient ways of implementing routines to update the `BackOrders` table. Although some amount of VBA code is virtually inevitable, a great deal of programming can be eliminated by using parameter queries and action queries. Since queries run faster than code in Access, the more code you replace with queries, the better.

To get full marks for the backorders aspect of the assignment, you have to create a more elegant alternative to the shortcut supplied here.

FIGURE 15.15: Flowchart for `UpdateBackOrders()`.

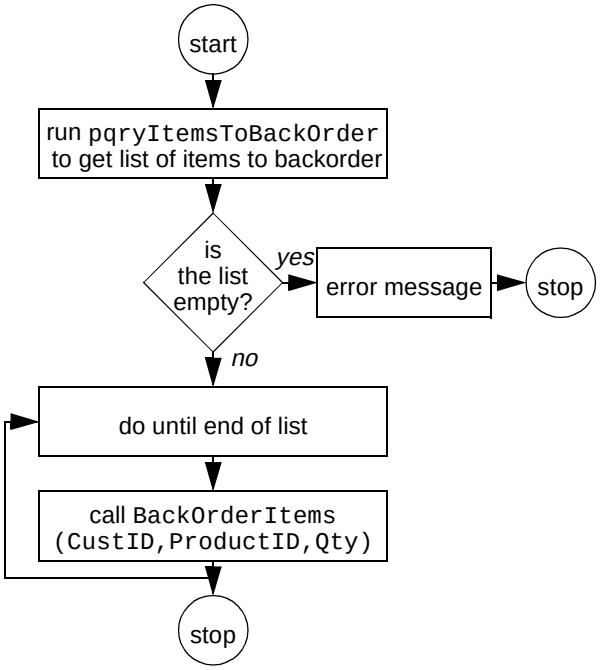
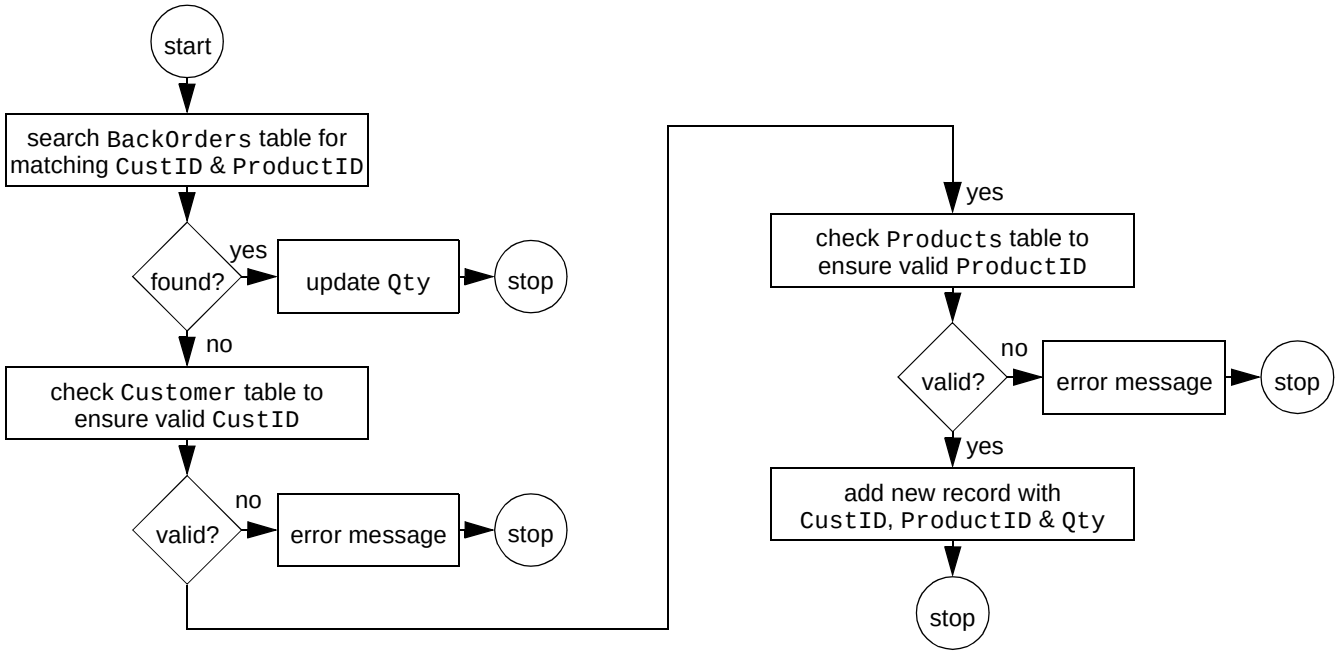


FIGURE 15.16: Flowchart for the `BackOrderItem` subroutine.



15.4.4 Annotated source code for the backorders shortcut module.

In the following sections, the two procedures in the shortcut module are examined. In each case, the code for the procedure is presented followed by comments on specific lines of code.

15.4.4.1 The `UpdateBackOrders()` function

```
Function UpdateBackOrders(ByVal
 lngOrdID As Long, ByVal lngCustID As
 Long)
Set dbCurr = CurrentDb
Dim rsBOItems As Recordset
dbCurr.QueryDefs!pqryItemsToBackOrder.
 Parameters!pOrderID = lngOrdID
Set rsBOItems =
 dbCurr.QueryDefs!pqryItemsToBackOrder
 .OpenRecordset()
If rsBOItems.RecordCount = 0 Then
MsgBox "Back order cannot be processed:
 order contains no items"
Exit Sub
End If
Do Until rsBOItems.EOF
Call BackOrderItem(lngCustID,
 rsBOItems!ProductID, rsBOItems!Qty)
rsBOItems.MoveNext
Loop
rsBOItems.Close
End Function
```

15.4.4.2 Explanation of the `UpdateBackOrders()` function

`Function UpdateBackOrders(ByVal lngOrdID As Long, ByVal lngCustID As Long)` — This statement declares the function and its parameters. Each item in the parameter list contains three elements: `ByVal` or `ByRef` (optional), the variable's name, and the variable's type (optional). The `ByVal`

keyword simply means that a copy of the variables value is passed the subroutine, not the variable itself. As a result, variables passed by value cannot be changed by the sub-procedure. In contrast, if a variable is passed by reference (the default), its value can be changed by the sub-procedure.

`Set dbCurr = CurrentDb` — Declaring a variable and setting it to be equal to something are distinct activities. In this case, the variable `dbCurr` (which is declared in the declarations section) is set to point to a database object. Note that the database object is not created, it already exists.

`CurrentDb` is a function supported in Access version 7.0 and higher that returns a reference to the current database. In Access version 2.0, this function does not exist and thus the current database must be found by starting at the top level object in the Access DAO hierarchy, as discussed in Section 14.3.1.

`Dim rsBOItems As Recordset` — In this declaration statement, a pointer to a Recordset object is declared. This recordset contains a list of all the items to add to the `BackOrders` table.

`dbCurr.QueryDefs!pqryItemsToBackOrder .Parameters!pOrderID = lngOrdID` — This one is a bit tricky: the current database (`dbCurr`) contains a collection of objects called QueryDefs (these are what you create when you use the QBE query designer). Within the collection of QueryDefs, there is one called `pqryItemsToBackOrder` (which you created in Section 15.4.2.1).

Within every QueryDef, there is a collection of zero or more **Parameters**. In this case, there is one called `pOrderID` and this sets the value of the parameter to the value of the variable `lngOrderID` (which was passed to the function as a parameter).

`Set rsBOItems = dbCurr.QueryDefs!pqryItemsToBackOrder.OpenRecordset()` — Here

is another set statement. In this one, the variable `rsBOItems` is set to point at a recordset object. Unlike the current database object above, however, this recordset does not yet exist and must be created by running the `pqryItemsToBackOrder` parameter query.

`OpenRecordset` is a method that is defined for objects of type TableDef or QueryDef that creates an image of the data in the table or query. Since the query in question is a parameter query, and since the parameter query is set in the previous statement, the resulting recordset consists of a list of backordered items with an order number equal to the value of `pOrderID`.

`If rsBOItems.RecordCount = 0 Then` — The only thing you need to know at this point about the *RecordCount* property of a recordset is that it returns zero if the recordset is empty.

`MsgBox "Back order cannot be processed: order contains no items"` — The `MsgBox` statement pops up a standard message box with an *Okay* button in the middle.

`Exit Sub` — If this line is reached, the list contains no items. As such, there is no need to go any further in this subroutine.

`End If` — The syntax for `If… Then… Else…` statements requires an `End If` statement at the end of the conditional code. That is, everything between the `If` and the `End If` executes if the condition is true; otherwise, the whole block of code is ignored.

`Do Until rsBOItems.EOF` — The `EOF` property of a recordset is set to true when the "end of file" is encountered.

`Call BackOrderItem(lngCustID, rsBOItems!ProductID, rsBOItems!Qty)` — A subroutine is used to increase the modularity and

readability of this function. Note the way in which the current values of `ProductID` and `Qty` from the `rsBOItems` Recordset are accessed.

`rsBOItems.MoveNext` — `MoveNext` is a method defined for recordset objects. If this is forgotten, the `EOF` condition will never be reached and an infinite loop will be created. In VBA, the *Escape* key is usually sufficient to stop an infinite loop.

`Loop` — All `Do While`/`Do Until` loops must end with the `Loop` statement.

`rsBOItems.Close` — When you create a new object (such as a Recordset using the `OpenRecordset` method), you should close it before exiting the procedure. Note that you do not close `dbCurr` because you did not open it.

`End Function` — All functions/subroutines need an `End Function`/`End Sub` statement.

15.4.4.3 The `BackOrderItem()` subroutine

```
Sub BackOrderItem(ByVal lngCustID As
  Long, ByVal strProdID As String, ByVal
  intQty As Integer)
Set dbCurr = CurrentDb
Dim strSearch As String
Dim rsBackOrders As Recordset
Set rsBackOrders =
  dbCurr.OpenRecordset("BackOrders",
  dbOpenDynaset)
strSearch = "CustID = " & lngCustID & "
  AND ProductID = '" & strProdID & "'"
rsBackOrders.FindFirst strSearch
If rsBackOrders.NoMatch Then
Dim rsCustomers As Recordset
Set rsCustomers =
  dbCurr.OpenRecordset("Customers",
  dbOpenDynaset)
strSearch = "CustID = " & lngCustID
rsCustomers.FindFirst strSearch
```

```
If rsCustomers.NoMatch Then
MsgBox “An invalid Customer ID number
 has been passed to BackOrderItem”
Exit Sub
End If
Dim rsProducts As Recordset
Set rsProducts =
 dbCurr.OpenRecordset(“Products”,
 dbOpenDynaset)
strSearch = “ProductID = '" & strProdID
 & “'”
rsProducts.FindFirst strSearch
If rsProducts.NoMatch Then
MsgBox “An invalid Product ID number
 has been passed to BackOrderItem”
Exit Sub
End If
rsBackOrders.AddNew
rsBackOrders!CustID = lngCustID
rsBackOrders!ProductID = strProdID
rsBackOrders!Qty = intQty
rsBackOrders.Update
Else
rsBackOrders.Edit
rsBackOrders!Qty = rsBackOrders!Qty +
 intQty
rsBackOrders.Update
End If
End Sub
```

15.4.4.4 Explanation of the `BackOrderItem()` subroutine

Since many aspects of the language are covered in the previous subroutine, only those that are unique to this subroutine are explained.

`Set rsBackOrders = dbCurr.OpenRecordset(“BackOrders”, dbOpenDynaset)` — The `OpenRecordset` method used here is the one defined for a Database object. The most important argument is the source of the records, which can be

a table name, a query name, or an SQL statement. The `dbOpenDynaset` argument is a predefined constant that tells Access to open the recordset as a dynaset. You don't need to know much about this except that the format of these predefined constants is different between Access version 2.0 and version 7.0 and higher. In version 2.0, constants are of the form: `DB_OPEN_DYNASET`.

`strSearch = “CustID = ”& lngCustID & “ AND ProductID = ’” & strProdID & “'”` — A string variable has been used to break the search process into two steps. First, the search string is constructed; then the string is used as the parameter for the `FindFirst` method. The only tricky part here is that `lngCustID` is a long integer and `strProdID` is a string. The difference is that the value of `strProdID` has to be enclosed in quotation marks when the parameter is passed to the *FindFirst* method. To do this, single quotes are used within the search string.

`rsBackOrders.FindFirst strSearch` — `FindFirst` is a method defined for Recordset objects that finds the first record that meets the criteria specified in the method's argument. Its argument is the text string stored in `strSearch`.

`If rsBackOrders.NoMatch Then` — The `NoMatch` property should always be checked after searching a record set. Since it is a Boolean variable (True / False) it can be used without an comparison operator.

`rsBackOrders.AddNew` — Before information can be added to a table, a new blank record must be created. The `AddNew` method creates a new empty record, makes it the active record, and enables it for editing.

`rsBackOrders!CustID = lngCustID` — Note the syntax for changing a variable's value. In this case, the null value of the new empty record is replaced with the value of a variable passed to the subroutine.

`rsBackOrders.Update` — After any changes are made to a record, the `Update` method must be invoked to "commit" the changes. The `AddNew` / `Edit` and `Update` methods are like bookends around changes made to records.

`rsBackOrders.Edit` — The `Edit` method allows the values in a record to be changed. Note that these changes are not saved to the underlying table until the `Update` method is used.