Microsoft Access Tutorials: Table of Contents

1. Introduction to Microsoft Access

Last update: 25-Aug-1997

2. Tables

3. Relationships

4. Basic Queries Using QBE

5. Basic Queries using SQL

6. Form Fundamentals

7. Subforms

8. Combo Box Controls

9. Advanced Forms

Access Tutorial 1: Introduction to Microsoft Access

The purpose of these tutorials is not to teach you Microsoft Access, but rather to teach you some generic information systems concepts and skills using Access. Of course, as a side effect, you will learn a great deal about the software—enough to write your own useful applications. However, keep in mind that Access is an enormously complex, nearly-industrial-strength software development environment. The material here only scrapes the surface of Access development and database programming.

1.1 Introduction: What is Access?

Microsoft Access is a relational database management system (DBMS). At the most basic level, a DBMS is a program that facilitates the storage and retrieval of structured information on a computer's hard drive. Examples of well-know industrial-strength relational DBMSes include

- Oracle
- Microsoft SQL Server
- IBM DB2
- Informix

Well-know PC-based ("desktop") relational DBMSes include

- Microsoft Access
- Microsoft FoxPro
- Borland dBase

1.1.1 The many faces of Access

Microsoft generally likes to incorporate as many features as possible into its products. For example, the Access package contains the following elements:

- a **relational database system** that supports two industry standard query languages: Structured Query Language (SQL) and Query By Example (QBE);

Last update: 24-Aug-1997

- a full-featured **procedural programming language**—essentially a subset of Visual Basic,
- a simplified procedural **macro language** unique to Access;
- a **rapid application development environment** complete with visual form and report development tools;
- a sprinkling of **objected-oriented extensions**; and,
- various **wizards and builders** to make development easier.

For new users, these "multiple personalities" can be a source of enormous frustration. The problem is that each personality is based on a different set of assumptions and a different view of computing. For instance,

- the relational database personality expects you to view your application as sets of data;
- the procedural programming personality expects you to view your application as commands to be executed sequentially;
- the object-oriented personality expects you to view your application as objects which encapsulate state and behavior information.

Microsoft makes no effort to provide an overall logical integration of these personalities (indeed, it is unlikely that such an integration is possible). Instead, it is up to you as a developer to pick and choose the best approach to implementing your application.

Since there are often several vastly different ways to implement a particular feature in Access, recognizing the different personalities and exploiting the best features (and avoiding the pitfalls) of each are important skills for Access developers.

The advantage of these multiple personalities is that it is possible to use Access to learn about an enormous range of information systems concepts without

having to interact with a large number of “single-personality” tools, for example:

- Oracle for relational databases
- PowerBuilder for rapid applications development,
- SmallTalk for object-oriented programming.

Keep this advantage in mind as we switch back and forth between personalities and different computing paradigms.

1.1.2 What is in an Access database file?

Although the term “database” typically refers to a collection of related data tables, an Access database includes more than just data. In addition to tables, an Access database file contains several different types of **database objects**:

- saved queries for organizing data,
- forms for interacting with the data on screen,
- reports for printing results,
- macros and Visual Basic programs for extending the functionality of database applications.

All these database objects are stored in a single file named `<filename>.mdb`. When you are running Access, a temporary “locking” file named `<filename>.ldb` is also created. You can safely ignore the *.ldb file; everything of value is in the *.mdb file.

1.2 Learning objectives

- ❒ How do I get started?
- ❒ How do I determine the version I am using?
- ❒ How do I create or edit a database object?
- ❒ What is the database window and what does it contain?
- ❒ How do I import an Excel spreadsheet?
- ❒ How do I delete or rename database objects?

- ❒ How do I get help from the on-line help system?
- ❒ How do I compact a database to save space?

1.3 Tutorial exercises

In this tutorial, you will start by creating a new database file.

1.3.1 Starting Access

- To start Access, you double click the Access icon ( for version 8.0 and 7.0 or  for version 2.0) from within Microsoft Windows.

If you are working in the Commerce PC Lab, you will be working with Access version 2.0. If you are working at home, you will able be to tell what version you are using by watching the screen “splash” as the program loads. Alternatively, select *Help > About Access* from the main menu to see which version you are using.

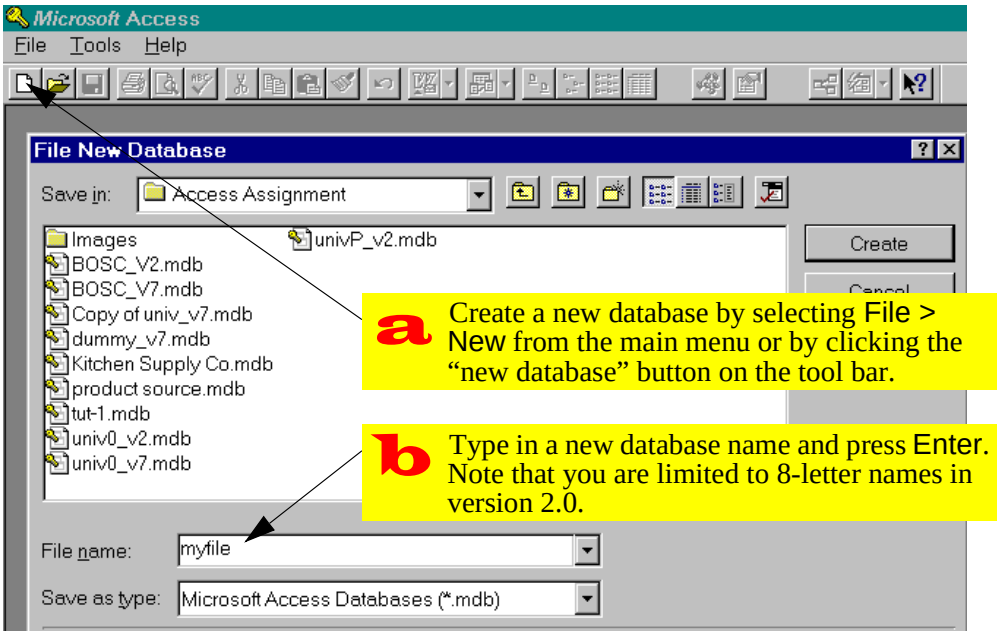
All the screen shots in these tutorials are taken from Access version 7.0 (released as part of Office 95). Although there are some important differences between version 2.0 and version 7.0, the concepts covered here are the same for both. Version 8.0 (released as part of Office 97) is only slightly different from version 7.0.

Whenever the instructions given in the tutorial differ significantly from version 7.0, a warning box such as this is used.

1.3.2 Creating a new database

- Follow the directions in Figure 1.1 to create a new database file called `myfile.mdb`.

FIGURE 1.1: Select the name and location of your new (empty) database.



- Examine the main features of the database window—including the tabs for viewing the different database objects—as shown in Figure 1.2.

1.3.3 Opening an existing database

Since an empty database file is not particularly interesting, you are provided with an existing database file containing information about university courses. For the remainder of this tutorial, we will use a file called `univ0_v7.mdb`, which is available from the tutorial's Internet site.

If you are using version 2.0, you will need to use the `univ0_v2.mdb` database instead. Although you can open a version 2.0 database with version 7.0, you cannot open a version 7.0 database with version 2.0. Importing and exporting across versions is possible, however.

If you are using version 8.0, you can use either `univ0_v2.mdb` or `univ0_v7.mdb` for the tutorials. When you open the file, Access will ask you if you want to convert it to version 8.0. Select yes and provide a new name for the converted file (e.g., `univ0_v8.mdb`)

- Open the `univ0_vx.mdb` file and examine the contents of the `Sections` table, as shown in Figure 1.3.

1.3.4 Importing data from other applications

Access makes it easy to import data from other applications. In this section, you will create a new table using data from an Excel spreadsheet.

- Select *File > Get External Data > Import* from the main menu and import the `depts.xls` spread-

FIGURE 1.2: The *database window* contains all the *database objects* for a particular application.

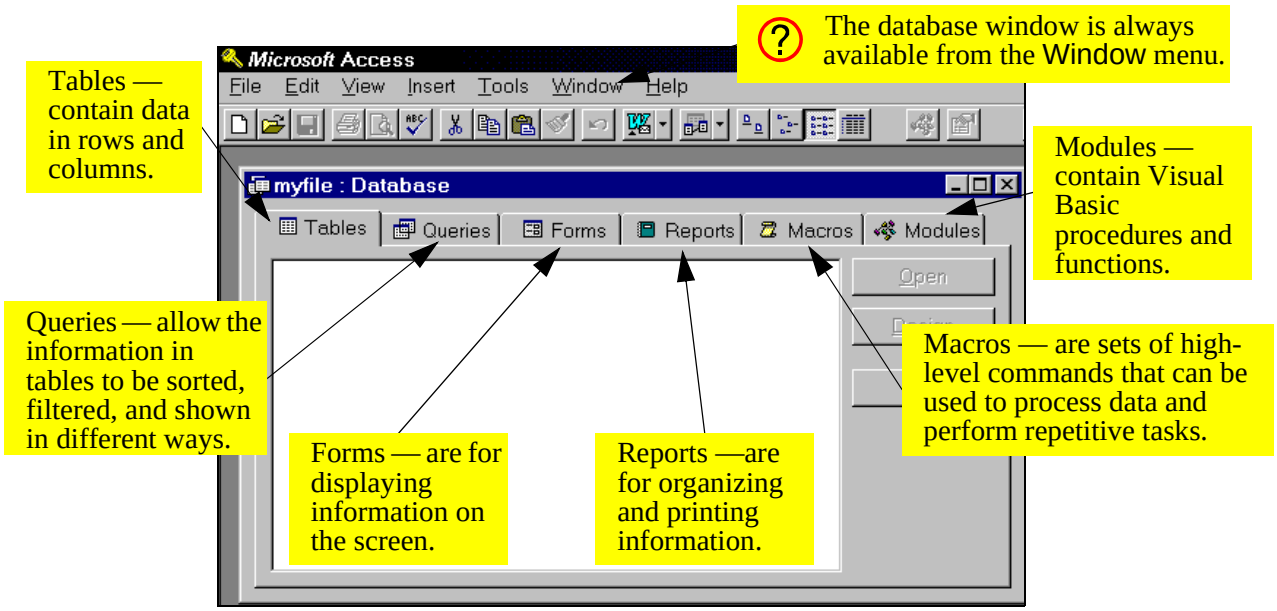
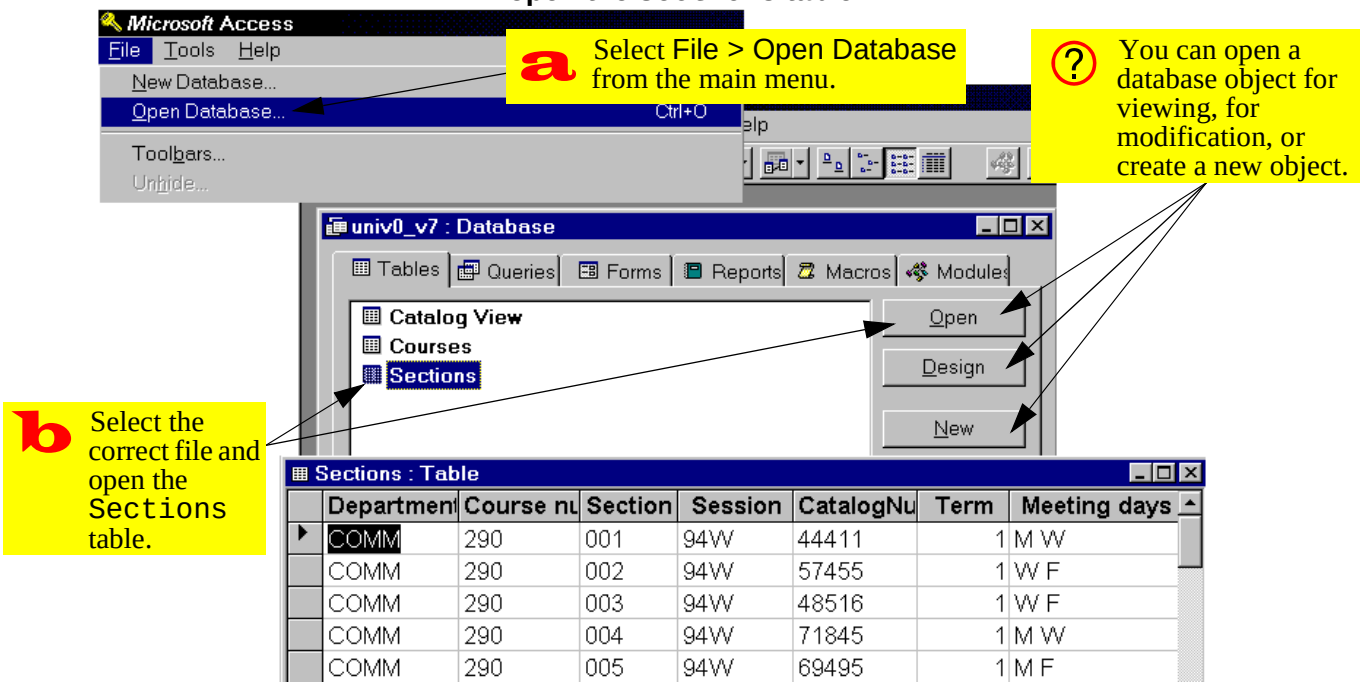


FIGURE 1.3: Open the `univ0_vx.mdb` file for the version of Access that you are using and then open the `Sections` table



sheet as a new table called `Departments` (see Figure 1.4).

In version 2.0, the menu structure is slightly different. As such, you must use *File > Import*.

- Use the import wizard specify the basic import parameters. You should accept all the defaults provided by the wizard except for those shown in Figure 1.5.
- Double click the `Departments` table to ensure it was imported correctly.

If you make a mistake, you can rename or delete a table (or any database object in the database window) by selecting it and right-clicking (pressing the right mouse button once).

1.3.5 Getting help

A recent trend in commercial software (especially from Microsoft) is a reliance on on-line help and documentation in lieu of printed manuals. As a consequence, a good understanding of how to use the on-line help system is essential for learning any new software. In this section, you will use Access' on-line help system to tell you how to compact a database.

- Press *F1* to invoke the on-line help system. Find information on compacting a database, as shown in Figure 1.6.
- Familiarize yourself with the basic elements of the help window as shown in Figure 1.7.

1.3.6 Compacting your database

- Follow the directions provided by the on-line help window shown in Figure 1.7 to compact your database.

FIGURE 1.4: Import the `dept.xls` spreadsheet as a table called `Departments`.

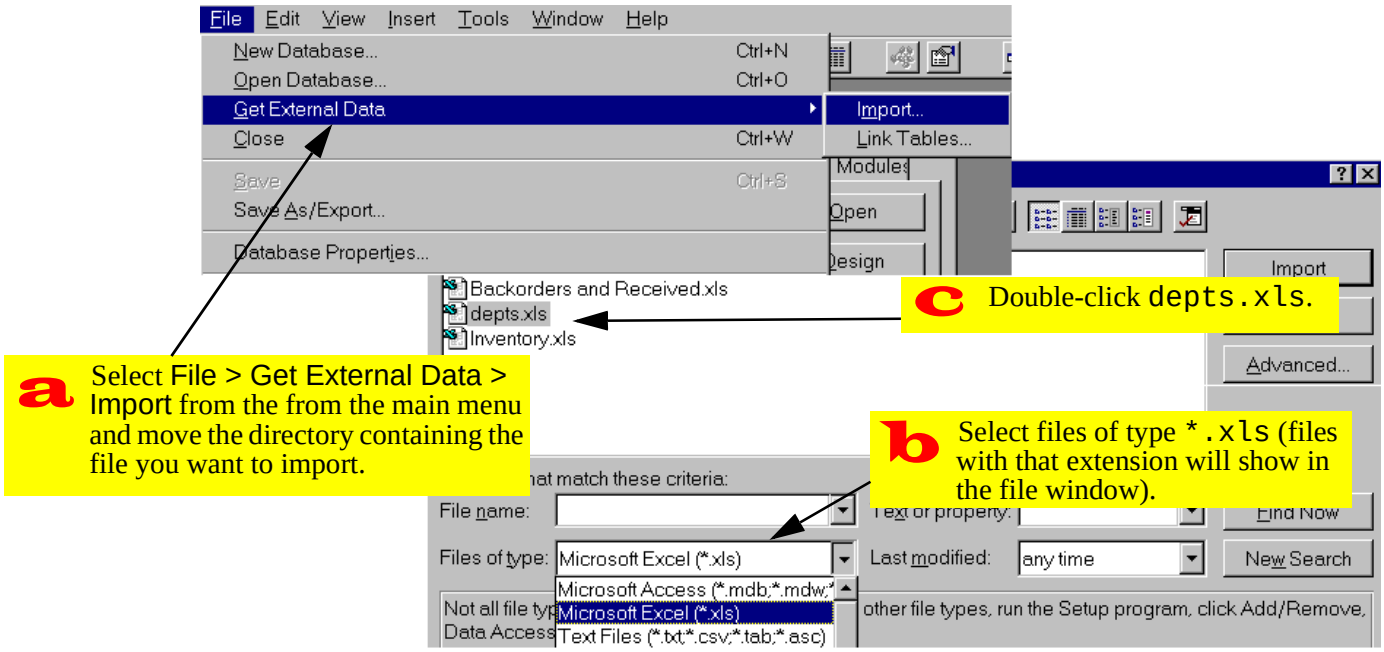


FIGURE 1.5: Use the spreadsheet import wizard to import the Excel file.

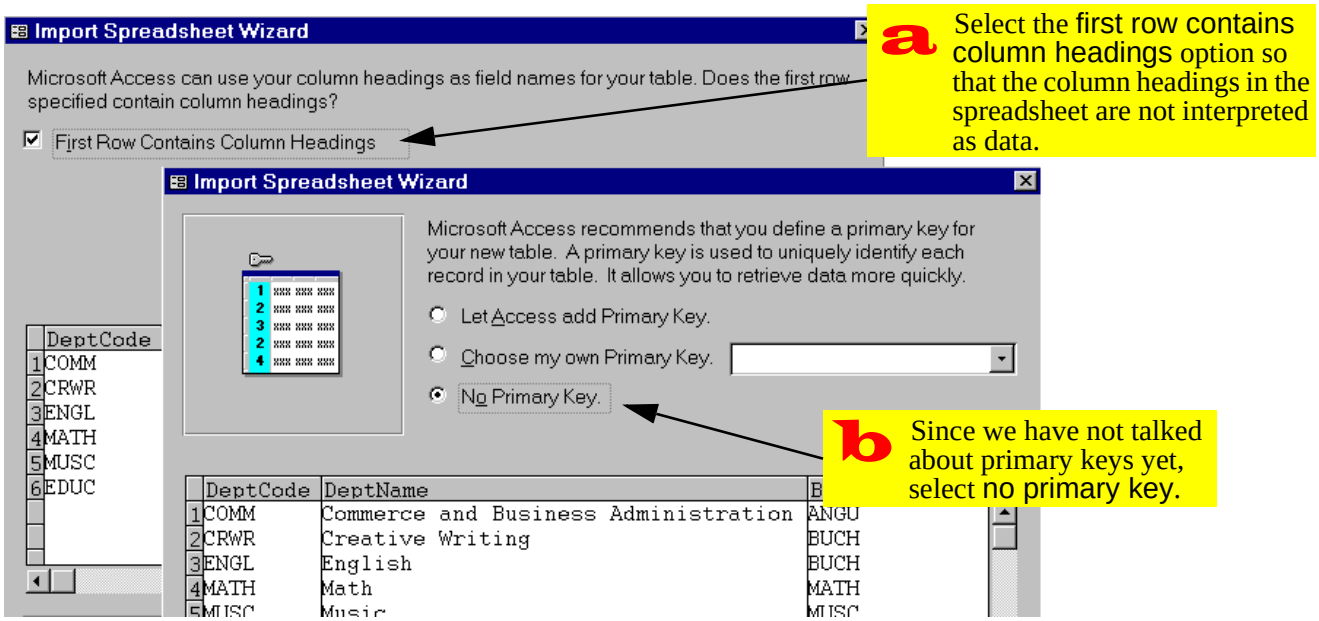


FIGURE 1.6: Use the help system to find information on a specific topic

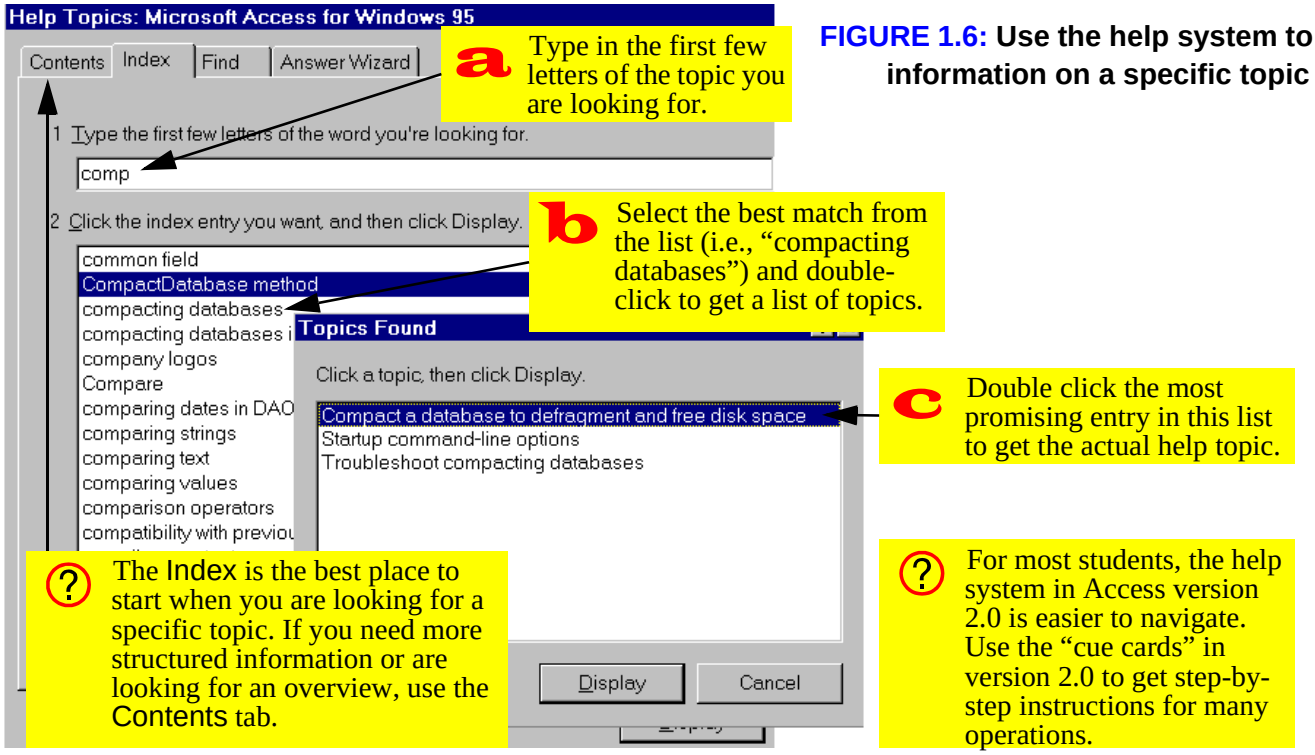
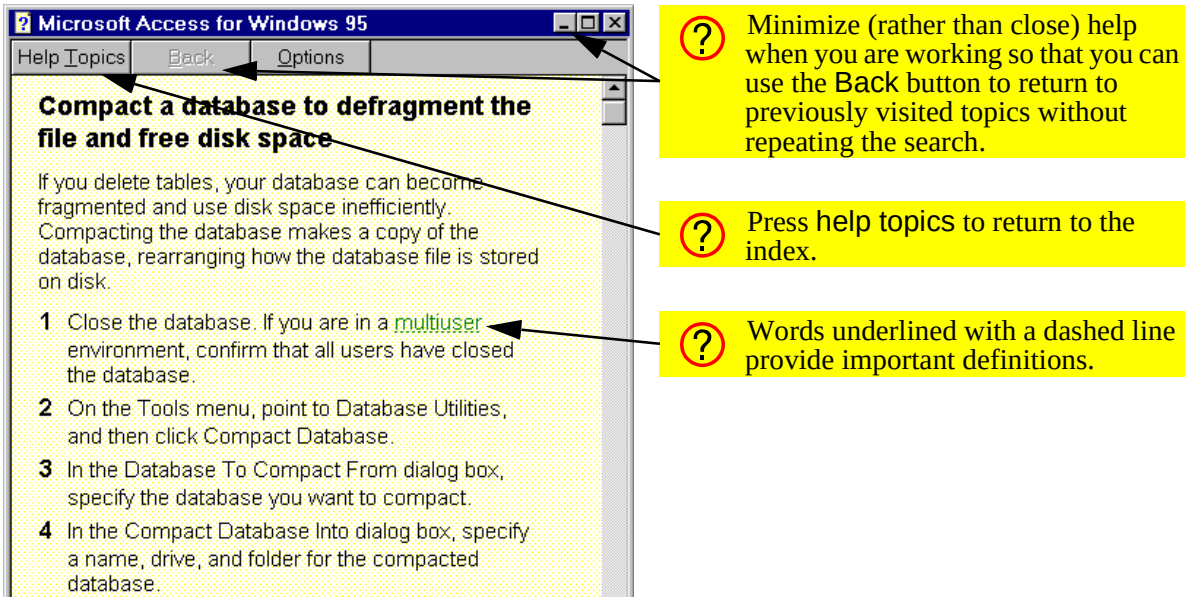


FIGURE 1.7: Follow the instructions provided by help to compact your database



1.4 Discussion

1.4.1 The database file in Access

The term “database” means different things depending on the DBMS used. For example in dBase IV, a database is a file (`<filename>.dbf`) containing a single table. Forms and reports are also stored as individual files with different extensions. The net result is a clutter of files.

In contrast, an Oracle database has virtually no relationship to individual files or individual projects. For instance, a database may contain many tables from different projects/applications and may also be stored split into one or more files (perhaps on different machines).

Access strikes a convenient balance—all the “objects” (tables, queries, forms, reports, etc.) for a single project/application are stored in a single file.

1.4.2 Compacting a database

As the help system points out, Access database files can become highly fragmented and grow to become much larger than you might expect given the amount of data they contain (e.g., multiple megabytes for a handful of records). Compacting the database from time to time eliminates fragmentation and can dramatically reduce the disk space requirement of your database.

1.4.3 Renaming a database

It is often the case that you are working with a database and want to save it under a different name or save it on to a different disk drive. However, one command on the *File* menu that is conspicuous by its absence is *Save As*.

However, when compacting your database, Access asks for the name and destination of the compacted file. As a result, the compact database utility can be

used as a substitute for the *Save As* command. This is especially useful in situations in which you cannot use the operating system to rename a file (e.g., when you do not have access to the Windows file manager).

1.4.4 Developing applications in Access

In general, there are two basic approaches to developing information systems:

- in-depth systems analysis, design, and implementation,
- rapid prototyping (in which analysis, design, and implementation are done iteratively)

Access provides a number of features (such as graphical design tools, wizards, and a high-level macro language) that facilitate rapid prototyping. Since you are going to build a small system and since time is limited, you will use a rapid prototyping approach to build your application. The recommended sequence for prototyping using Access is the following:

1. Model the information of interest in terms of entities and relationships between the entities (this is covered in the lecture portion of the course).
2. Create a table for each entity (Tutorial 2).
3. Specify the relationships between the tables (Tutorial 3).
4. Organize the information in your tables using queries (Tutorial 4, Tutorial 5, Tutorial 10)
5. Create forms and reports to support input and output transactions (Tutorial 6, Tutorial 7).
6. Enhance you forms with input controls (Tutorial 8)
7. Create action queries (Tutorial 11), macros (Tutorial 13), or Visual Basic programs (Tutorial 12, Tutorial 14) to perform the transaction processing functions of the application.

8. Create “triggers” (procedures attached to events) to automate certain repetitive tasks (Tutorial 15).

1.4.5 Use of linked tables

Most professional Access developers do not put their tables in the same database file as their queries, forms, reports, and so on. The reason for this is simple: keep the application’s data and interface separate.

Access allows you to use the “linked table” feature to link two database files: one containing all the tables (“data”) and another containing all the interface and logic elements of the application (“interface”). The linked tables from the data file show up in the interface file with little arrows (indicating that they are not actually stored in the interface file).

In this way, you can modify or update the interface file without affecting the actual data in any way. You just copy the new interface file over to the user’s machine, update the links to the data file, and the upgrade is done.

Do not used linked tables in the assignment. The links are dependent on the absolute directory structure. As a result, if the directory structure on your machine is different from that on the marker’s machine, the marker will not be able to use your application without first updating the links (a time consuming process for a large number of assignments).

1.5 Application to the assignment

After completing this tutorial you should be ready to create the database file that you will use for the remainder of the course.

1. Create an empty database file called `<your groupID>.mdb`. Remember that your group number consists of eight digits.

2. Import the `inventor.xls` spreadsheet as your `Products` table.
3. Use the compact utility to make a backup copy of your database (use a different name such as `backup.mdb`).

Access Tutorial 2: Tables

2.1 Introduction: The importance of good table design

Tables are where data in a database is stored; consequently, tables form the core of any database application. In addition to basic data, Access permits a large amount of domain knowledge (such as captions, default values, constraints, etc.) to be stored at the table level.

Extra time spent thinking about table design can result in enormous time savings during later stages of the project. Non-trivial changes to tables and relationships become increasingly difficult as the application grows in size and complexity.

2.2 Learning objectives

- How do I enter and edit data in the datasheet view of a table?
- How do I create a new table?
- How do I set the primary key for a table?
- How do I specify field properties such as the input mask and caption?
- Why won't an autonumber field restart counting at one?
- What are the different types of keys?

2.3 Tutorial exercises

In this tutorial, you will learn to interact with existing tables and design new tables.

Last update: 25-Aug-1997

2.3.1 Datasheet basics

- If you have not already done so, open the `univ0_vx.mdb` database file from Tutorial 1.
- Open the `Departments` table. The important elements of the **datasheet** view are shown in Figure 2.1.
- Use the field selectors to adjust the width of the `DeptName` field as shown in Figure 2.1.
- Add the Biology department (BIOL) to the table, as shown in Figure 2.2.
- Delete the "Basket Weaving" record by clicking on its **record selector** and pressing the *Delete* key.

2.3.2 Creating a new table

In this section you will create and save a very basic skeleton for table called `Employees`. This table could be used to keep track of university employees such as lecturers, department heads, departmental secretaries, and so on.

- Return to the database window and create a new table as shown in Figure 2.3.
- In the **table design window** shown in Figure 2.4, type in the following information:

Field name	Data type	Description (optional)
`EmployeeID`	Text	use employee S.I.N.
`FName`	Text	First name
`LName`	Text	Last name
`Phone`	Text	
`Salary`	Currency	

- Select *File > Save* from the main menu (or press *Control-S*) and save the table under the name `Employees`.

FIGURE 2.1: The datasheet view of the `Departments` table.

The field names are shown in the "field selectors" across the top of the columns.

You can temporarily sort the records in a particular order by right-clicking any of the field selectors.

The records are shown as rows.

The black triangle indicates the "current record".

The grey boxes are "record selectors".

a Resize the `DeptName` column by clicking near the column border and dragging the border to the right.

Departments : Table

	DeptCode	DeptName	Building
▶	BSKW	Basket Weaving	ANGU
	COMM	Commerce and	ANGU
	CRWR	Creative Writing	BUCH
	EDUC	Education	SCRF
	ENGL	English	BUCH
	MATH	Math	MATH
	MUSC	Music	
*			

The asterisk (*) indicates a place holder for a new record.

The "navigation buttons" at the bottom of the window indicate the current record number and allow you to go directly to the first, previous, next, last, or new record.

Record: 1 of 7

FIGURE 2.2: Adding and saving a record to the table.

a Add a new record by clicking in the `DeptCode` field of the "new record" field (marked by the asterisk).

Departments : Table

	DeptCode	DeptName	Building
	BSKW	Basket Weaving	ANGU
	COMM	Commerce and Business Administration	ANGU
	CRWR	Creative Writing	BUCH
	EDUC	Education	SCRF
	ENGL	English	BUCH
	MATH	Math	MATH
	MUSC	Music	MUSC
▶	BIOL	Biology	BIOL

b To permanently save the change to the data, click on the record selector (note the icon changes from a pencil to a triangle).

It is seldom necessary to explicitly save new records (or changes to existing records) since Access automatically saves whenever you move to another record, close the table, quit Access, etc.

FIGURE 2.3: Create a new table.

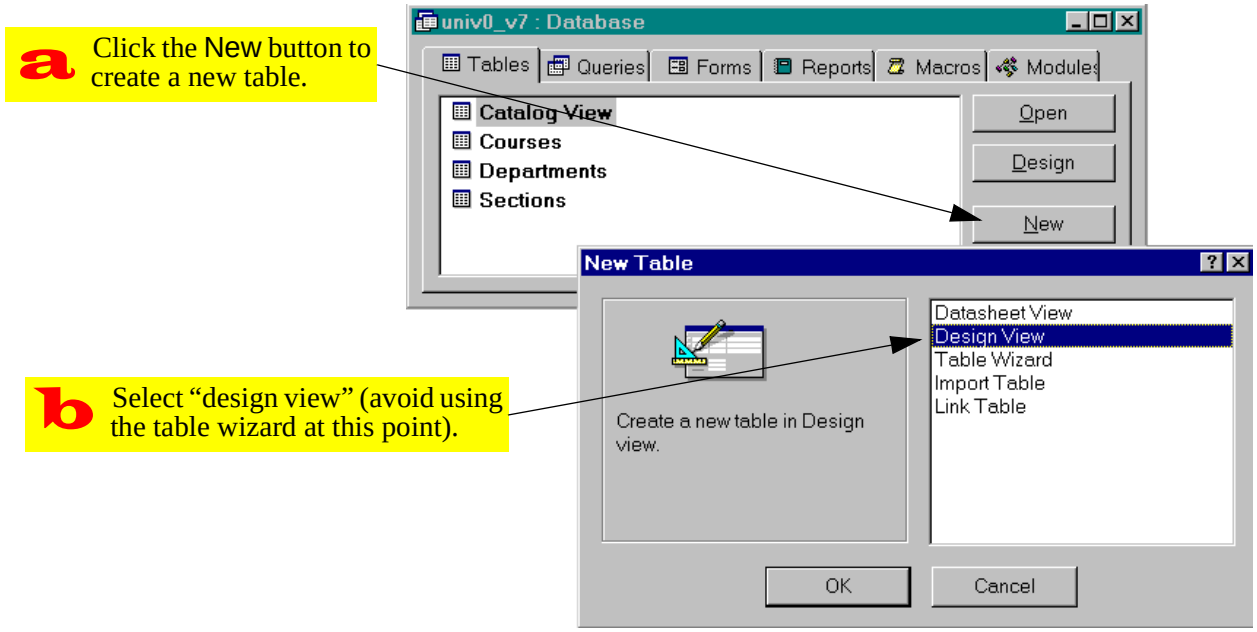
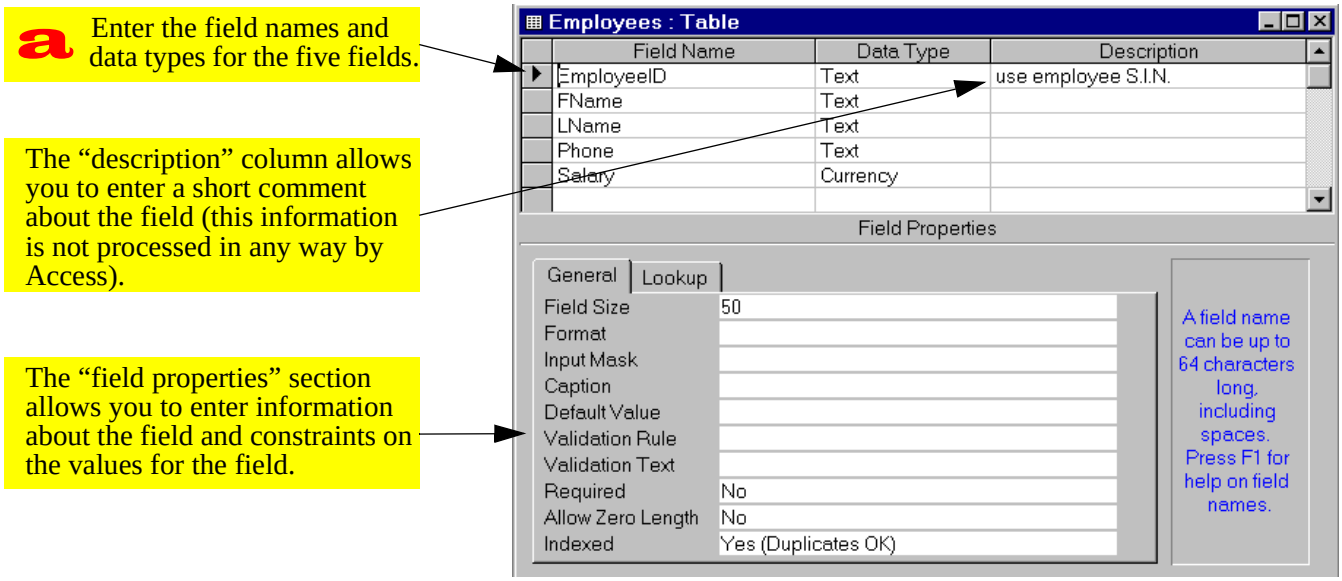


FIGURE 2.4: Use the table design window to enter the field properties for the `Employees` table.



2.3.3 Specifying the primary key

Tables normally have a primary key that uniquely identifies the records in the table. When you designate a field as the primary key, Access will not allow you to enter duplicate values into the field.

- Follow the steps in Figure 2.5 to set the primary key of the table to `EmployeeID`.

2.3.4 Setting field properties

In this section, you will specify a number of field properties for the `EmployeeID` field, as shown in Figure 2.6.

- Since we are going to use the employees' Social Insurance Number (S.I.N.) to uniquely identify them, set the *Field Size* property to 11 characters (9 for numbers and 2 for separating spaces)
- Set the *Input Mask* property to the following: `000\ 000\ 000;0`
- Set the *Caption* property to `Employee ID`

FIGURE 2.6: Set the field properties for the `EmployeeID` field.

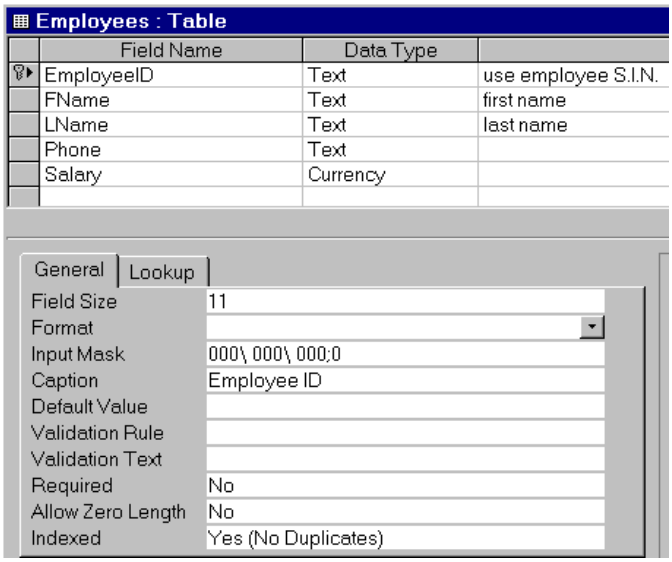
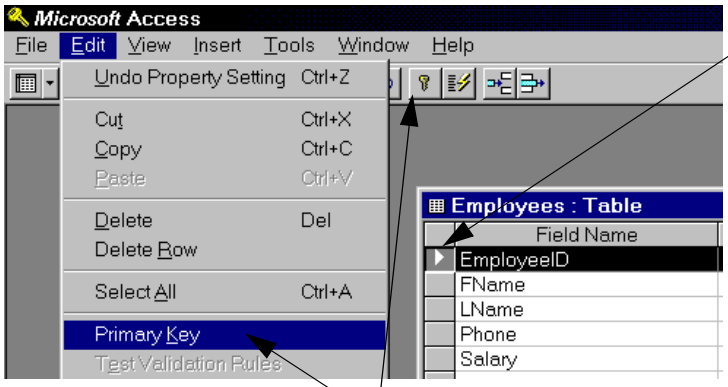


FIGURE 2.5: Set the primary key for the `Employees` table.



a Click on the grey box beside the field (or fields) that form the primary key.

? To select more than one field for use as the primary key, hold down the Control key while clicking on the grey boxes.

b Either click the key-shaped icon in the tool bar or select Edit > Primary Key from the menu.

- Select *View > Datasheet* from the main menu to switch to datasheet mode as shown in Figure 2.7. Enter your own S.I.N. and observe the effect of the input mask and caption on the `EmployeeID` field.
- Select *View > Table Design* from the main menu to return to design mode.
- Set the field properties for `FName` and `LName` (note that *Length* and *Caption* are the only two properties that are relevant for these two fields)

2.3.5 Using the input mask wizard

In this section, you will use the input mask wizard to create a complex input mask for a standard field type. You will also use the help system to learn more about the meaning of the symbols used to create input masks.

- Select the `Phone` field, move the cursor to the input mask property, and click the button with three small dots (...) to invoke the input mask wizard.
- Follow the instructions provided by the wizard as shown in Figure 2.8.
- Press *F1* while the cursor is still in the input mask property. Scroll down the help window to find the meaning of the “0”, “9”, “>” and “L” input mask symbols.

2.4 Discussion

2.4.1 Key terminology

A key is one or more fields that uniquely determine the identity of the real-world object that the record is meant to represent. For example, there is a record in the student information system that contains information about you as a student. To ensure that the record is associated with you and only you, it con-

FIGURE 2.7: Observe the effect of the input mask and caption properties on the behavior of the `EmployeeID` field during data entry

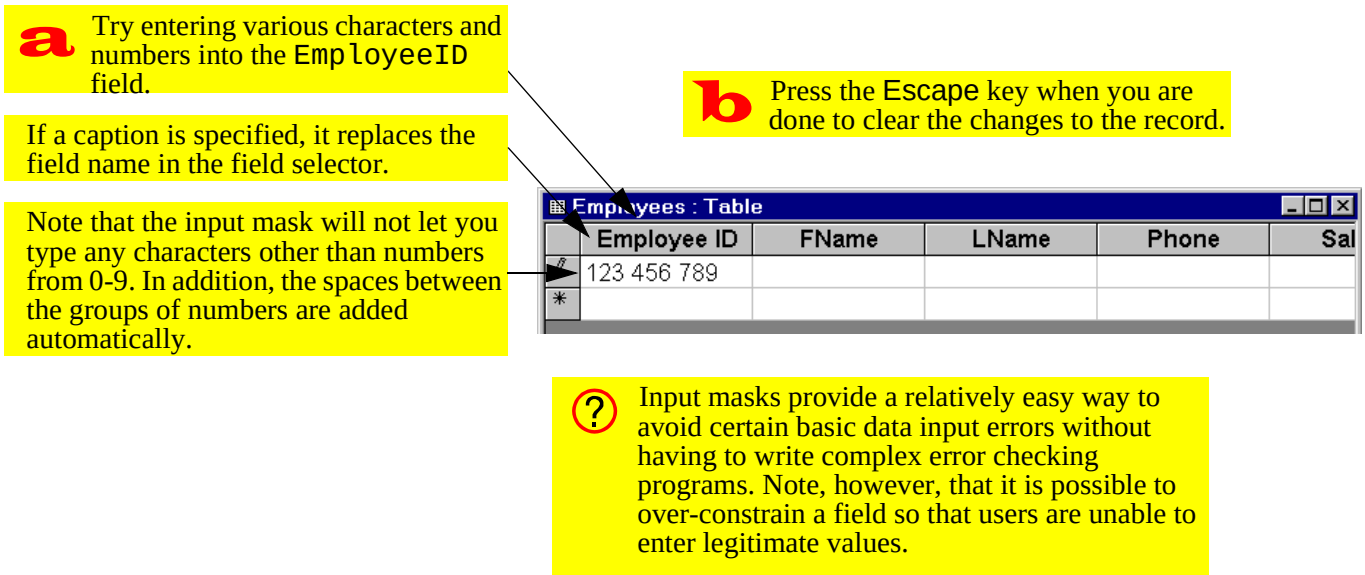
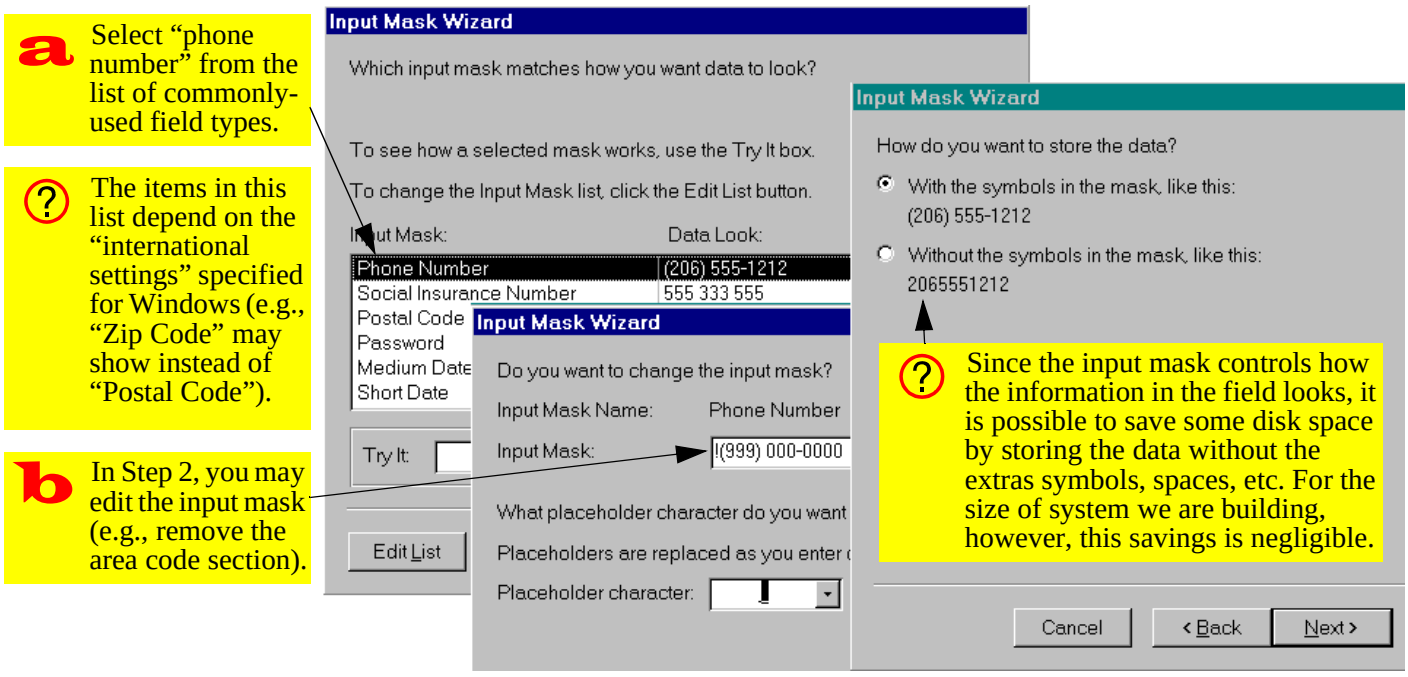


FIGURE 2.8: Use the input mask wizard to create an input mask.



tains a field called "student number" that is guaranteed to be unique.

The advantage of using student number as a key instead of some other field—like "student name"—is that there may be more than one person with the same first and last name. The combination of student name and address is probably unique (it is improbable that two people with the same name will at the same address) but using these two fields as a key would be cumbersome.

Since the terminology of keys can be confusing, the important terms are summarized below.

1. **Primary key** — The terms "key" and "primary key" are often used interchangeably. Since there may be more than one **candidate** key for an application, the designer has to select one: this is the primary key.
2. **Concatenated key**: The verb "concatenate" means to join together in a series. A concatenated key is made by joining together two or more fields. Course numbers at UBC provide a good example of a concatenated key made by joining together two fields: `DeptCode` and `CrsNum`. For example, department alone cannot be the primary key since there are many courses in each department (e.g., COMM 335, COMM 391). Similarly, course number cannot be used as a key since there are many courses with the same number in different departments (e.g., COMM 335, HIST 335, MATH 335). However, department and course number together form a concatenated key (there is only one COMM 335).
3. **Foreign key**: In a one-to-many relationship, a foreign key is a field (or fields) in the "child" record that uniquely identifies the correct "parent" record. For example, `DeptCode` and `CrsNum` in the `Sections` table are foreign keys since these two keys taken together are the primary key of

the Courses table. Foreign keys are identified in Access by creating relationships (see Tutorial 3).

2.4.2 Fields and field properties

2.4.2.1 Field names

Access places relatively few restrictions on field names and thus it is possible to create long, descriptive names for your fields. The problem is that you have to type these field names when building queries, macros, and programs. As such, a balance should be struck between readability and ease of typing. You are advised to use short-but-descriptive field names with no spaces.

For example, in Section 2.3.2 you created a field with name FName. However, you can use the caption property to provide a longer, more descriptive label such as First name. The net result is a field name that is easy to type when programming and a field caption that is easy to read when the data is viewed.

In addition, you can use the comment field in the table design window to document the meaning of field names.

It is strongly recommended that you avoid all non-alphanumeric characters whenever you name a field or database object. Although Access will permit you to use names such as Customer#, non-alphanumeric characters (such as #, /, $, %, ~, @, etc.) may cause undocumented problems later on.

2.4.2.2 Data types

The field's data type tells Access how to handle the information in the field. For instance, if the data type is date/time, then Access can perform date/time arithmetic on information stored in the field. If the same date is stored as text, however, Access treats it just like any other string of characters. Normally,

the choice of data type is straightforward. However, the following guidelines should be kept in mind:

1. Do not use a **numeric** data type unless you are going to treat the field as a number (i.e., perform mathematical operations on it). For instance, you might be tempted to store a person's student number as an integer. However, if the student number starts with a zero, then the first digit is dropped and you have to coerce Access into displaying it. Similarly, a UBC course number (e.g., 335) might be considered a number; however, since courses like 439B have to accommodated, a numeric data type for the course number field is clearly inappropriate.
2. Access provides a special data type called **Auto Number** (**Counter** in version 2.0). An autonumber/counter is really a number of type **Long Integer** that gets incremented by Access every time a new record is added. As such, it is convenient for use as a primary key when no other key is provided or is immediately obvious.

Since an autonumber is really Long Integer and since relationships can only be created between fields with the same data type, it is important to remember that if an autonumber is used on the “one” side of a relationship, a long integer must be used for the “many” side.

2.4.2.3 “Disappearing” numbers in autonumber fields

If, during the process of testing your application, you add and delete records from a table with an autonumber key, you will notice that the deleted keys are not reclaimed.

For instance, if you add records to your Customer table (assuming that CustID is an autonumber), you will have a series of CustID values: 1, 2, 3… If you

later delete customer 1 and 2, you will notice that your list of customers now starts at 3.

Clearly, it would be impossible for Access to renumber all the customers so the list started at 1. What would happen, for instance, to all the printed invoices with `CustID` = 2 on them? Would they refer to the original customer 2 or the newly renumbered customer 2?

The bottom line is this: once a key is assigned, it should never be reused, even if the entity to which it is assigned is subsequently deleted. Thus, as far as you are concerned, there is no way to get your customers table to renumber from `CustID` = 1.

Of course, there is a long and complicated way to do it, but since used an autonumber in the first place, you do not care about the actual value of the key—you just want it to be unique. In short, it makes absolutely no difference whether the first customer in your customers table is `CustID` = 1 or 534.

2.4.2.4 Input masks

An input mask is a means of restricting what the user can type into the field. It provides a "template" which tells Access what kind of information should be in each space. For example, the input mask `>LLLL` consists of two parts:

1. The right brace > ensures that every character the user types is converted into upper case. Thus, if the user types `comm`, it is automatically converted to `COMM`.
2. The characters `LLLL` are place holders for letters from A to Z with blank spaces not allowed. What this means is that the user has to type in exactly four letters. If she types in fewer than four or types a character that is not within the A to Z scope (e.g., &, 7, %), Access will display an error message.

There are a large number of special symbols used for the input mask templates. Since the meaning of many of the symbols is not immediately obvious, there is no requirement to remember the character codes. Instead, simply place the cursor on the input mask property and press *F1* to get help. In addition, the wizard can be used to provide a basic input mask which can later be modified.

2.4.2.5 Input masks and literal values

To have the input mask automatically insert a character (such as a space or a dash) in a field, use a slash to indicate that the character following it is a literal.

For example, to create an input mask for local telephone numbers (e.g., 822-6109), you would use the following template: `000\-0000;0` (the dash is a literal value and appears automatically as the user enters the telephone number).

The semicolon and zero at the end of this input mask are important because, as the on-line help system points out, an input mask value actually consists of three parts (or "arguments"), each separated by a semicolon:

- the actual template (e.g., `000\-0000`),
- a value (0 or 1) that tells Access how to deal with literal characters, and
- the character to use as a place holder (showing the user how many characters to enter).

When you use a literal character in an input mask, the second argument determines whether the literal value is simply displayed or displayed *and* stored in the table as part of the data.

For example, if you use the input mask `000\-0000;1`, Access will not store the dash with the telephone number. Thus, although the input mask will always display the number as "822-6109", the number is actually stored as "8226109". By using the

input mask `000\-0000;0`, however, you are telling Access to store the dash with the rest of the data.

If you use the wizard to create an input mask, it asks you a simple question about storing literal values (as shown in Figure 2.8) and fills in the second argument accordingly. However, if you create the input mask manually, you should be aware that by default, Access does not store literal values. In other words, the input mask `000\-0000` is identical to the input mask `000\-0000;1`. This has important consequences if the field in question is subject to referential integrity constraints (the value “822-6109” is not the same as “8226109”).

2.5 Application to the assignment

You now have the skills necessary to implement your tables.

- Create all the tables required for the assignment.
- Use the autonumber data type (counter in version 2.0) for your primary keys where appropriate.
- Specify field properties such as captions, input mask, and defaults where appropriate.

If you create an input mask for `ProductID`, ensure you understand the implications of Section 2.4.2.5.

- Set the *Default* property of the `OrderDate` field so that the current date is automatically inserted into the field when a new order is created (hint: see the `Date()` function in the on-line help system).

- Do not forget to modify your `Products` table (the data types, lengths, and field properties of imported tables normally need to be fine tuned)
- Populate (enter data into) your master tables. Do not populate your transaction tables.

For the purpose of the assignment, the term “transaction” tables refers to tables that contain information about individual transactions (e.g., `Orders`, `OrderDetails`, `Shipments`, `ShipmentDetails`). “Master” tables, in contrast, are tables that either do not contain information about transactions (e.g., `Customers`) or contain only summary or status information about transactions (e.g., `BackOrders`).

Access Tutorial 3: Relationships

3.1 Introduction: The advantage of using tables and relationships

A common mistake made by inexperienced database designers (or those who have more experience with spreadsheets than databases) is to ignore the recommendation to model the domain of interest in terms of entities and relationships and to put all the information they need into a single, large table. Figure 3.1 shows such a table containing information about courses and sections.

- If you have not already done so, open the `univ0_vx.mdb` database.
- Open the `Catalog View` table.

The advantage of the single-table approach is that it requires less thought during the initial stages of application development. The disadvantages are too numerous to mention, but some of the most important ones are listed below:

1. Wasted space — Note that for COMM 290, the same basic course information is repeated for every section. Although the amount of disk space wasted in this case is trivial, this becomes an important issue for very large databases.
2. Difficulty in making changes — What happens if the name of COMM 290 is changed to “Mathematical Optimization”? This would require the same change to be made eight times. What if the person responsible for making the change forgets to change all the sections of COMM 290? What then is the “true” name of the course?
3. Deletion problems — What if there is only one section of COMM 290 and it is not offered in a particular year? If section 001 is deleted, then the system no longer contains any information about the course itself, including its name and number of credits.

FIGURE 3.1: The “monolithic” approach to database design—the `Catalog View` table contains information about courses and sections.

The course “COMM 290” consists of many sections.

Each section has some information unique to that section (such as `Time`, `Days`, `Building`, `Room`); however, the basic course information (e.g., `Title`, `Credits`) is the same for all sections of a particular course.

Catalog View : Table

CatalogNum	DeptCode	CrsNum	Title	Section
34134	COMM	290	Introduction to Quai	006
44411	COMM	290	Introduction to Quai	001
69495	COMM	290	Introduction to Quai	005
57455	COMM	290	Introduction to Quai	002
48516	COMM	290	Introduction to Quai	003
71845	COMM	290	Introduction to Quai	004
45938	COMM	290	Introduction to Quai	007
27839	COMM	290	Introduction to Quai	008
83920	COMM	291	Applied Statistics in	002
30293	COMM	291	Applied Statistics in	003

4. Addition problems — If a new section is added to any course, all the course information has to be typed in again. Not only is this a waste of time, it increases the probability of introducing errors into the system.

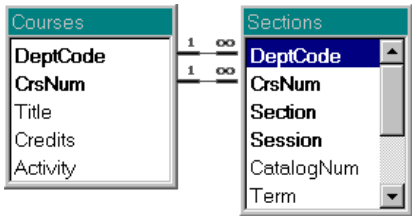
3.1.1 "Normalized" table design

The problems identified above can be avoided by spitting the `Catalog View` table into two separate tables:

1. `Courses`— information about courses only
2. `Sections` — information about sections only.

The key to making this work is to specify a relationship between `Courses` and `Sections` so that when we look at a section, we know which course it belongs to (see Figure 3.2). Since each course can have one or more sections, such a relationship is called "one-to-many".

FIGURE 3.2: A one-to-many relationship between `Courses` and `Sections`.



Access uses relationships in the following way: Assume you are looking at Section 004 of COMM 290. Since `Dept` and `CrsNum` are included in the `Sections` table, and since a relationship line exists between the same two fields in the `Courses` table, Access can trace back along this line to the `Courses` table and find all the course-specific information. All other sections of COMM 290 point back

to the same record in the `Courses` table so the course information only needs to be stored once.

3.2 Learning objectives

- Why do I want to represent my information in multiple tables connected by relationships?
- How do I create relationships in Access?
- How do I edit or change relationships?
- What is referential integrity and why is it important?

3.3 Tutorial exercises

3.3.1 Creating relationships between tables

- Close the `Catalog View` table and return to the database window.
- Select *Tools > Relationships* from the main menu.

In version 2.0 the menu structure is slightly different. As such, you select *Edit > Relationships* instead.

- To add a table to the relationship window, select *Relationships > Show Table* from menu or press the show table icon ( ) on the tool bar.
- Perform the steps shown in Figure 3.3 to add the `Courses` and `Sections` tables.
- Specify the relationship between the **primary key** in `Courses` and the **foreign key** in `Sections`. This is shown in Figure 3.4.

Do not check cascading deletions or updates unless you are absolutely sure what they mean. See on-line help if you are curious.

FIGURE 3.3: Add the `Courses` and `Sections` tables to the relationship window.

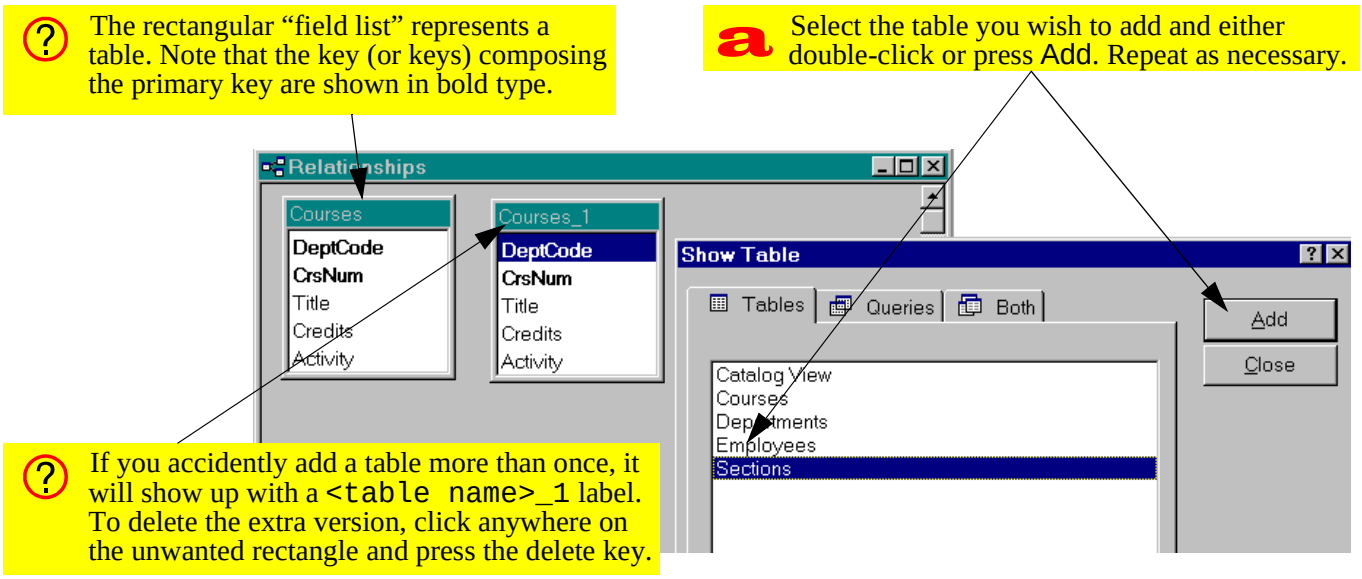
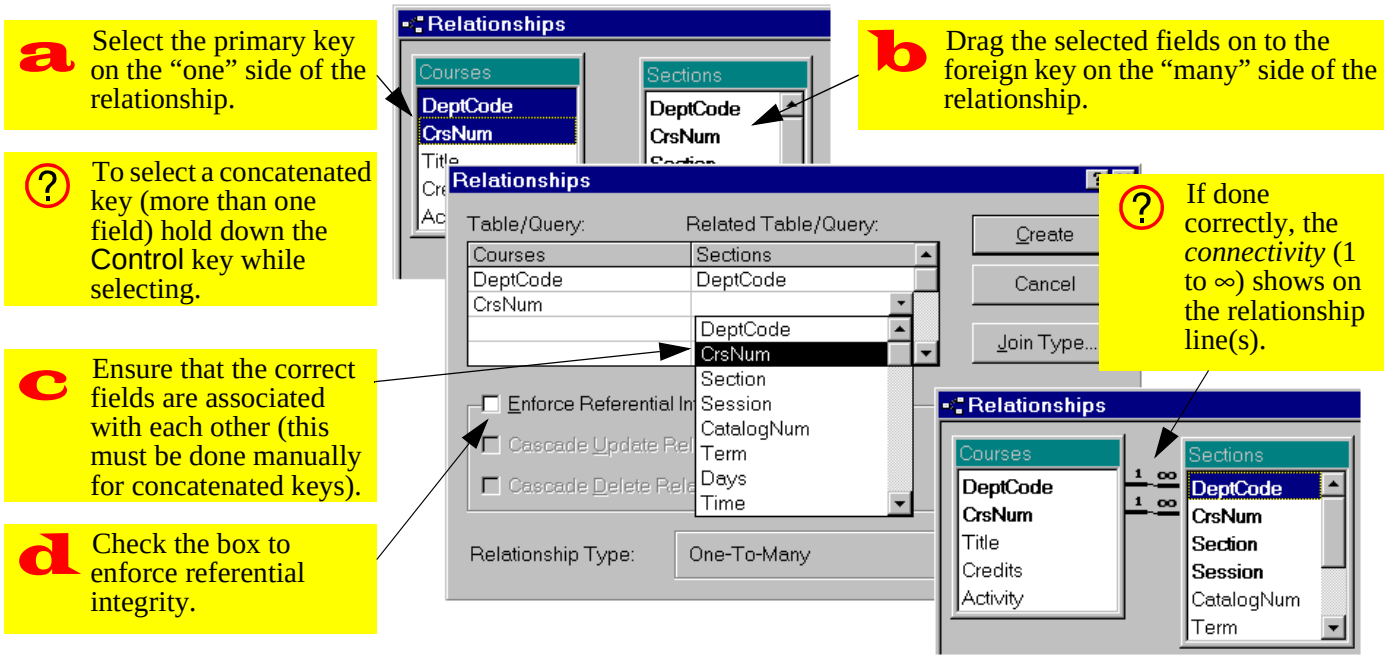


FIGURE 3.4: Create a relationship between the two tables.



3.3.2 Editing and deleting relationships

There are two common reasons for having to edit or delete a relationship:

1. You want to change the data type of one of the fields in the relationship — Access will not let you do this without first deleting the relationship (after you change the data type, you must re-create the relationship).
2. You forget to specify referential integrity — if the "1" and "∞" symbols do not appear on the relationship line, then you have not checked the box to enforce referential integrity.

In this section, assume that we have forgotten to enforce referential integrity between `Courses` and `Sections`.

- Perform the steps shown in Figure 3.5 to edit the relationship between `Courses` and `Sections`.

Note that simply deleting the table in the relationship window does not delete the relationship, it merely hides it from view.

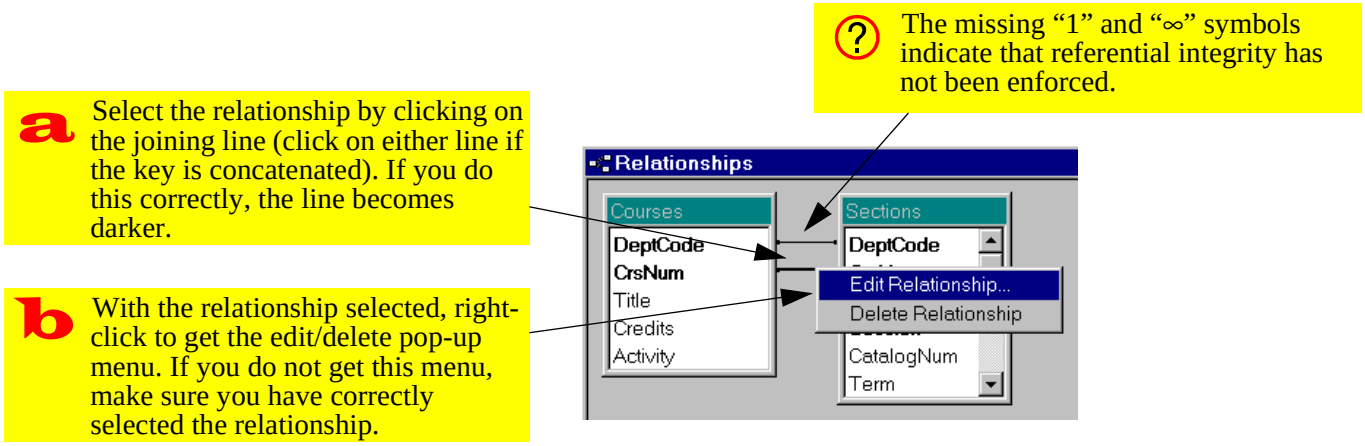
3.4 Discussion

3.4.1 One-to-many relationships

There are three types of relationships that occur in data modeling:

1. **one-to-one** — A one-to-one relationship exists between a student and a student number.
2. **one-to-many** — A one-to-many relationship exists between courses and sections: each course may consist of many sections, but each section is associated with exactly one course.
3. **many-to-many** — A many-to-many relationship exists between students and courses: each student can take many courses and each course can contain many students.

FIGURE 3.5: Edit an existing relationship.



Although the data modeling technique used most often in information system development—**Entity-Relationship diagraming**—permits the specification of many-to-many relationships, these relationships cannot be implemented in a relational database. As a consequence, many-to-many relationships are usually broken down into a series of one-to-many relationships via "composite entities" (alternatively, "bridging tables"). Thus to implement the student-takes-course relationship, three tables are used: `Students`, `Courses`, and `StudentTakesCourse`.

3.4.2 Referential integrity

One important feature of Access is that it allows you to enforce referential integrity at the relationship level. What is referential integrity? Essentially, referential integrity means that every record on the "many" side of a relationship has a corresponding record on the "one" side.

Enforcing referential integrity means that you cannot, for instance, create a new record in the `Sections` table without having a valid record in the `Courses` table. This is because having a section called "BSKW 101 Section 001" is meaningless unless there is a course called "BSKW 101". In addition, referential integrity prevents you from deleting records on the "one" side if related records exist on the "many" side. This eliminates the problem of "orphaned" records created when parent records are deleted.

Referential integrity is especially important in the context of transaction processing systems. Imagine that someone comes into your store, makes a large purchase, asks you to bill customer number "123", and leaves. What if your order entry system allows you to create an order for customer "123" without

first checking that such a customer exists? If you have no customer 123 record, where do you send the bill?

In systems that do not automatically enforce referential integrity, these checks have to be written in a programming language. This is just one example of how table-level features can save you enormous programming effort.

Enforcing referential integrity has obvious implications for data entry: You cannot populate the "many" side of the table until you populate the "one" side.

3.5 Application to the assignment

- Specify all relationships—including referential integrity constraints—between tables in your system. You are not responsible for cascading updates/deletions in this assignment.

A primary key and a foreign key must be of the same data type before a relationship can be created between them. Because of this, it is important to remember that the autonumber data type (or counter in version 2.0) is really a long integer.

It never makes sense to have a relationship between two autonumber fields. A foreign key cannot be an autonumber since referential integrity constraints require it to take on a an existing value from a parent table.

Access Tutorial 4: Basic Queries Using QBE

4.1 Introduction: Using queries to get the information you need

At first glance, it appears that splitting information into multiple tables and relationships creates more of a headache than it is worth. Many people like to have all the information they need on one screen (like a spreadsheet, for instance); they do not want to have to know about foreign keys and relationships and so on.

Queries address this problem. They allow the user to join data from one or more tables, order the data in different ways, calculate new fields, and specify criteria to filter out certain records.

The important thing is that *the query itself contains no data*—it merely reorganizes the data from the table (or tables) on which it is built without changing the "underlying tables" in any way.

Once a query is defined, it can be used in exactly the same way as a table. Because of this, it is useful to think of queries as "virtual tables". Similarly, in some DBMSes, queries are called "views" because they allow different users and different applications to have different views of the same data.

4.2 Learning objectives

- ❒ Do queries contain any data?
- ❒ How do I create a query?
- ❒ What can I do with a query?
- ❒ How do I create a calculated field?
- ❒ Why does Access add square brackets around field names?
- ❒ What names should I give the queries I create?
- ❒ What does the ampersand operator (&) do?
- ❒ What is a non-updatable recordset? How do I tell whether a query results in a non-updatable recordset?

Last update: 25-Aug-1997

4.3 Tutorial exercises

4.3.1 Creating a query

- Use the *New* button in the *Queries* pane of the database window to create a new query as shown in Figure 4.1.
- Add the `Courses` table to the query as shown in Figure 4.2.
- Examine the basic elements of the query design screen as shown in Figure 4.3.
- Save your query (*Control-S*) using the name `qryCourses`.

4.3.2 Five basic query operations

4.3.2.1 Projection

Projecting a field into a query simply means including it in the query definition. The ability to base a query on a subset of the fields in an underlying table (or tables) is particularly useful when dealing with tables that contain some information that is confidential and some that is not confidential. For instance, the `Employees` table you created in Tutorial 2 contains a field called `Salary`. However, most of the queries seen by end-users would not include this information, thereby keeping it private.

- Perform the steps shown in Figure 4.4 to project the `DeptCode`, `CrsNum`, and `Title` fields into the query definition.
- Select *View > Datasheet* from the menu to see the results of the query. Alternatively, press the datasheet icon ( ) on the tool bar.

FIGURE 4.1: Create a new query.

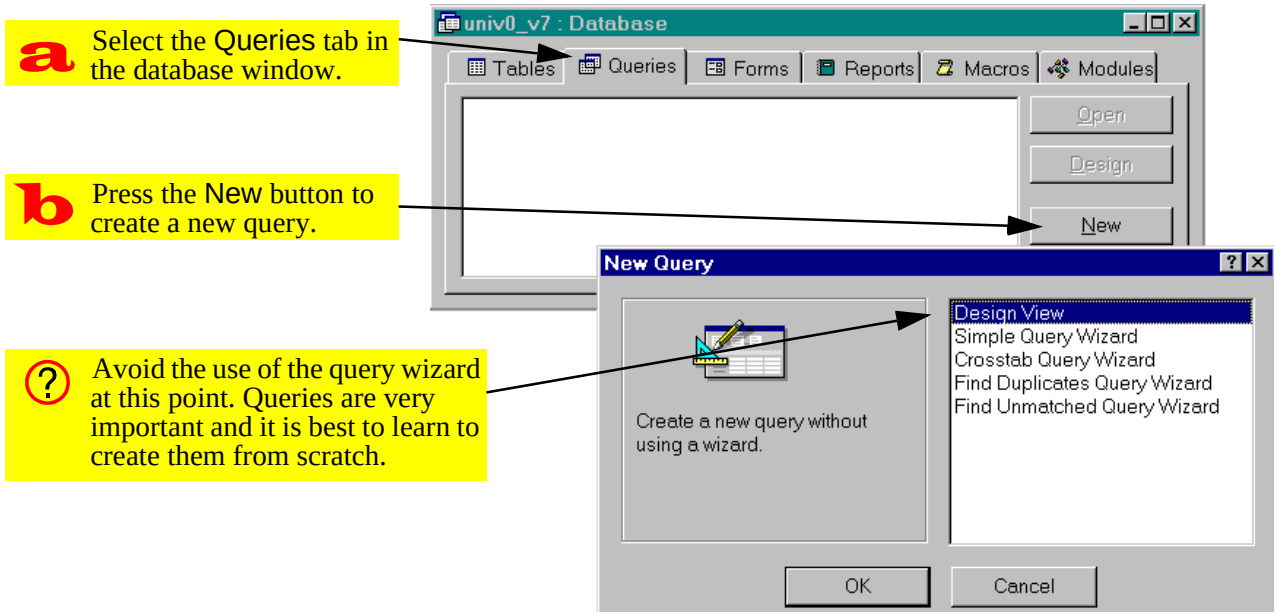


FIGURE 4.2: Add tables to your query using the "show table" window.

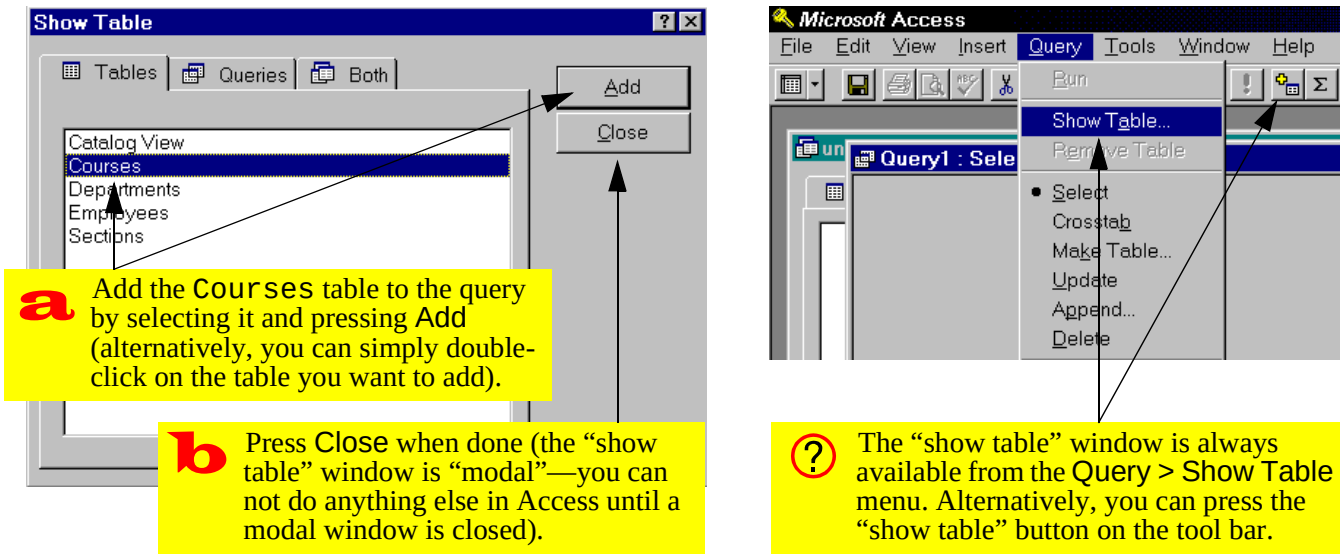


FIGURE 4.3: The basic elements of the query design screen.

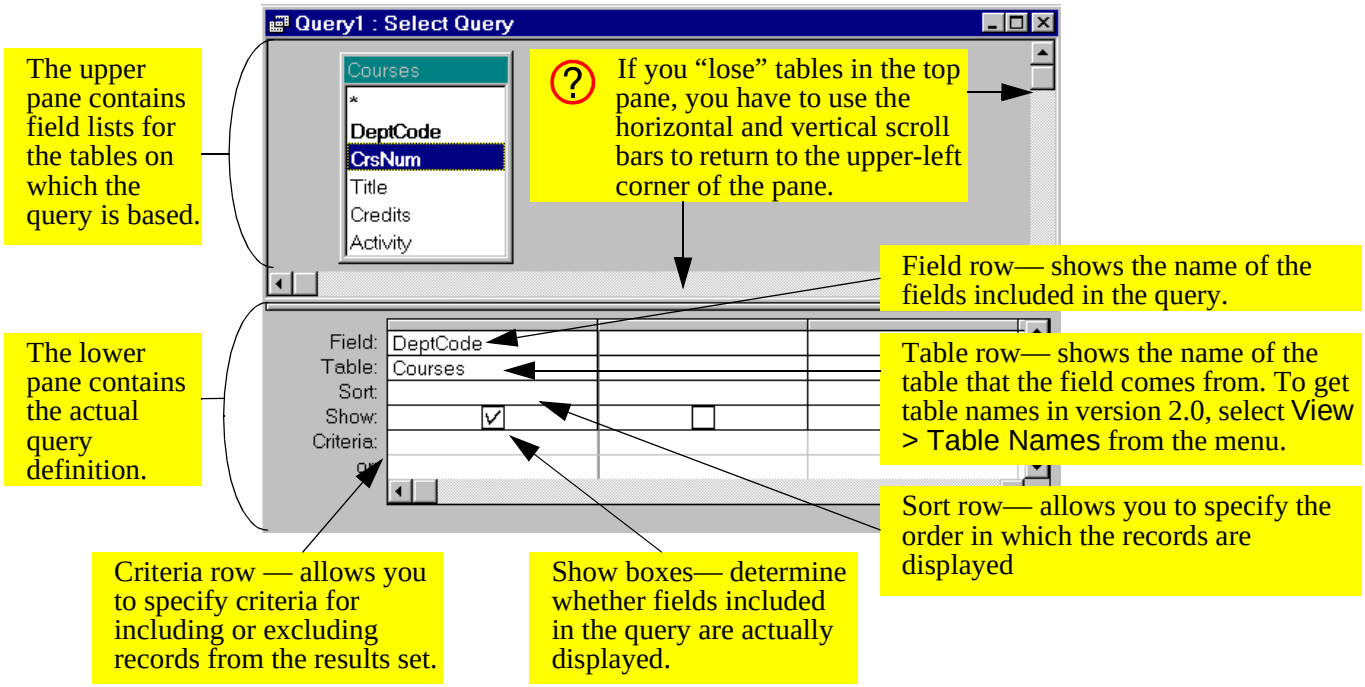
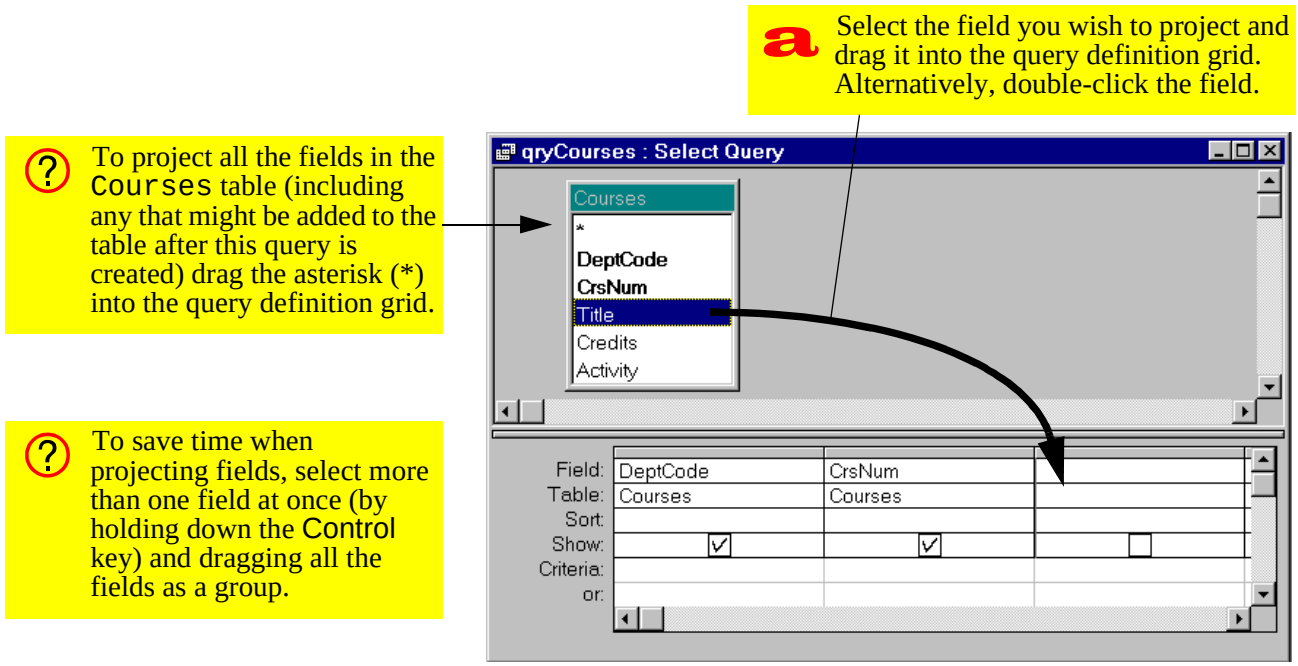


FIGURE 4.4: Project a subset of the available fields into the query definition.



- Select *View > Query Design* to return to design mode. Alternatively, press the design icon ( ) on the tool bar.

4.3.2.2 Sorting

When you use a query to sort, you do not change the physical order of the records in the underlying table (that is, you do not sort the table). As a result, different queries based on the same table can display the records in different orders.

- Perform the steps shown in Figure 4.5 to sort the results of `qryCourses` by `DeptCode` and `CrsNum`.

Since a query is never used to display data to a user, you can move the fields around within the query definition to get the desired sorting precedence. You then reorder the fields in the form or report for presentation to the user.

4.3.2.3 Selection

You select records by specifying conditions that each record must satisfy in order to be included in the results set. In "query-by-example" you enter examples of the results you desire into the criteria row.

- Perform the steps shown in Figure 4.6 to select only those courses with a `DeptCode = "COMM"`.

4.3.2.4 Complex selection criteria

It is also possible to create complex selection criteria using Boolean constructs such as AND, OR, and NOT.

- Project the `Credits` field into the query.
- Perform the steps shown in Figure 4.7 to create a query giving the following result:
  "Show the department, course number, and title of all courses in the Commerce department for which the number of credits is greater than three."

FIGURE 4.5: Sorting the results set on one or more fields.

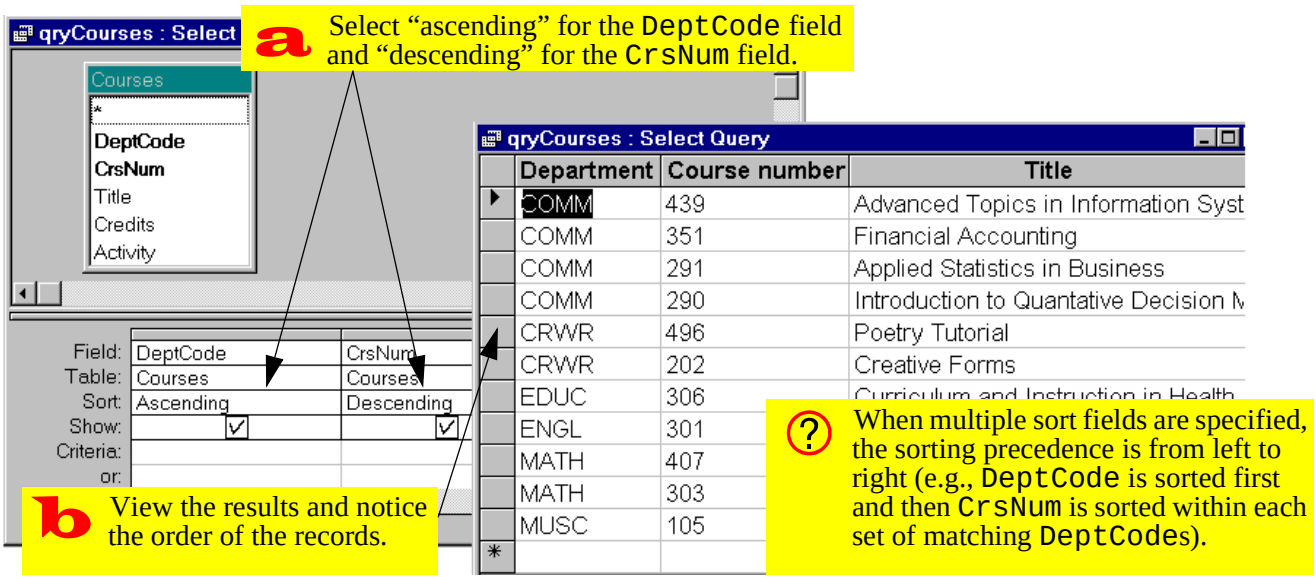


FIGURE 4.6: Select a subset of records from the `Courses` table matching a specific criterion.

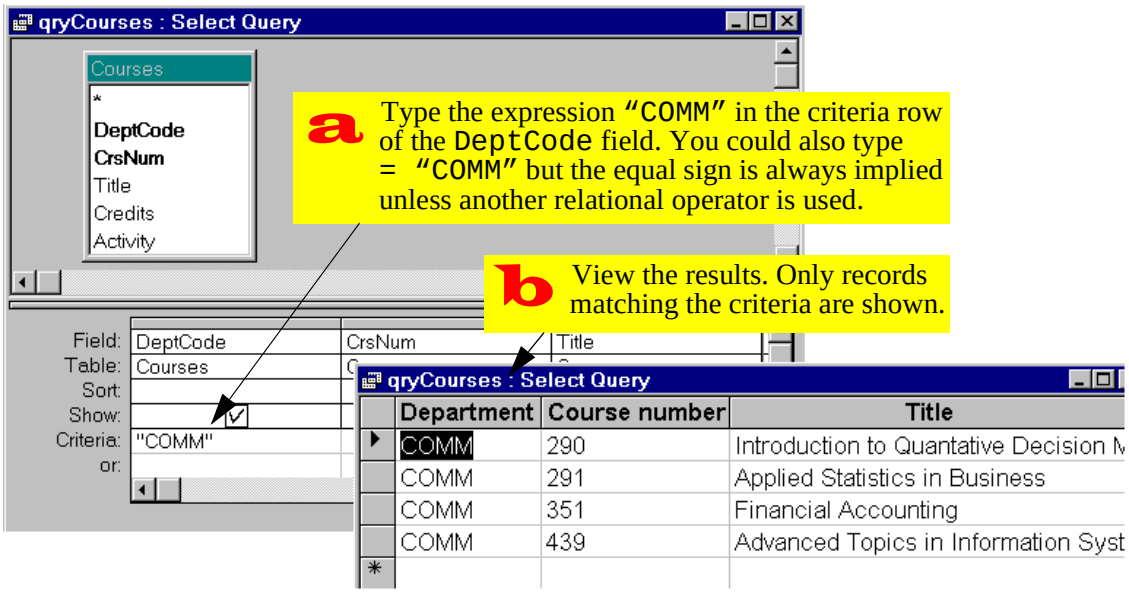
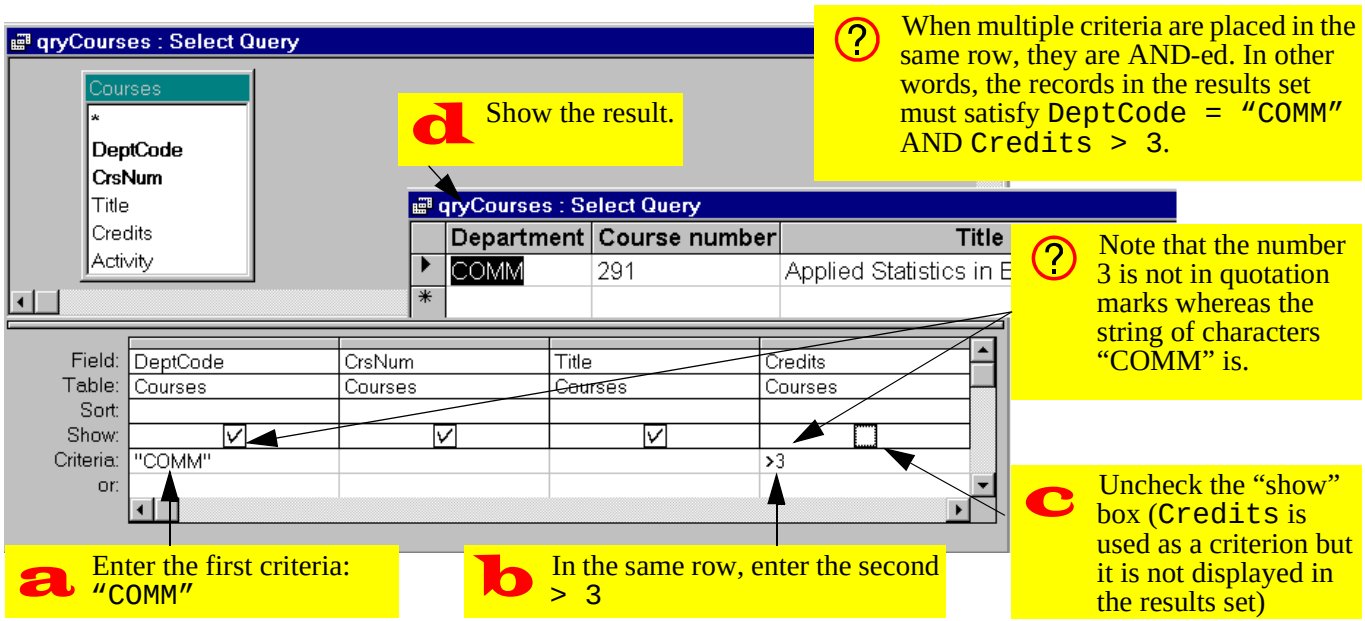


FIGURE 4.7: Select records using an AND condition.



- Perform the steps shown in Figure 4.8 to create a query giving the following result:
  "Show the department, course number, and title of *all* courses from the Commerce department and also show those from the Creative Writing department for which the number of credits is greater than three."

4.3.2.5 Joining

In Tutorial 3, you were advised to break you information down into multiple tables with relationships between them. In order to put this information back together in a usable form, you use a join query.

- Close `qryCourses`.
- Open the relationships window and ensure you have a relationship defined between `Courses` and `Sections`. If you do not, create one now (do not forget to enforce referential integrity).
- Create a new query called `qryCatalogNum` based on the `Courses` and `Sections` tables.
- Project `Title` from the `Courses` table and `DeptCode`, `CrsNum`, `Section` and `CatalogNum` from the `Sections` table (see Figure 4.9).
- Follow the instructions in Figure 4.10 to move `CatalogNum` to the far left of the query definition grid.

Access performs an automatic lookup of information from the "one" side of the relationship whenever the a valid value is entered into the foreign key of the "many" side of the relationship. To see how this works, create a new section of "MUSC 105":

- Scroll to the bottom of the query in datasheet mode and click on the department field.
- Enter "MUSC".
- Enter "105" in the course number field.

Once Access knows the `DeptCode` and `CrsNum` of a section, it can uniquely identify the course that the section belongs to (which means it also knows the values of `Title`, `Credits`, `Activity`, etc.)

FIGURE 4.8: Select records using an AND and an OR condition.

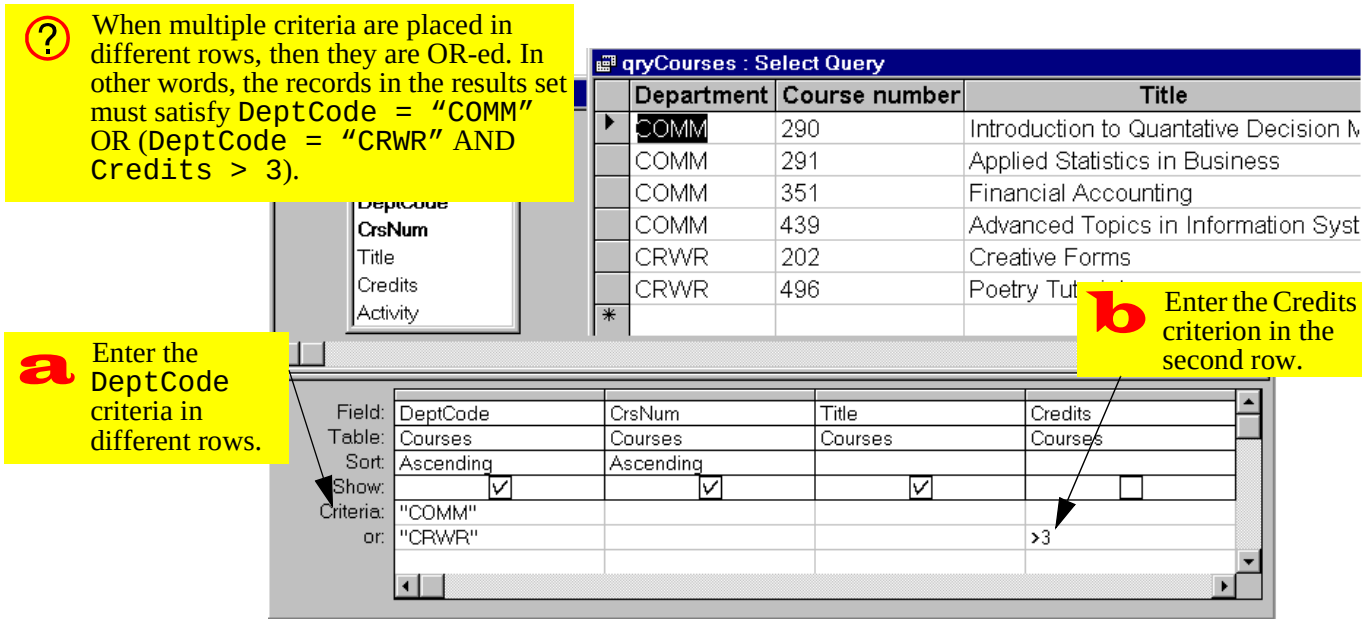


FIGURE 4.9: Create a query that joins `Courses` and `Sections`.

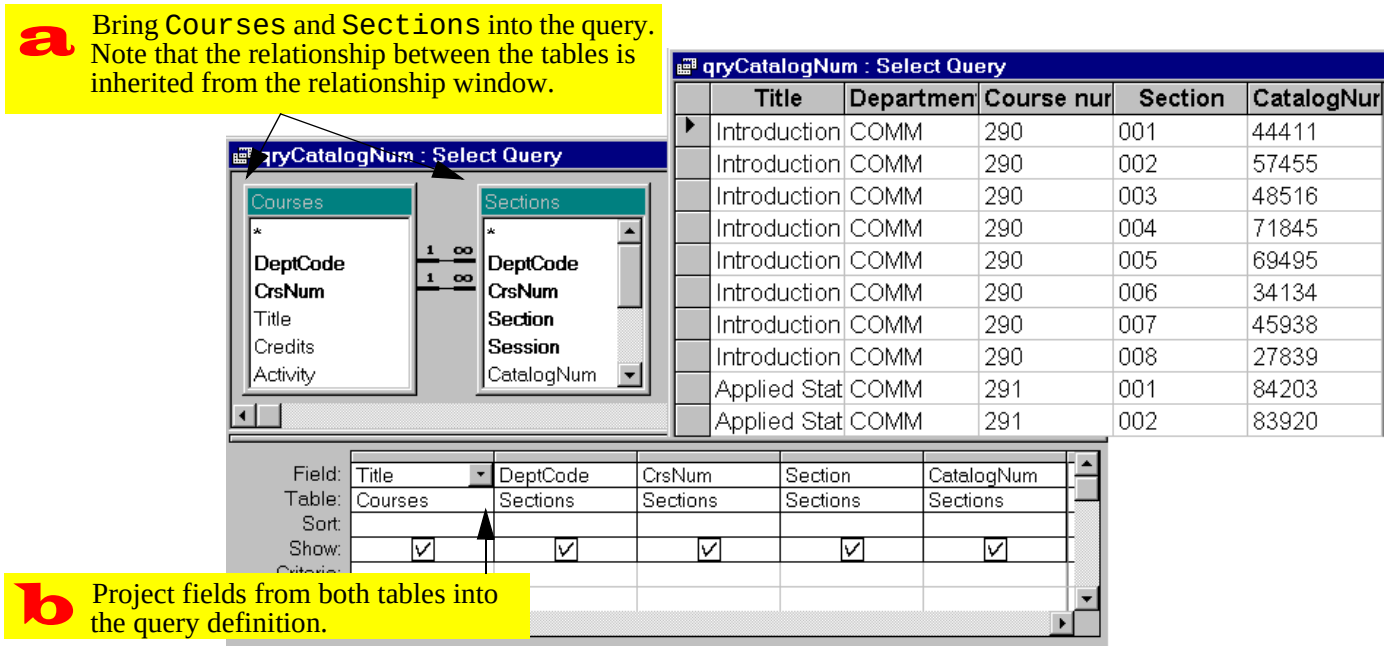
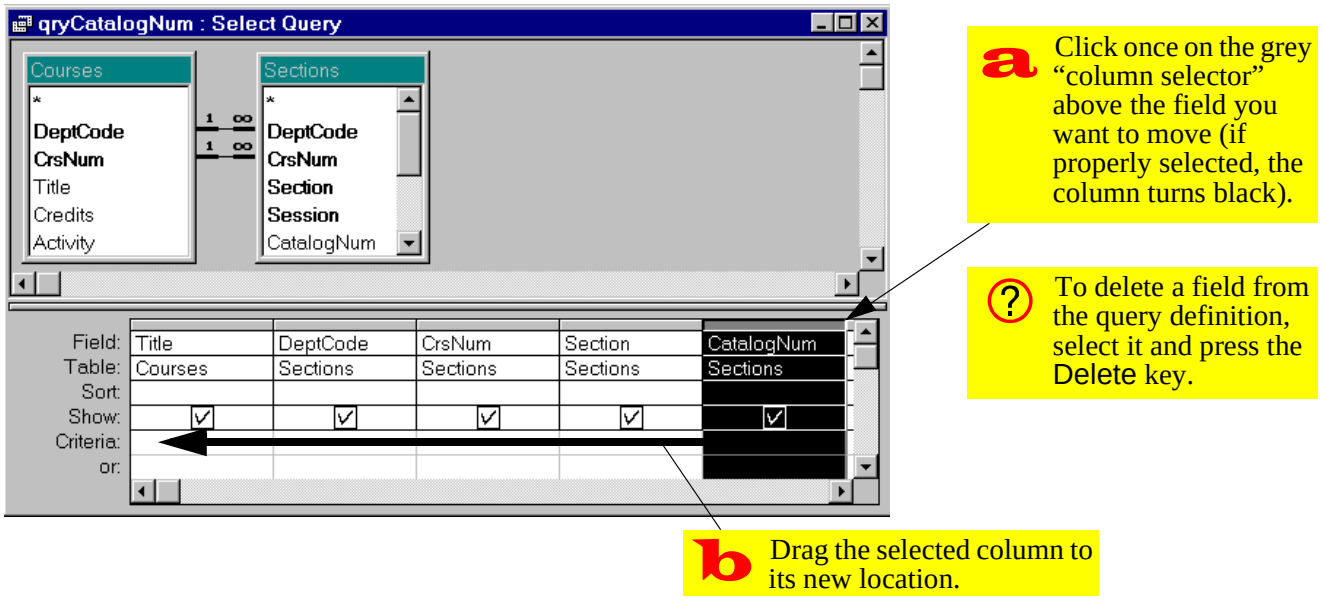


FIGURE 4.10: Move a field within the query definition grid.



4.3.3 Creating calculated fields

A calculated field is a "virtual field" in a query for which the value is a function of one or more fields in the underlying table. To illustrate this, we will create two calculated fields:

1. one to combine `DeptCode` and `CrsNum` into one field,
2. one to translate the `Credits` field into a dichotomous string variable (`full year` or `half year`).

The syntax of a calculated field is always the same:
`<calc field name>: <definition>`

For example, the syntax for the calculated field called `Course` is:
`Course: DeptCode & CrsNum`

The calculated field name can be just about anything, as long as it is unique. The definition is any expression that Access can evaluate. In this case, the expression involves two fields from the `Courses` table (`DeptCode` and `CrsNum`) and the ampersand operator (see Section 4.4.2 for more information on using the ampersand operator).

- Create a new query called `qryCourseLengths` based on the `Courses` table.
- Follow the instructions in Figure 4.11 to create the calculated field `Course`
- Run the query to verify the results, as shown in Figure 4.12.

When you use field names in expressions, Access normally adds square brackets. This is not cause for concern because in Access, square brackets simply indicate the name of a field (or some other object in the Access environment). However, if your field name contains blank spaces (e.g., `Dept Code`), the square brackets are NOT optional—you must

FIGURE 4.11: Create a calculated field based on two other fields.

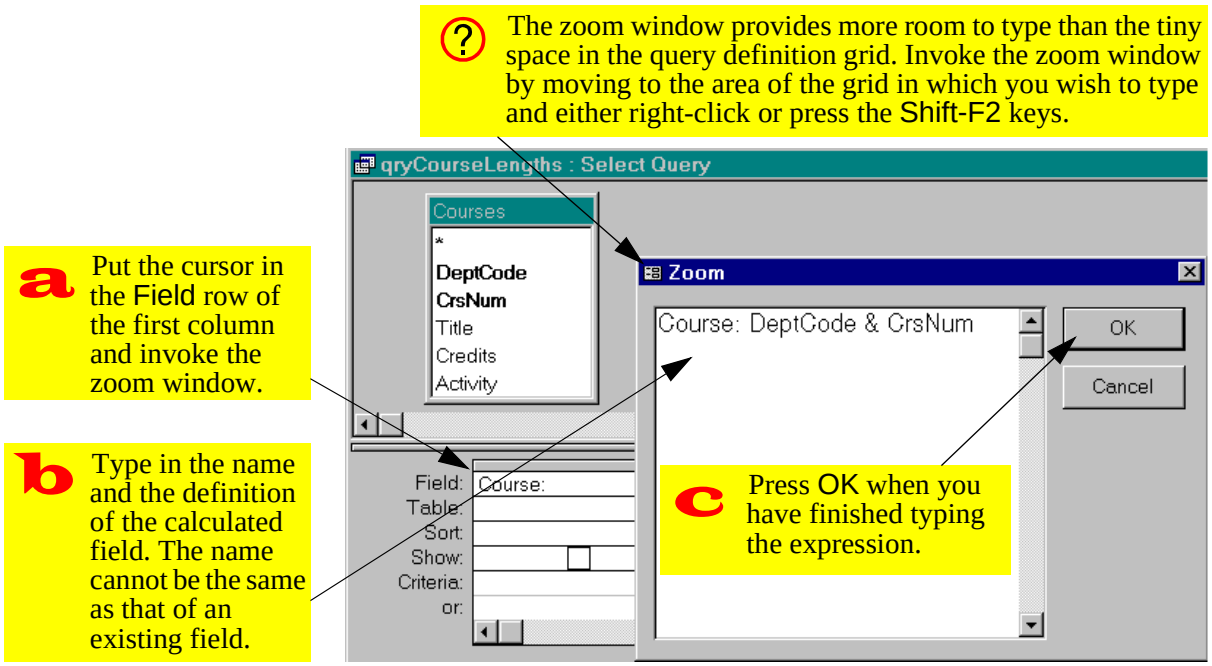
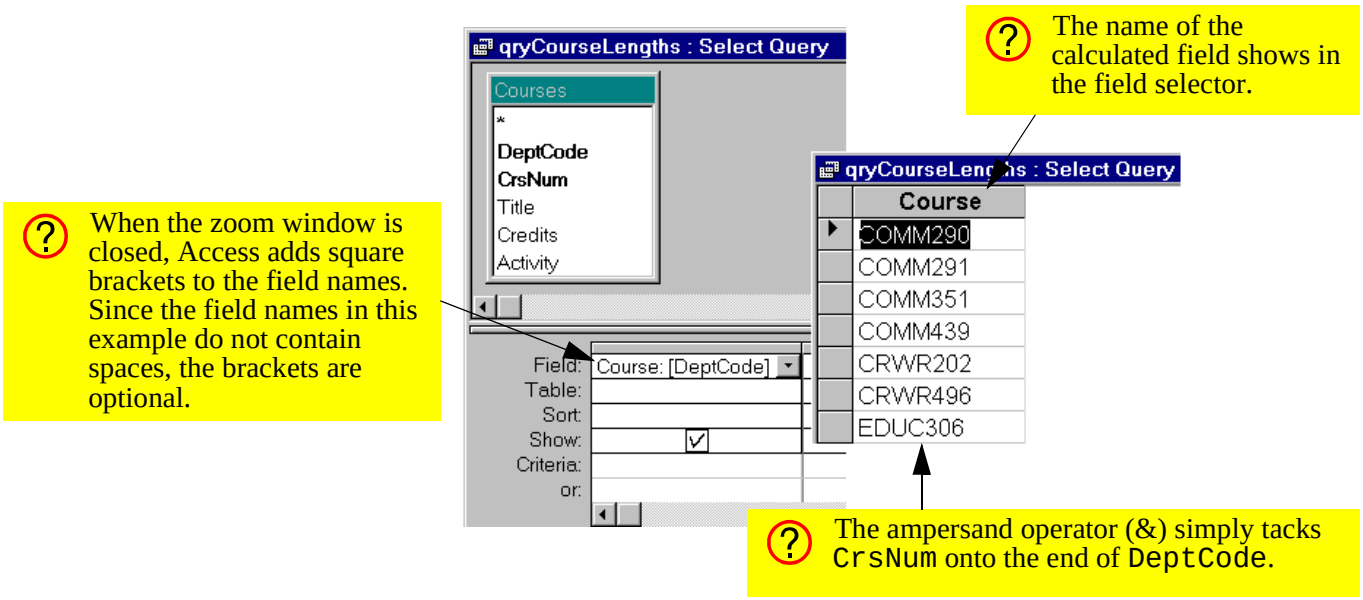


FIGURE 4.12: The resulting calculated field.



type them every time you use the field name in an expression.

4.3.3.1 Refining the calculated field

Instead of having `DeptCode` and `CrsNum` run together in the new `Course` field, you may prefer to have a space separating the two parts.

- Edit the `Courses` field by clicking on the field row and invoking the zoom box.
- Add a space (in quotation marks) between the two constituent fields:
  `Course: DeptCode & " " & CrsNum`
- Switch to datasheet mode to see the result.

4.3.3.2 A more complex calculated field

To create a calculated field that maps `Credits` to a dichotomous string variable, we need a means of testing whether the value of `Credits` exceeds a certain threshold (e.g., any course with more than three credits is a full-year course). To do this, we will use the "immediate if" (`iif`) function.

- Search on-line help for information about the `iif()` function.

Basically, the function uses the following syntax:

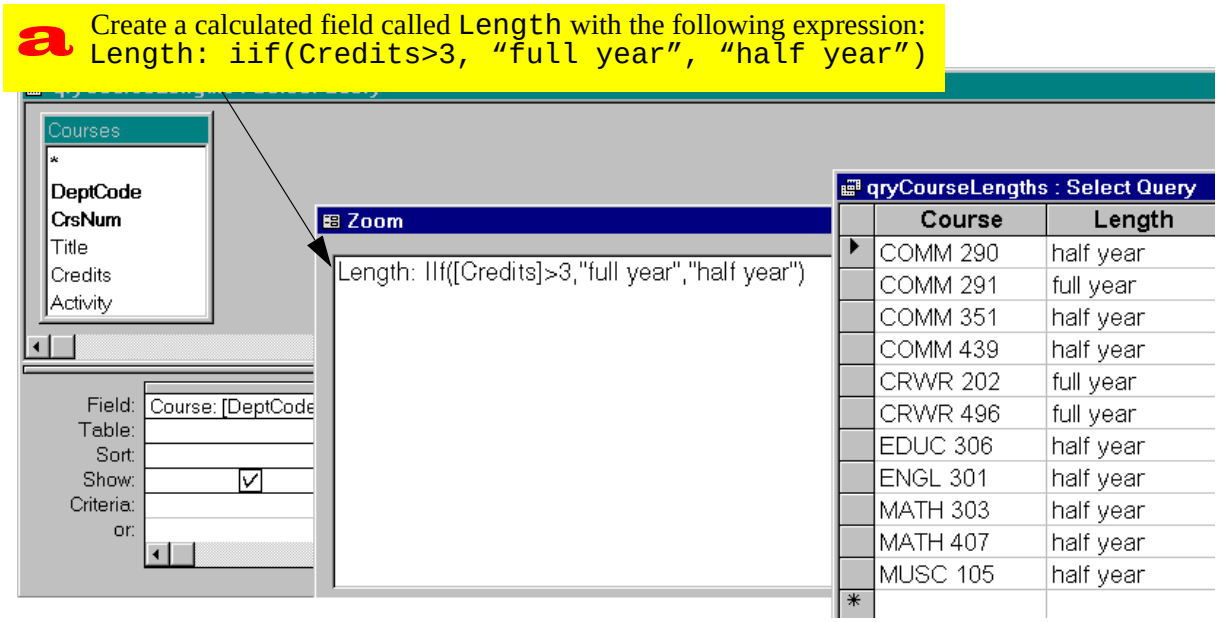
```
iif(<expression>, <true part>,
 <false part>)
```

to implement the following logic:

```
IF <expression> = TRUE THEN
   RETURN <true part>
ELSE
   RETURN <false part>
END IF
```

- Create a new calculated field called `Length`:
  `Length: iif(Credits > 3, "full year", "half year")`
- Verify the results, as shown in Figure 4.13.

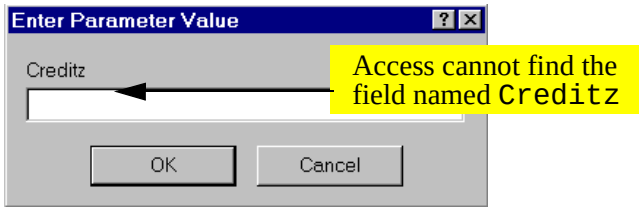
FIGURE 4.13: Create a calculated field using the “immediate if” function



4.3.4 Errors in queries

It may be that after defining a calculated field, you get the “enter parameter” dialog box shown in Figure 4.14 when you run the query. This occurs when you spell a field name incorrectly. Access cannot resolve the name of the misspelled field and thus asks the user for the value. To eliminate the problem, simply correct the spelling mistake.

FIGURE 4.14: A spelling error in a calculated field.



4.4 Discussion

4.4.1 Naming conventions for database objects

There are relatively few naming restrictions for database objects in Access. However, a clear, consistent method for choosing names can save time and avoid confusion later on. Although there is no hard and fast naming convention required for the assignment, the following points should be kept in mind:

- Use meaningful names — An object named `Table1` does not tell you much about the contents of the table. Furthermore, since there is no practical limit to the length of the names, you should not use short, cryptic names such as `s96w_b`. As the number of objects in your database grows, the time spent carefully naming your objects will pay itself back many times.

- Use capitalization rather than spaces to separate words — Unlike many database systems, Access allows spaces in object names. However, if you choose to use spaces, you will have to enclose your field names in square brackets whenever you use them in expressions (e.g., `[Back Orders]`). As such, it is slightly more efficient to use a name such as `BackOrders` than `Back Orders`.
- Give each type of object a distinctive prefix (or suffix) — This is especially important in the context of queries since tables and queries cannot have the same name. For example, you cannot have a table named `BackOrders` and a query named `BackOrders`. However, if all your query names are of the form `qryBackOrders`, then distinguishing between tables and queries is straightforward.
- Stick to standard alphanumeric characters — You should limit yourself to the characters [A...Z], [a...z], [0...9], and perhaps underscore (_) and dash (-). Although Access allows you to use virtually any character, undocumented problems have been encountered in the past with non-alphanumeric characters such as the pound sign (#).

Table 4.1 shows a suggested naming convention for Access database objects (you will discover what these objects are in the course of doing the tutorials).

4.4.2 The ampersand (&) operator

The ampersand operator is like any other operator (e.g., +, -, ×, ÷) except that it is intended for use on strings of characters. What the ampersand does is simply add one string on to the end of another string (hence its other name: the "concatenation" operator). For example, the expression

```
"First string" & "Second string"
```


Table 4.1: A suggested naming convention for Access database objects.

Object type	Prefix	Example
table	(none)	`OrderDetails`
query	qry	`qryNonZeroBackOrders`
parameter query	pqry	`pqryItemsInOrder`
form	frm	`frmOrders`
sub form	sfrm	`sfrmOrderDetails`
switchboard form	swb	`swbMainSwitchboard`
report	rpt	`rptInvoice`
sub report	srpt	`srptInvoiceDetails`
macro	mcr	`mcrOrders`
Visual Basic module	bas	`basUtilities`

yields the result

```
First stringSecond string
```

However, if a space is include within the quotation marks of the second string (`" Second string"`), the result is:

```
First string Second string
```

4.4.3 Using queries to populate tables on the "many" side of a relationship

In Section 4.3.2.5, you added a record to the `Sections` table to demonstrate the automatic lookup feature of Access. However, a common mistake when creating queries for entering data into tables on the "many" side of a relationship is to forget to project the table's foreign key. That is, faced with two tables containing the fields `DeptCode` and `CrsNum`, you project the fields from the wrong table (the "one" side) into your query definition.

To illustrate the problem, do the following:

- Open the `qryCatalogNum` query and make the changes shown in Figure 4.15.
- Attempt to save the new section of “MUSC 105” as shown in Figure 4.16.

There are two ways to avoid this error when deciding which fields to project into your join queries:

1. Always show the table names when creating a query based on more than one table. That way, you can quickly determine whether the query makes sense.
2. Always ask yourself: “What is the purpose of this query?” If the answer is: “To add new records to the `Sections` table,” you automatically have to include *all* the fields from the `Sections` table. Fields from the `Courses` table are only shown for validation purposes.

4.4.4 Non-updatable recordsets

Another problem that sometimes occurs when creating join queries is that the query is not quite right in some way. In such cases, Access will allow you to view the results of the query, but it will not allow you to edit the data.

In this section, will look at a nonsensical query that results from an incompletely specified relationship. As you will probably discover, however, there are many different way to generate nonsensical queries.

- Create a new query called `qryNonUpdate` based on the `Courses` and `Sections` tables.
- Delete the `CrsNum` relationship but leave the `DeptCode` relationship intact, as shown in Figure 4.17.

The result of this query is that every section in a Commerce course will be associated with every Commerce course. Since allowing the user to update

FIGURE 4.15: Create a data-entry query without a foreign key.

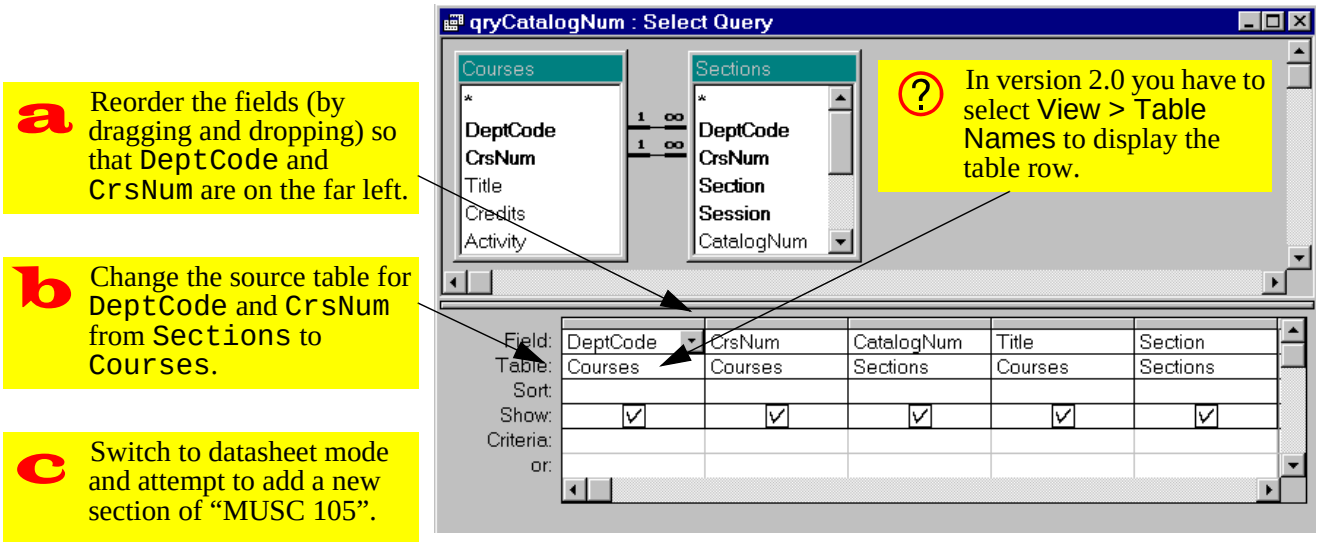


FIGURE 4.16: The result of attempting to save a record in which the foreign key is missing

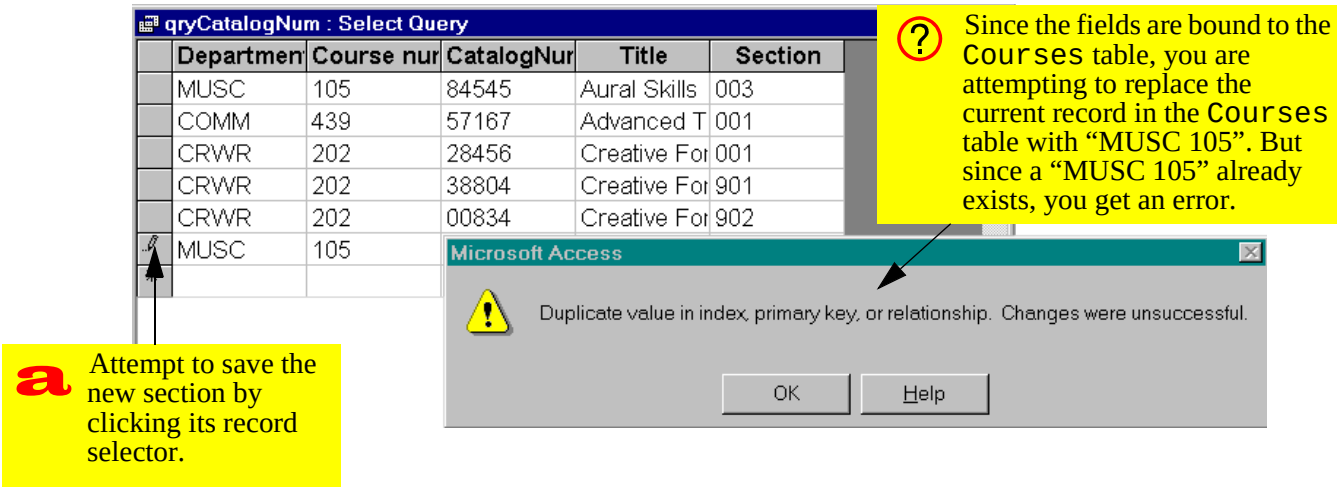
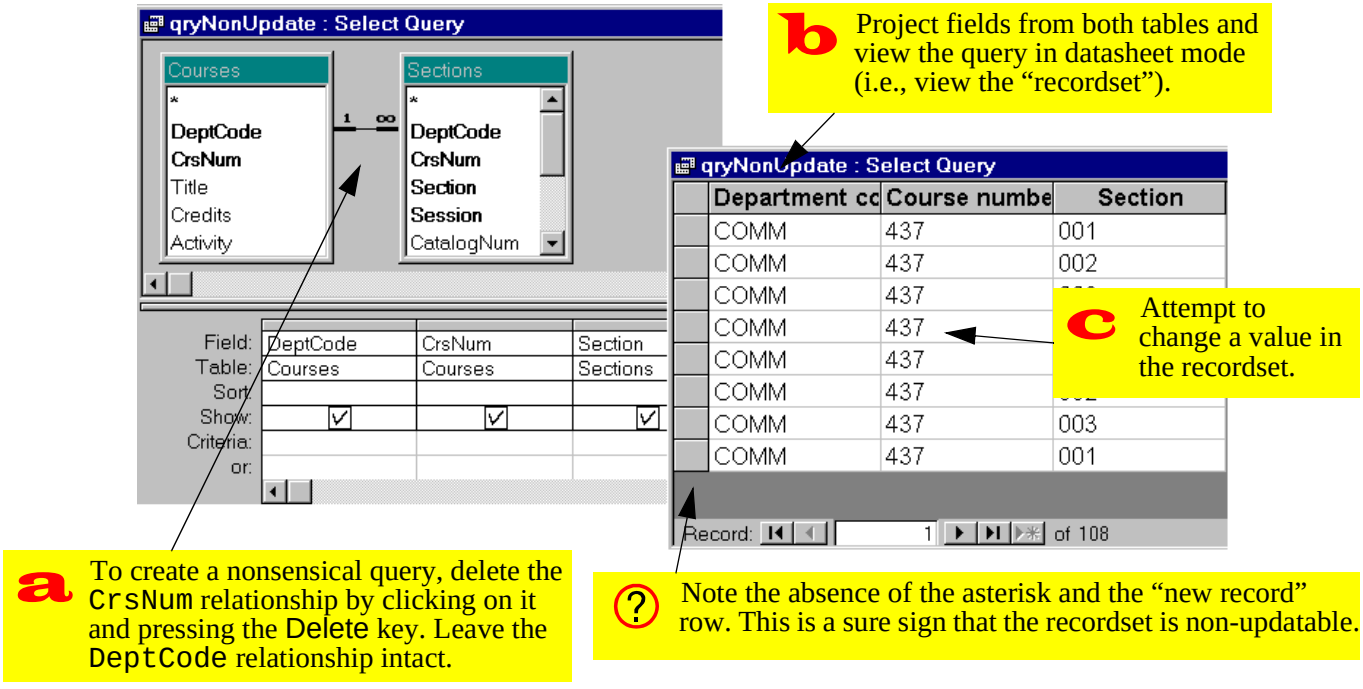


FIGURE 4.17: Create a non-updatable recordset.



the values in this recordset would create anomalies, Access designates the recordset as non-updatable.

A common mistake is to build data entry forms on nonsensical queries and to assume that there is a mistake in the form when the forms do not work. Clearly, if a query is non-updatable, a form based on the query is also going to be non-updatable. A quick check for a "new record" row in the query can save time and frustration.

4.5 Application to the assignment

- Create a query to sort the `Products` table by `ProductID`.
- Create a query that joins the `OrderDetails` and `Products` tables. When you enter a valid `ProductID`, the information about the product (such as name, quantity on hand, and so on) should appear automatically. If they do not, see Section 4.4.3.
- Create a calculated field in your `qryOrderDetails` query that calculates the extended price (quantity shipped × price) of each order detail.
- Enter the first order into your system by entering the information directly into tables or queries. This involves creating a single `Orders` record and several `OrderDetails` records. You must also consult the `Products` and `BackOrders` tables to determine the quantity of each item to ship.

Entering orders into your system will be much less work once the input forms and triggers are in place. The goal at this point is to get you thinking about the order entry process and ways in which it can be automated.

Access Tutorial 5: Basic Queries using SQL

5.1 Introduction: The difference between QBE and SQL

Query-By-Example (QBE) and Structured Query Language (SQL) are both well-known, industry-standard languages for extracting information from relational database systems. The advantage of QBE (as you saw in Tutorial 4) that it is graphical and relatively easy to use. The advantage of SQL is that it has achieved nearly universal adoption within the relational database world.

With only a few exceptions (which you probably will not encounter in this assignment) QBE and SQL are completely interchangeable. If you understand the underlying concepts (projection, selection, sorting, joining, and calculated fields) of one, you understand the underlying concepts of both. In fact, in Access you can switch between QBE and SQL versions of your queries with the click of a mouse.

Last update: 22-Aug-1997

Although you normally use QBE in Access, the ubiquity of SQL in organizations necessitates a brief overview.

5.2 Learning objectives

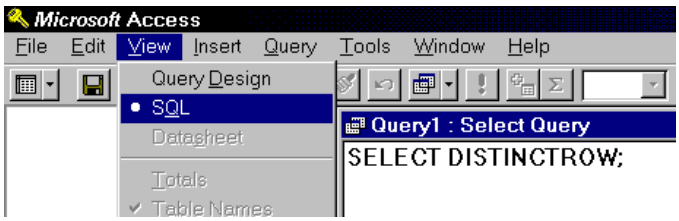
- ❒ What is the difference between QBE and SQL?
- ❒ How do I create an SQL query?

5.3 Tutorial exercises

In this section, you will create a few simple queries in SQL.

- Create a new query but close the “show table” dialog box with out adding tables.
- Select *View > SQL* to switch to the SQL editor as shown in Figure 5.1.

FIGURE 5.1: Open a query in SQL mode



5.3.1 Basic SQL queries

A typical SQL statement resembles the following:

```
SELECT DeptCode, CrsNum, Title FROM
Courses WHERE DeptCode = “COMM”;
```

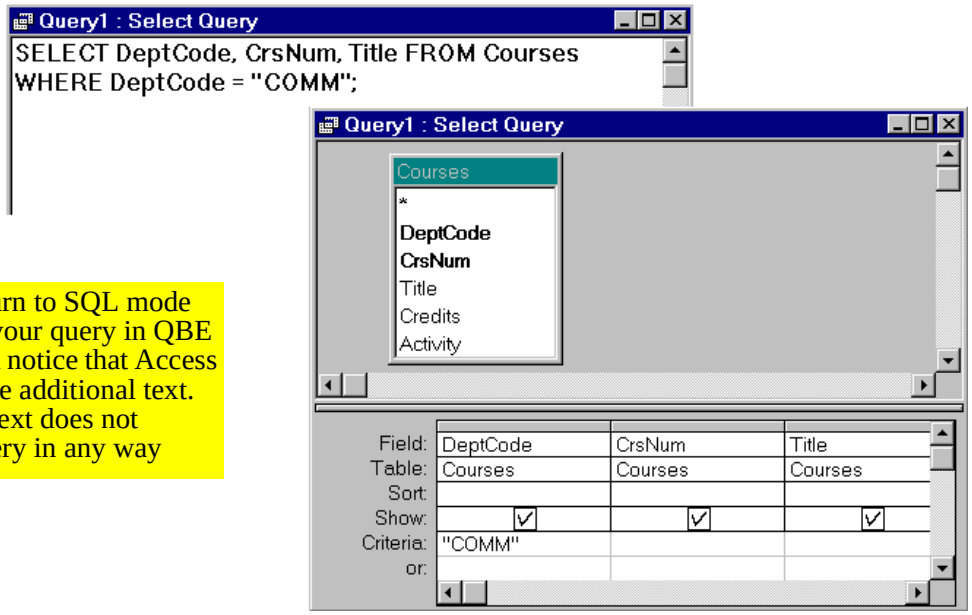
There are four parts to this statement:

1. `SELECT <field`$_1$`, field`$_2$`, …, field`$_n$`>` … — specifies which fields to project (the `DISTINCTROW` predicate shown in Figure 5.1 is optional and will not be discussed in this tutorial);
2. … `FROM <table>` … — specifies the underlying table (or tables) for the query;
3. … `WHERE <condition`$_1$ `AND/OR condition`$_2$`, …, AND/OR condition`$_n$`>` — specifies one or more conditions that each record must satisfy in order to be included in the results set;
4. `;` (semicolon) — all SQL statements must end with a semicolon (but if you forget it, Access will add it for you).

These can now be put together to build an SQL query:

- Type the following into the SQL window:
  ```
  SELECT DeptCode, CrsNum, Title FROM
  Courses WHERE DeptCode = “COMM”;
  ```
- Select *View > Datasheet* to view the results.
- Select *View > Query Design* to view the query in QBE mode, as shown in Figure 5.2.
- Save your query as `qryCoursesSQL`.

FIGURE 5.2: The SQL and QBE views are interchangeable.



When you return to SQL mode after viewing your query in QBE mode, you will notice that Access has added some additional text. This optional text does not change the query in any way

5.3.2 Complex WHERE clauses

You can use AND, OR, and NOT conditions in your WHERE clauses in a straightforward manner.

- Change your query to the following to get all Commerce courses with more than three credits:

```
SELECT DeptCode, CrsNum, Title
FROM Courses
WHERE DeptCode = “COMM” AND Credits
  > 3
```

Note that since `DeptCode` is a text field, its criterion must be a string (in this case, the literal string “COMM”). However, `Credits` is a numeric field and its criterion must be a number (thus, there cannot be quotation marks around the 3).

5.3.3 Join queries

Join queries use the same elements as a basic select query. The only difference is that the FROM statement is replaced with a statement that describes the tables to be joined and the relationship (i.e., foreign key) between them:

```
... FROM table1 INNER JOIN table2 ON
  table1.field1 = table2.field2 ...
```

Note that since both tables contain the fields `DeptCode` and `CrsNum`, the `<table name>.<field name>` notation must be used to remove any ambiguity.

- Create a new SQL query containing the text:

```
SELECT Courses.DeptCode,
  Courses.CrsNum, Courses.Title,
  Sections.CatalogNum
FROM Courses INNER JOIN Sections ON
  Courses.CrsNum = Sections.CrsNum
```

```
    AND Courses.DeptCode =
    Sections.DeptCode
  WHERE Courses.DeptCode=”COMM”;
```

5.4 Discussion

Although the syntax of SQL is not particularly difficult, writing long SQL queries is tedious and error-prone. For this reason, you are advised to use QBE for the assignment.

In the real world, however, when you say you know something about databases, it usually implies you know the “data definition” and “data manipulation” aspects of SQL in your sleep. If you plan to pursue a career in information systems, a comprehensive SQL reference book can be a worthwhile investment.

Access Tutorial 6: Form Fundamentals

6.1 Introduction: Using forms as the core of an application

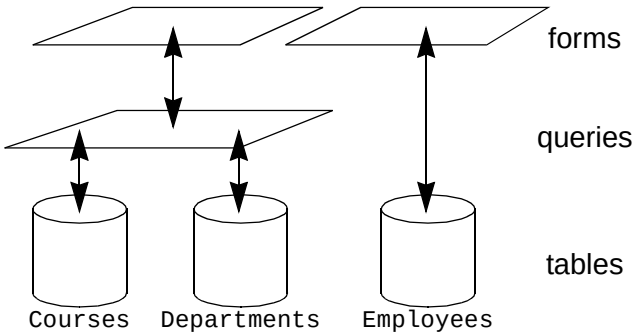
Forms provide a user-oriented interface to the data in a database application. They allow you, as a developer, to specify in detail the appearance and behavior of the data on screen and to exert a certain amount of control over the user's additions and modifications to the data.

Like queries, forms do not contain any data. Instead, they provide a "window" through which tables and queries can be viewed. The relationship between tables, queries, and forms is shown in Figure 6.1.

In this tutorial, we are going to explore the basic elements of form creation using Access' form design tools. In subsequent tutorials, we will extend the functionality and ease-of-use of our basic forms with subforms (Tutorial 7), "combo box" controls (Tutorial 8), and triggers (Tutorial 13).

Last update: 24-Aug-1997

FIGURE 6.1: The relationship between forms, queries, and tables.



6.2 Learning objectives

- ❒ Do forms contain data?
- ❒ How do I create a form?

- ❒ How do I make the contents of a field on a form read-only?
- ❒ What is an unbound text box? How do I create one?
- ❒ How do I create a form using the form wizard?
- ❒ What is the difference between a columnar (single-column) and tabular form?

6.3 Tutorial exercises

6.3.1 Creating a form from scratch

Although Access provides an excellent wizard for creating simple forms, you will start by building a form from scratch. This will give you a better appreciation of what it is the wizard does and provide you with the basic knowledge needed to customize and refine the wizard's output.

- Create a new blank form based on the `Courses` table, as shown in Figure 6.2.
- The basic elements of the design screen are shown in Figure 6.3. Use the *View* menu to display the **toolbox** and **field list** if they are not already visible.

6.3.1.1 Adding bound text boxes

- Add a "bound" text box for the `DeptCode` field by dragging `DeptCode` from the field list to the form background, as shown in Figure 6.4.
- Reposition the `DeptCode` text box in the upper left of the form.

Remember that you can always use the "undo" feature to reverse mistakes. Select *Edit > Undo* from the menu or simply press *Control-Z* (this works the same in virtually all Windows applications).

FIGURE 6.2: Create a new form to display data from the `Courses` table.

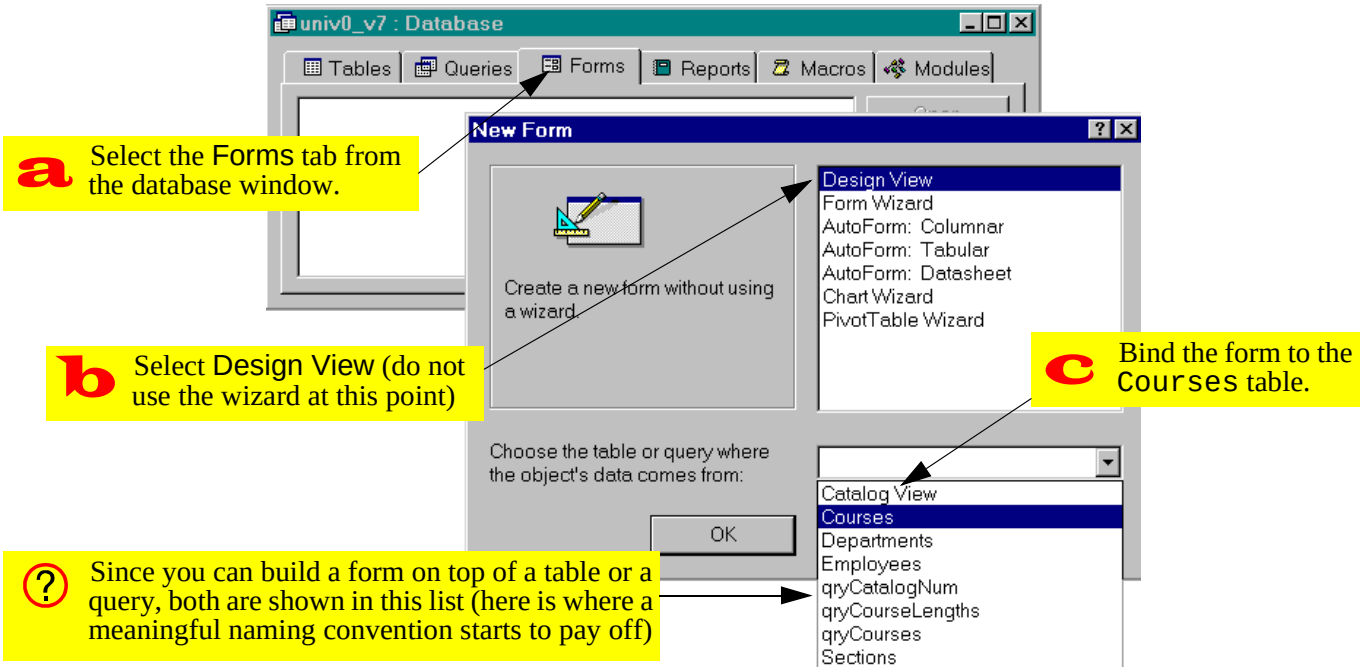


FIGURE 6.3: The basic elements of the form design screen.

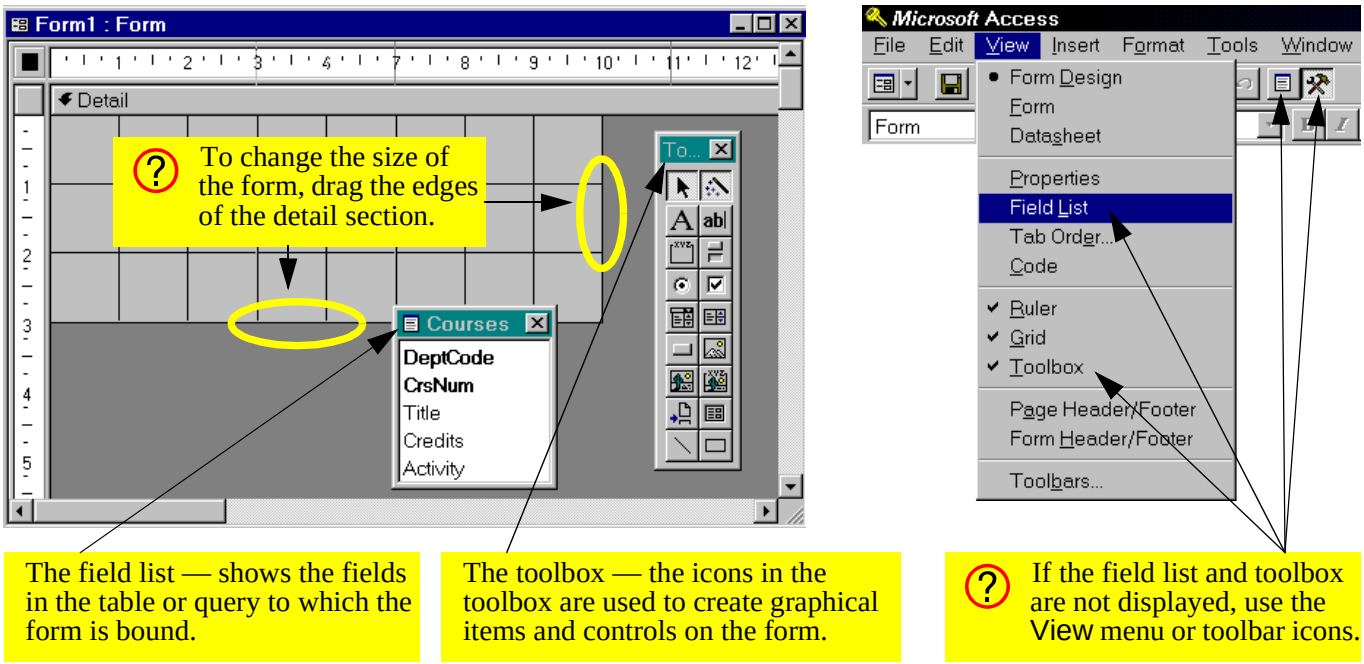
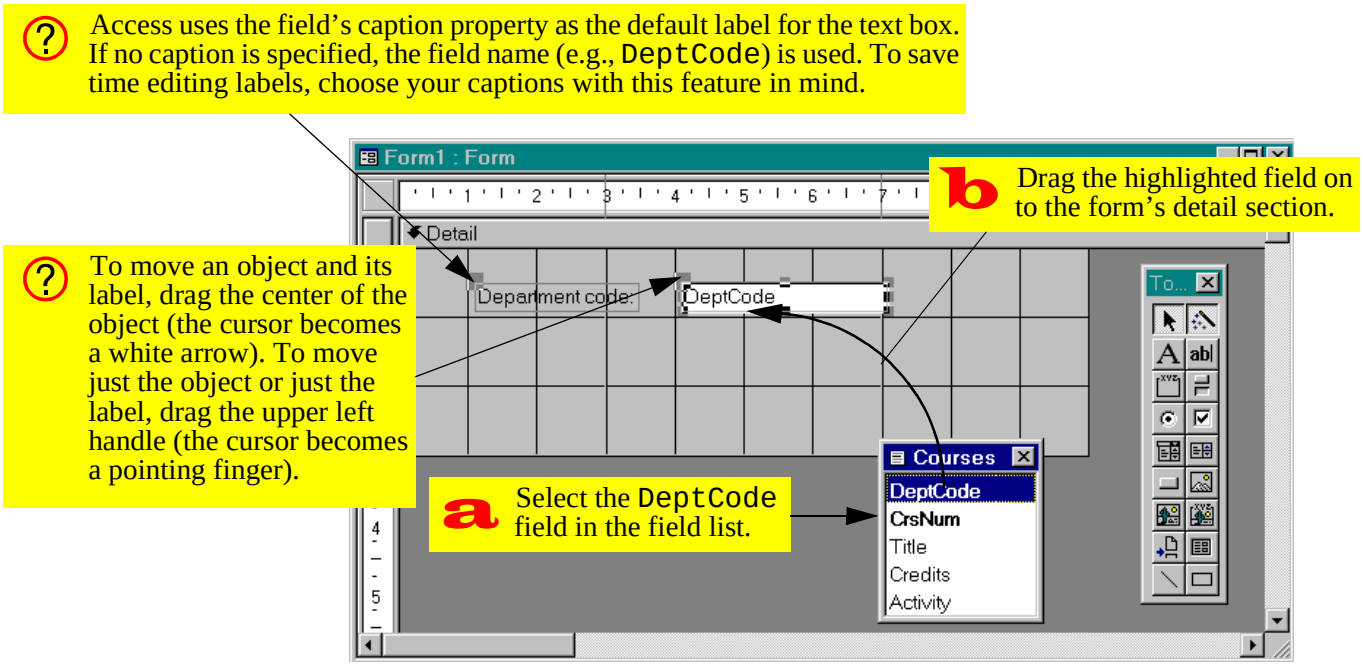


FIGURE 6.4: Create a bound text box for the `DeptCode` field.



- Drag the remaining fields on to the form, as shown in Figure 6.5 (do not worry about whether the fields are lined up perfectly).
- Select *View > Form* to see the resulting form. Alternatively, press the form view icon ( ).
- Select *View > Form Design* or press the design view icon ( ) to return to design mode.

6.3.1.2 Using a field's properties to protect its contents

Every object on an Access form (e.g., text box, label, detail section, etc.) has a set of properties that can be modified. In this section, you are going to use the *Locked* and *Enabled* properties to control the user's ability to change the information in a field.

- Select the `DeptCode` text box and right-click to bring up its property sheet, as shown in Figure 6.6.
- Scroll down the property sheet to the *Locked* property and set it to `Yes`, as shown in Figure 6.7.
- Switch to the form view and attempt to change the contents of the `DeptCode` field.

A stronger form of protection than locking a field is "disabling" it.

- Return to design mode and make the following changes: reset the *Locked* property to `No`; set the *Enabled* property to `No`.
- Attempt to change the contents of the `DeptCode` field in form view, as shown in Figure 6.8.
- Save the form as `frmCourses`.

6.3.1.3 Adding an unbound text box

All the text boxes created in the previous section were "bound" text boxes—that is, they were bound to a field in the underlying table or query. When you change the value in a bound text box, you are mak-

FIGURE 6.5: Add the text boxes and switch to form view to see the resulting form.

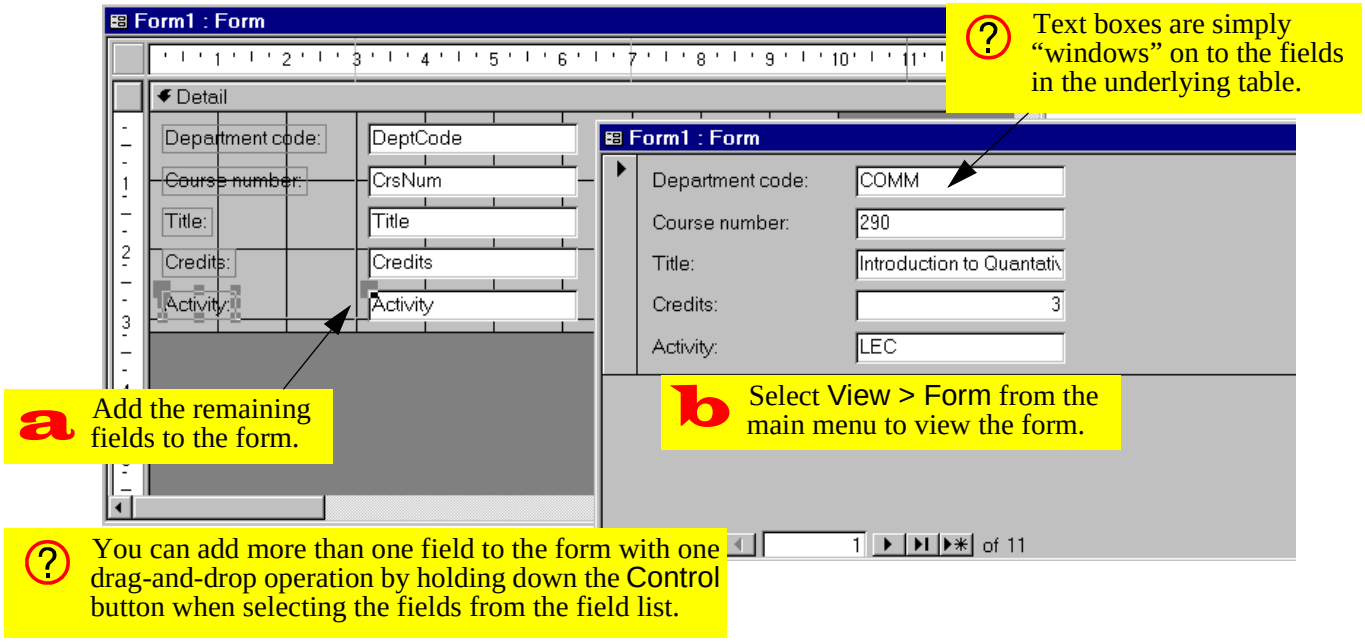


FIGURE 6.6: Bring up the property sheet for the `DeptCode` text box.

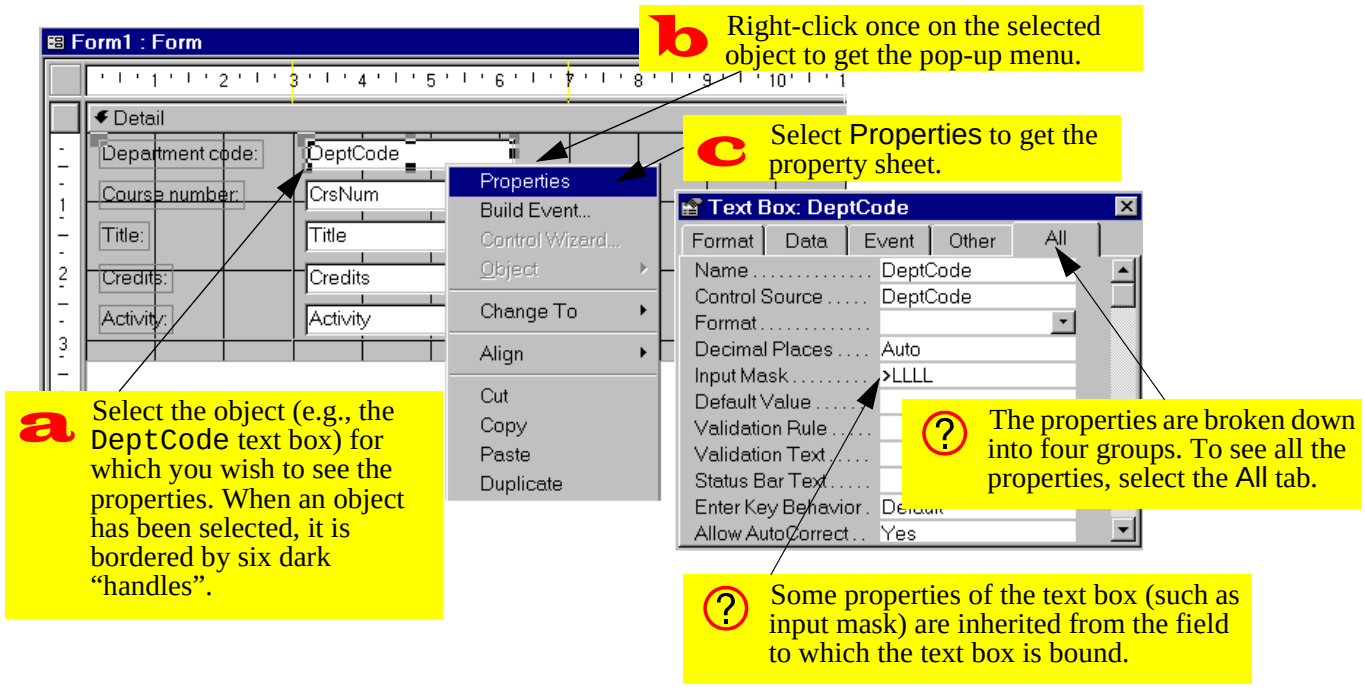
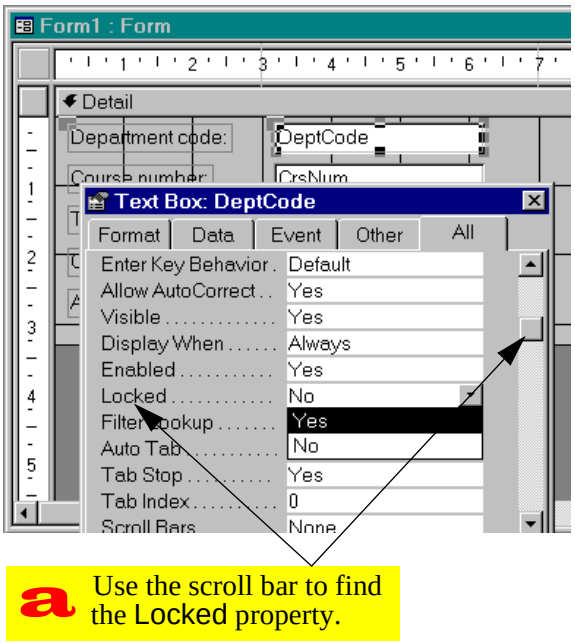


FIGURE 6.7: Change the *Locked* property of `DeptCode` to `Yes`.



ing the change directly to the data in the underlying table.

It is possible, however, to create objects on forms that are not bound to anything. Although you will not use many "unbound" text boxes in the assignment, it is instructive to see how they work.

- Create a new empty form bound to the `Courses` table and save it using the name `frmCoursesUB`.
- Select the text box tool () from the toolbox and create and unbound text box, as shown in Figure 6.9.

6.3.1.4 Binding an unbound text box to a field

The only difference between a bound and an unbound text box is that the *Control Source* property of a bound text box is set to the name of a field. In this section, you are going to change the unbound text box shown in Figure 6.9 to a bound text box.

FIGURE 6.8: Set the *Enabled* property of `DeptCode` to `No` and attempt to change the value in the field.

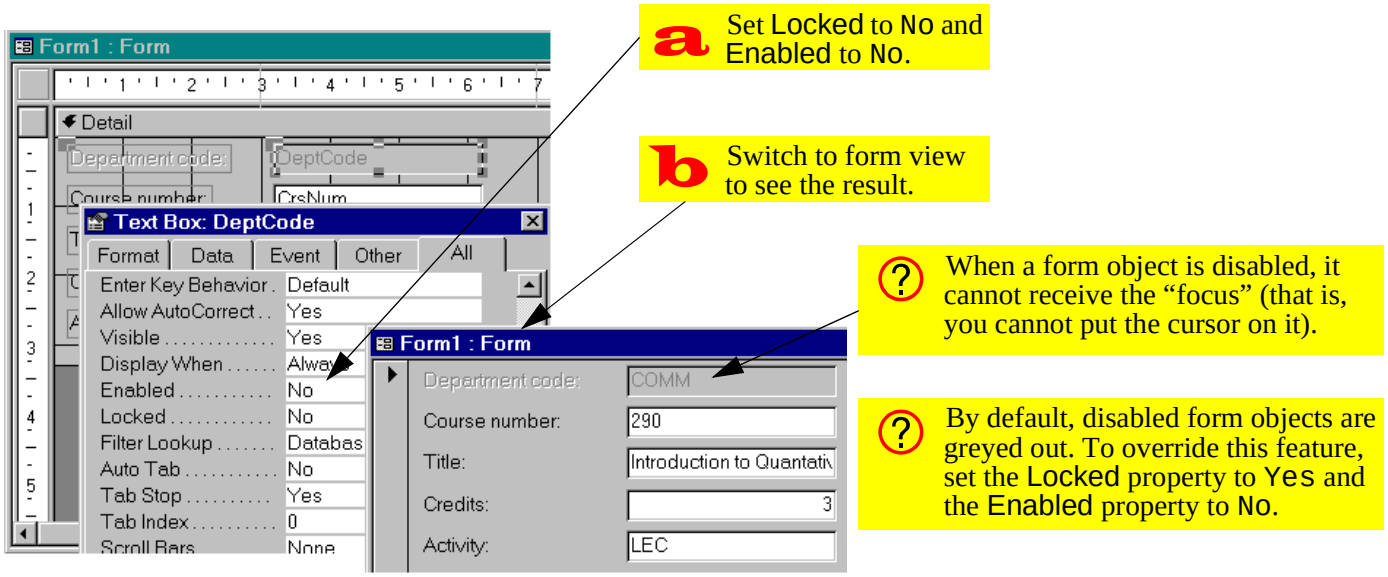
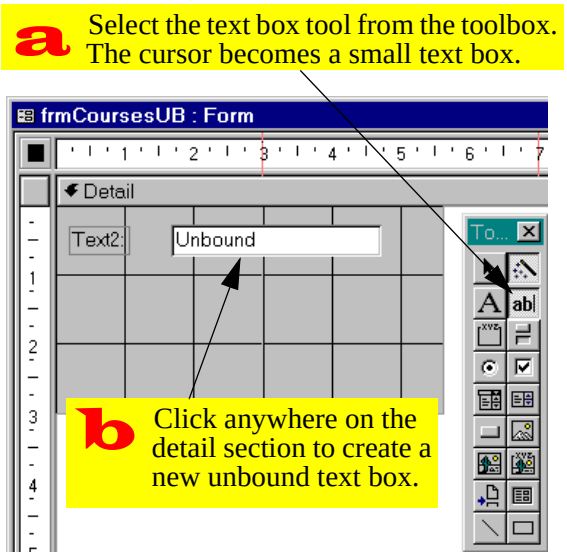


FIGURE 6.9: Create an unbound text box.



- Bring up the property sheet for the unbound text box. Change its *Control Source* property from null to `DeptCode`, as shown in Figure 6.10.

6.3.2 Creating a single-column form using the wizard

Now that you understand the basics of creating and modifying bound text boxes, you can rely on the form wizard to create the basic layout of all your forms.

- Create a new form bound to the `Courses` table using the form wizard, as shown in Figure 6.11.
- Use the form wizard to specify the fields you want on your form and the order in which they appear, as shown in Figure 6.12. Select "columnar" when prompted for the form type.

"Columnar" forms are called "single column" forms in version 2.0.

FIGURE 6.10: Set the Control Source property of an unbound text box.

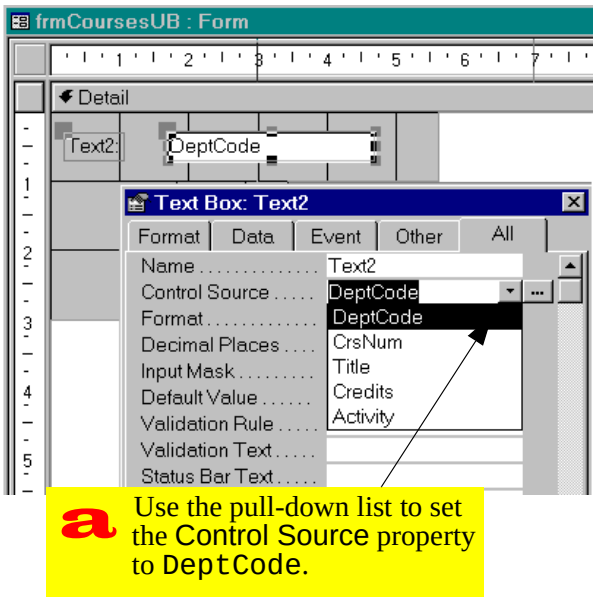


FIGURE 6.11: Create a new form using the form wizard.

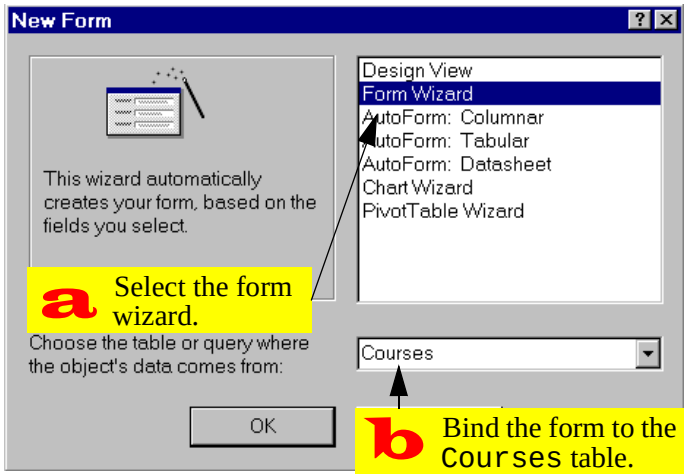
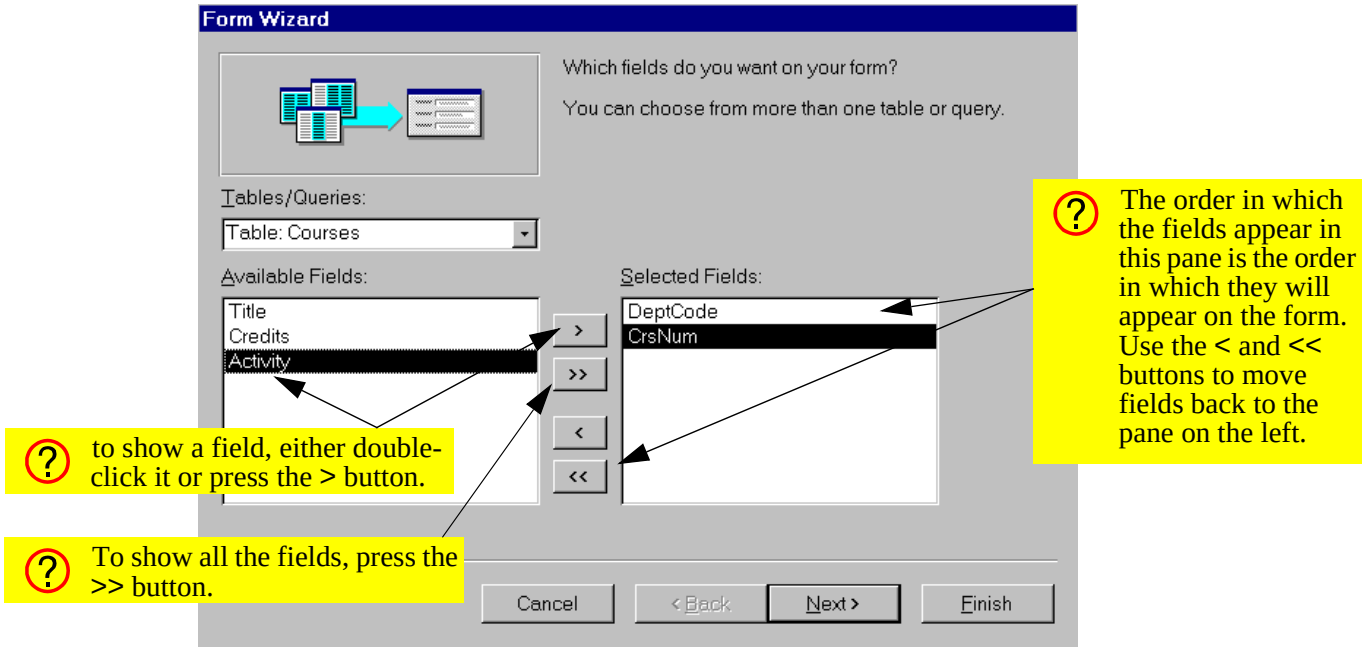


FIGURE 6.12: Use the form wizard to determine the order of fields on your form.



The primary advantage of the wizard is that it automatically creates, formats, and aligns the bound text boxes. Of course, once the wizard has created a form, you are free to modify it in any way.

If you make a mistake when creating a form (e.g., you put the fields in the wrong order) it is often easier to use the wizard and start over than to fix the problem manually.

6.4 Discussion

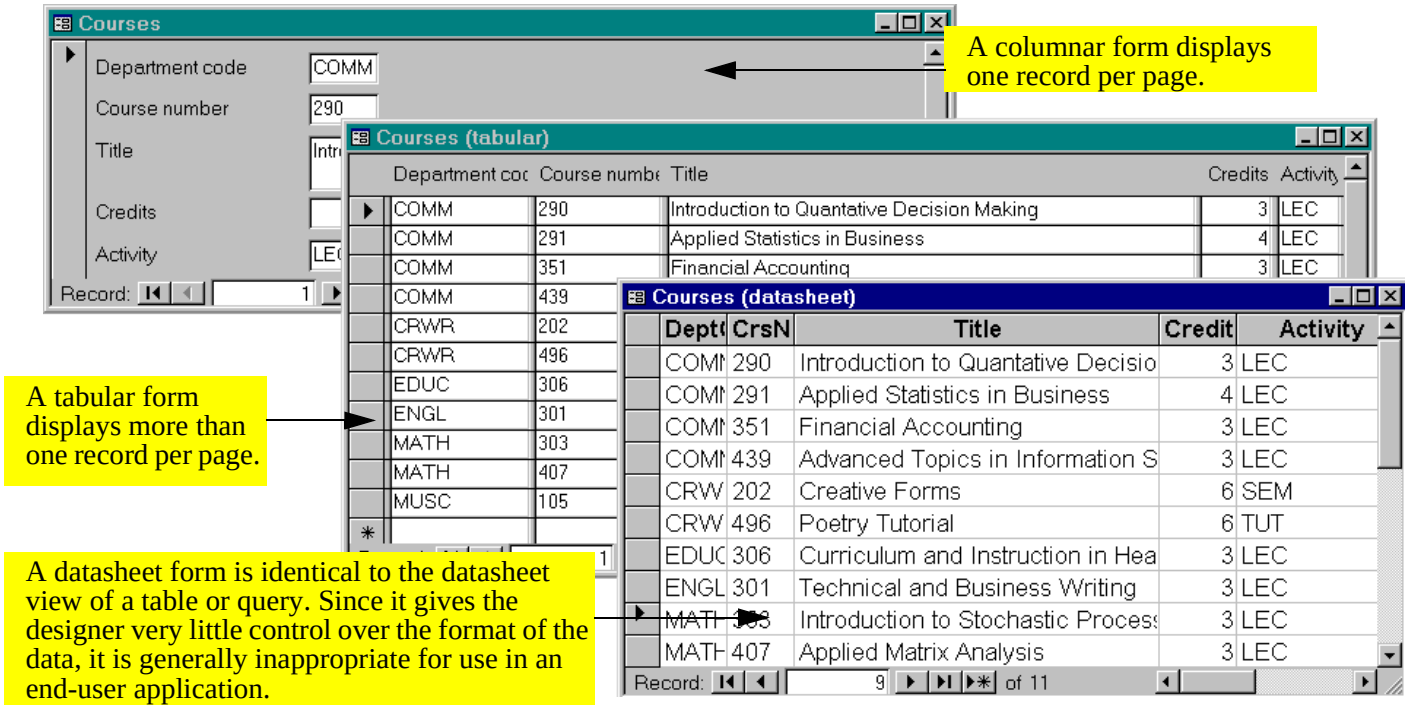
6.4.1 Columnar versus tabular versus datasheet forms

Columnar forms show one record per page. **Tabular** forms, in contrast, show many records per page and are used primarily as subforms. There is also a a **datasheet** form type, but it is seldom used since it gives the developer relatively little control over the look and behavior of the data. The three different types of forms are shown in Figure 6.13.

6.5 Application to the assignment

- Use the wizard to create columnar forms for all your master tables. Note that in some cases (e.g., `BackOrders`) you will want to base the form on a join query rather than table in order to show important information such as `CustName` and `ProductName`.

FIGURE 6.13: The same information displayed as a columnar, tabular, and datasheet form.



Access Tutorial 7: Subforms

7.1 Introduction: The advantages of forms within forms

A columnar/single-column main form with a tabular subform is a natural way of representing information from tables with a one-to-many relationship. For example, the form shown in Figure 7.1 is really two forms: the main form contains information about a specific course; the subform shows all the sections associated with the course.

In the `Courses` and `Sections` example, the foreign key (`DeptCode` and `CrsNum`) provides a link between the two forms. This connection allows Access to **synchronize** the forms, meaning:

- when you move to another course record, only the relevant sections are shown in the subform;
- when you add a new section, the foreign key in the `Sections` table is automatically filled in (in fact, there is no need to show `DeptCode` and `CrsNum` in the subform).

Although you will quickly learn to take a feature such as form/subform synchronization for granted, it is worthwhile to consider what this feature does and what it would take if you had to implement the same feature using a programming language.

7.2 Learning objectives

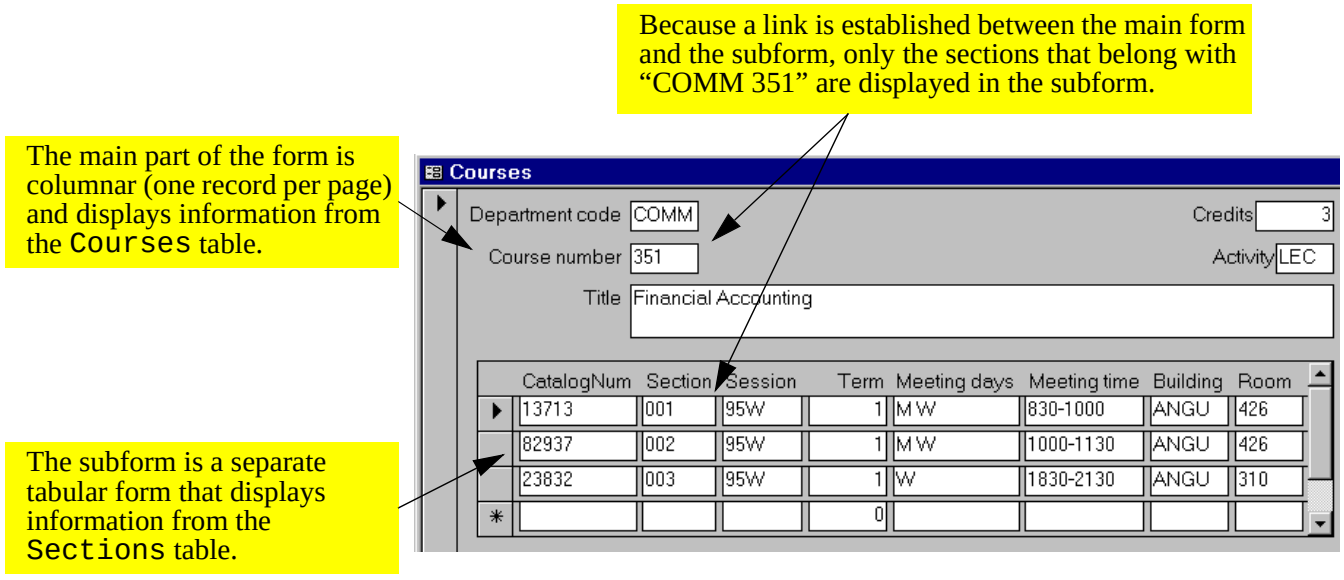
- ❒ What is form/subform synchronization?
- ❒ How do I create a form/subform combination?
- ❒ How do I link a form with a subform?

7.3 Tutorial exercises

Although there are a number of different ways to create a subform within a main form, the recommended procedure is the following:

Last update: 25-Aug-1997

FIGURE 7.1: A typical form/subform combination.



1. create and save both forms (one columnar, one tabular) separately;
2. drag the subform on to the main form; and,
3. verify the linkage between the two forms.

7.3.1 Creating the main form

- Use the wizard to create a columnar form based on the `Courses` table.
- Rearrange the fields so that they make efficient use of the top part of the form, as shown in Figure 7.2.
- Save the form as `frmCoursesMain`.

7.3.2 Creating the subform

- Use the wizard to create the subform, as shown in Figure 7.3 and Figure 7.4.
- Subforms created by the wizard typically require some fine tuning in order to reduce the amount of space they occupy. A number of editing issues are highlighted in Figure 7.5.
- Save the form as `sfrmSections` and close it.

7.3.3 Linking the main form and subform

In this section, you are going to return to the main form and drag the saved subform from the database window to an appropriate position on the main form.

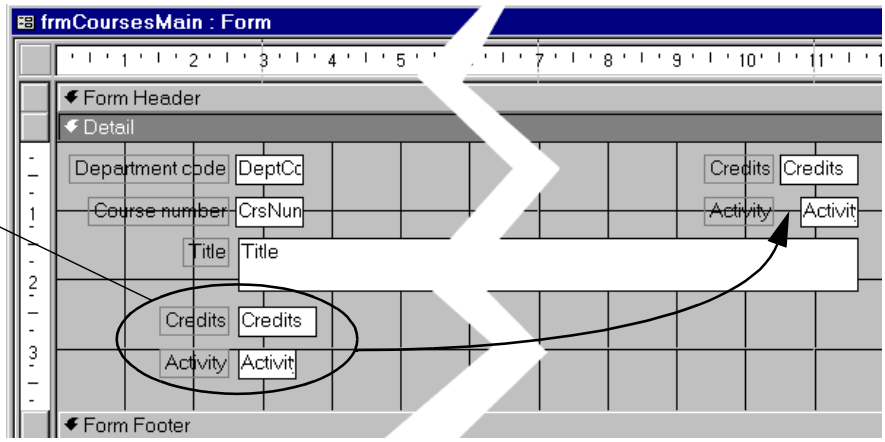
- Open the main form (`frmCoursesMain`) in design mode.
- Select *Window > univ0_vx: Database* to open the database window in the foreground. Alternatively, you can press the database window icon ( ) on the tool bar.
- Perform the steps shown in Figure 7.6 to drag the subform on to the main form.
- The result of the drag-and-drop operation are shown in Figure 7.7. The advantage of the drag-and-drop method of creating a sub form is that

FIGURE 7.2: Rearrange the text boxes on the main form to make room for the subform.

a Use the wizard to create a columnar form based on `Courses`.

b Enter form design mode and rearrange the text boxes to make room for the subform.

c Save the form under the name `frmCoursesMain`.



? To move more than one form object at a time, either hold down the Shift key when selecting or drag a box through the objects (click and drag to create a box).

FIGURE 7.3: Use the wizard to create the `Sections` subform (part 1).

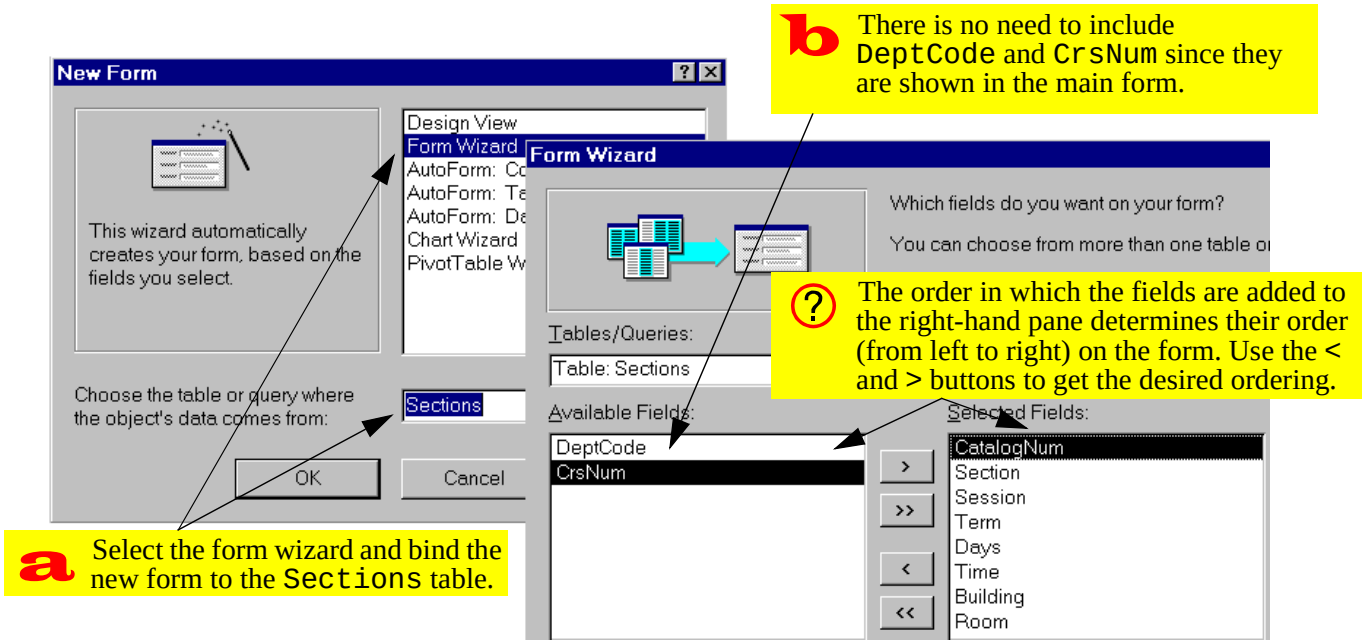


FIGURE 7.4: Use the wizard to create the `Sections` subform (continued)

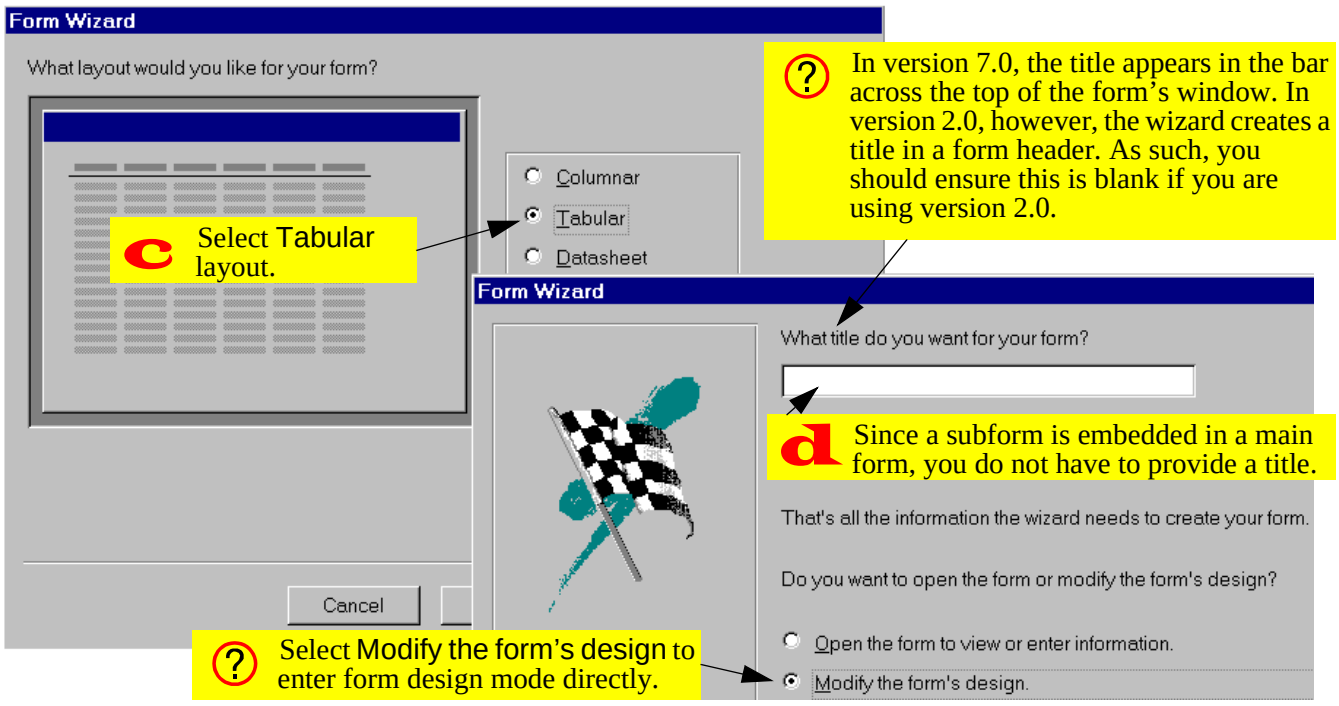


FIGURE 7.5: Edit the subform to reduce the amount of space it uses.

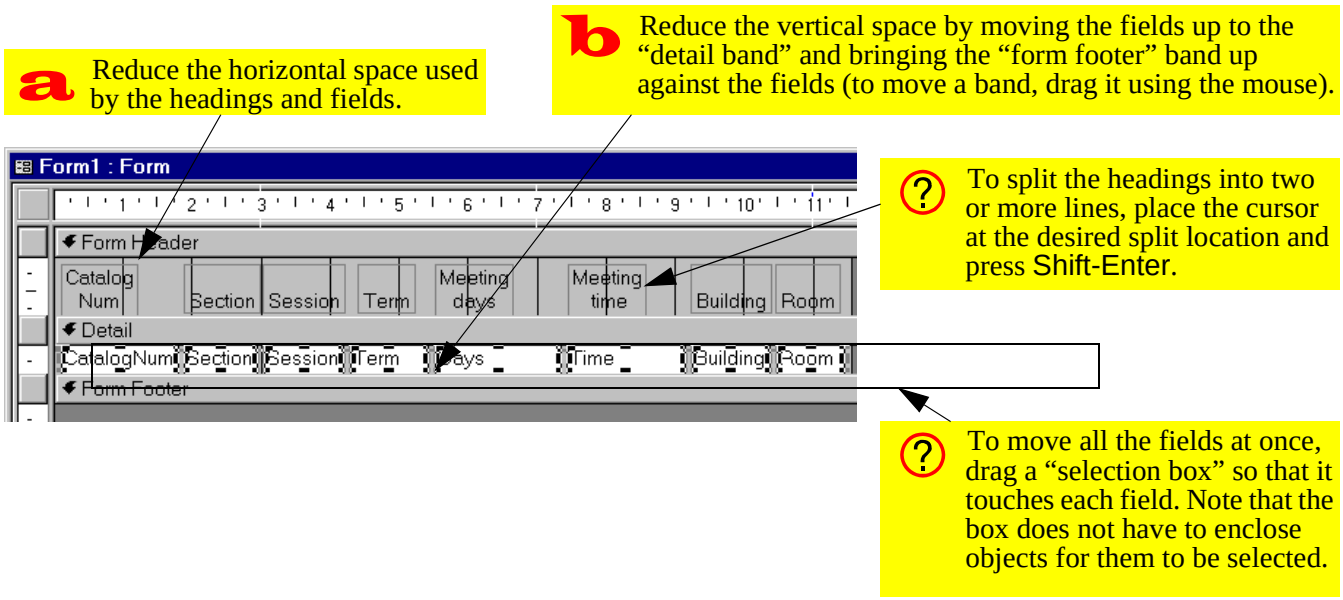
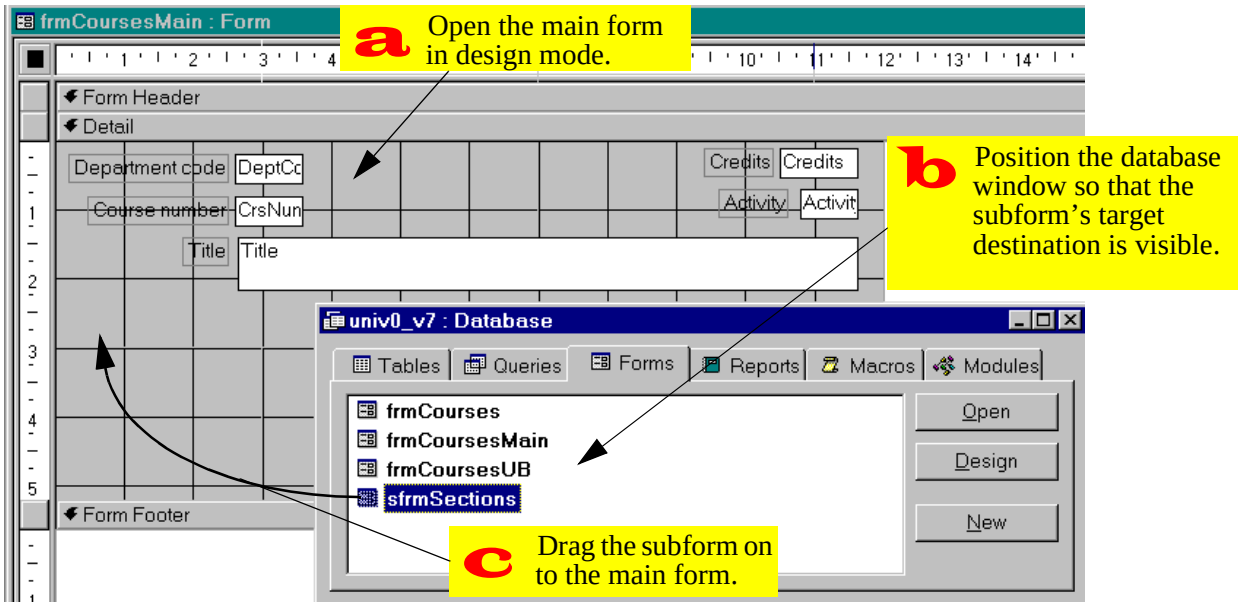


FIGURE 7.6: Drag the subform on to the main form.



the width of the subform control (the white window) is automatically set to equal the width of the subform.

If you make changes to the size of your subform once the subform control is created, you may have to resize the subform control by clicking and dragging a corner handle.

7.3.4 Linking forms and subforms manually

If both the form and the subform are based on tables, and if relationships have been defined between the tables, Access normally has no problem determining which fields "link" the information on the main form with the information in the subform. However, when the forms are built on queries, Access has no relationship information to rely on. As such, you have to specify the form/subform links manually.

Since both the forms created in Section 7.3.3 were built on tables, Access could automatically determine the relationship.

- Verify the link between the form and the subform by examining the property sheet of the subform control, as shown in Figure 7.8.

The terminology "link child field" and "link master field" is identical to "foreign key" and "primary key". The main form is the parent ("one" side) and the subform is the child ("many" side).

- View the resulting form. Notice that as you move from course to course, the number of sections shown in the subform changes (see Figure 7.9).

FIGURE 7.7: The drag-and-drop operation creates a subform control.

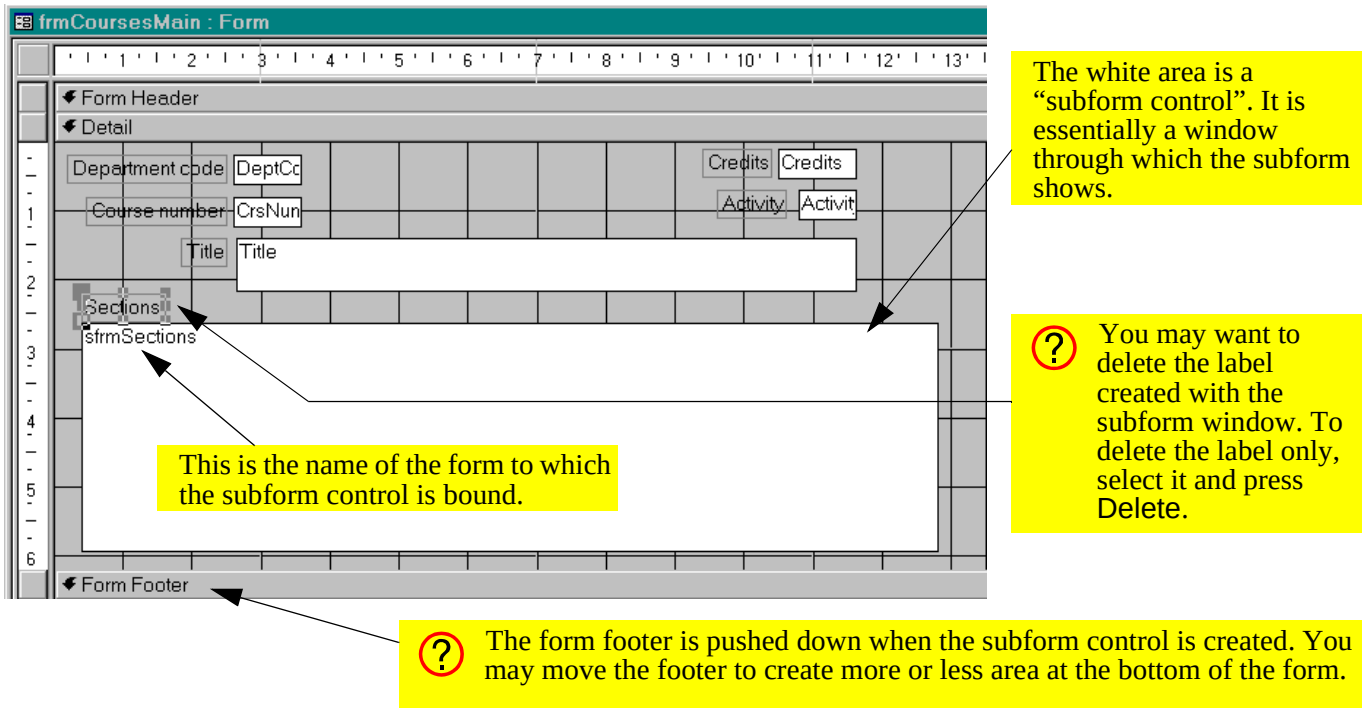


FIGURE 7.8: Verify the link fields for the form/subform.

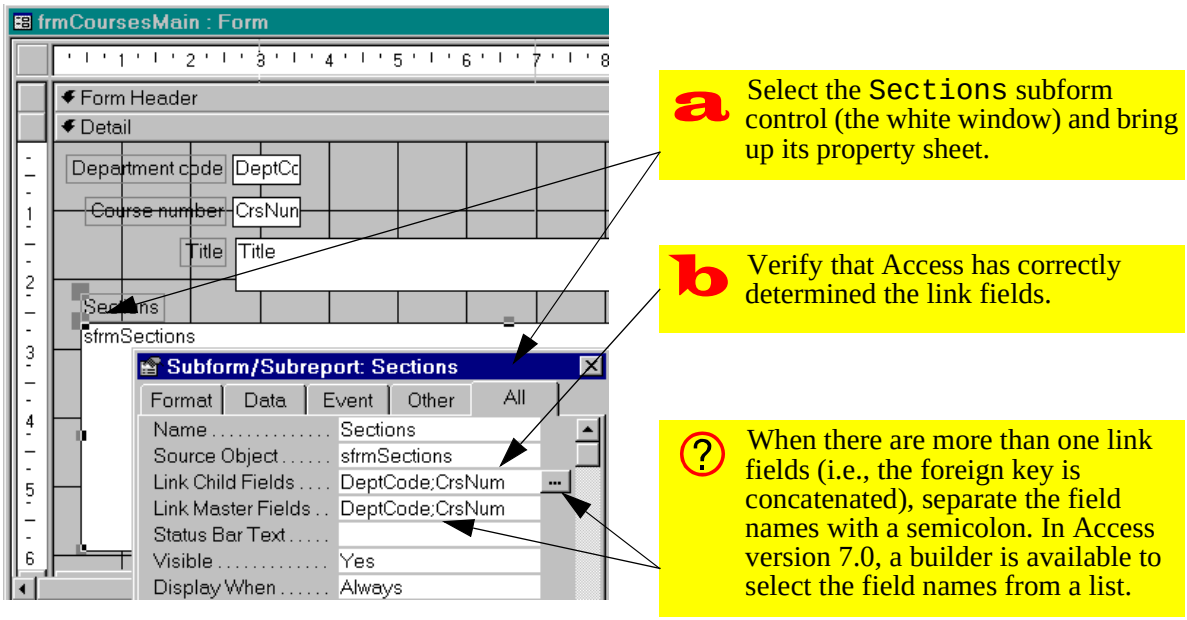
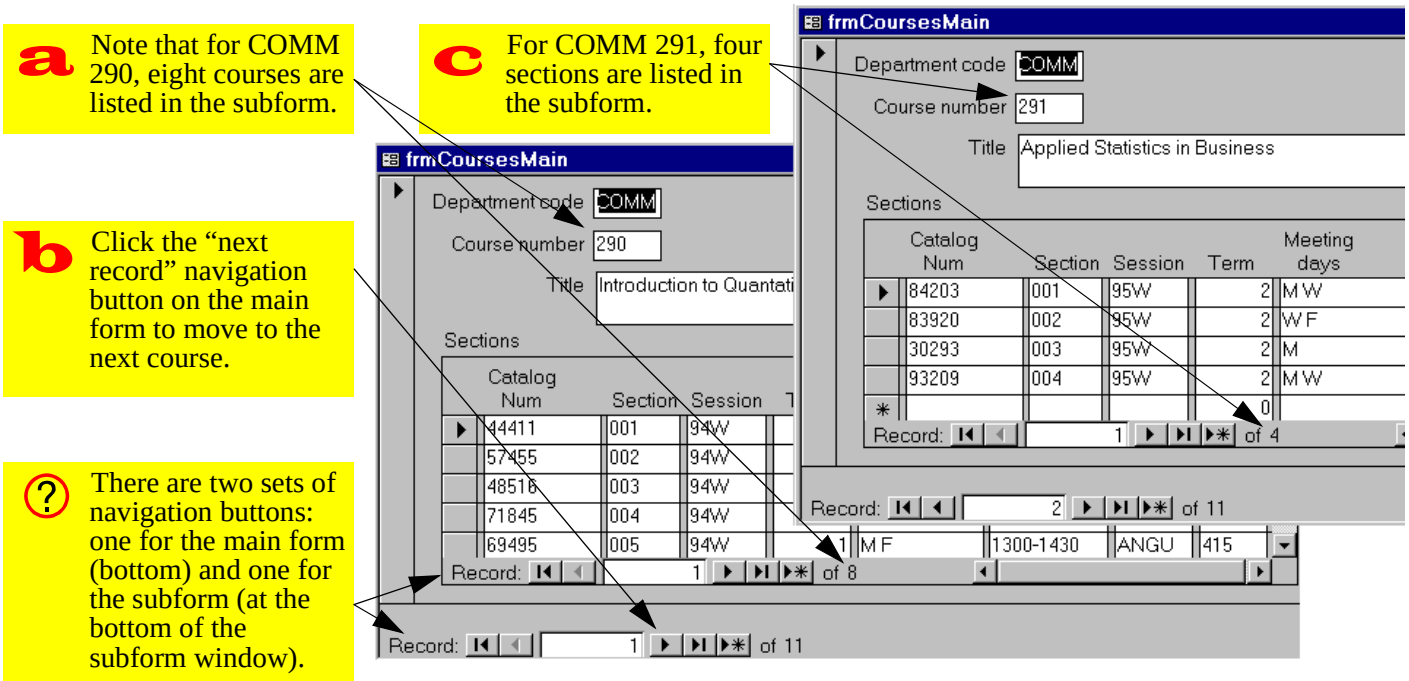


FIGURE 7.9: A synchronized main form/subform.



7.3.5 Non-synchronized forms

In this section, you will delete the link fields shown in Figure 7.8 in order to explore some of the problems associated with non-synchronized forms.

- Return to form design mode and delete the link fields (highlight the text and press the *Delete* key).
- View the form. Note that all records in the `Sections` table (not just those associated with a particular course) are shown.
- Attempt to add a new section to COMM 290 as shown in Figure 7.10.
- Re-establish the correct link fields and save the form.

7.3.6 Aesthetic refinements

In this section, you will modify the properties of several form objects (including the properties of the form itself) to make your form more attractive and easier to use.

In Figure 7.11, the basic form created in the previous sections is shown and a number of shortcomings are identified.

7.3.6.1 Changing the form's caption

- Select the form as shown in Figure 7.12.
- Change its *Caption* property to "Courses and Sections".

7.3.6.2 Eliminating unwanted scroll bars and navigation buttons

Scroll bars and navigation buttons are also form-level properties. However, in this case, you need to modify the properties of the subform.

- To quickly open the subform in design mode, double-click the subform control when viewing the main form in design mode (this takes some practice)

FIGURE 7.10: A non-synchronized main form/subform.

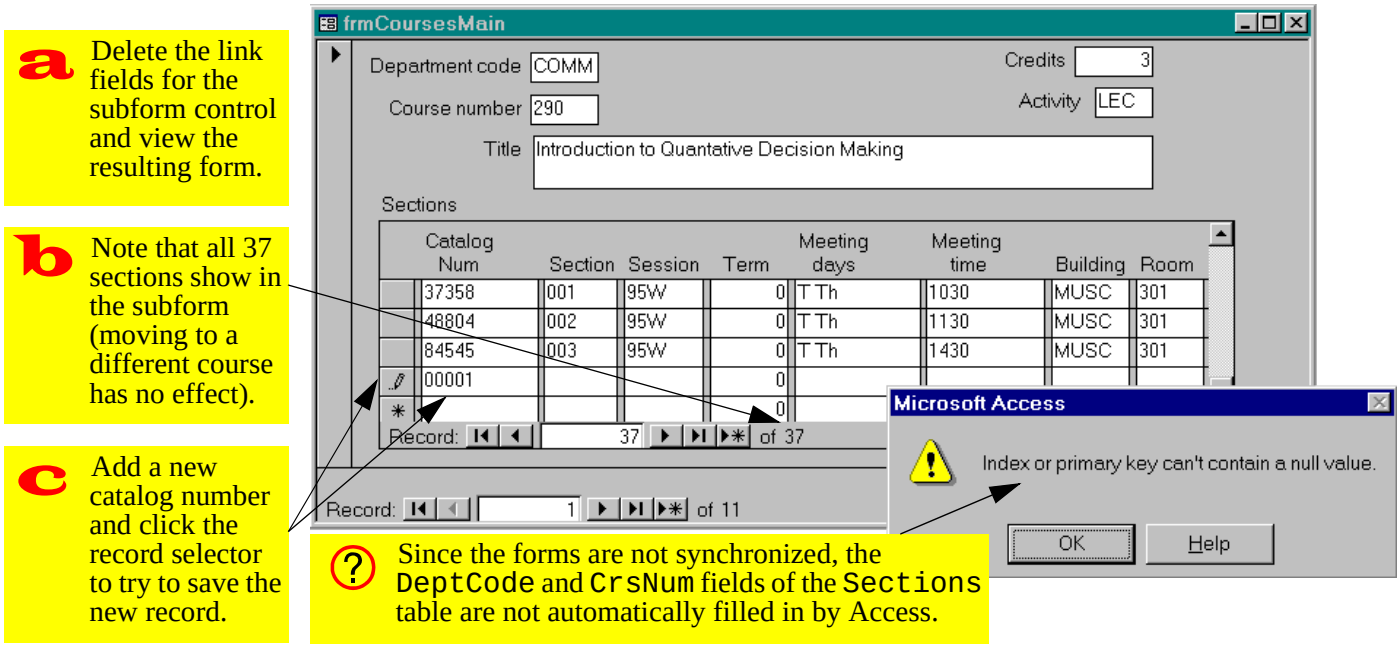


FIGURE 7.11: A form/subform in need of some basic aesthetic refinements.

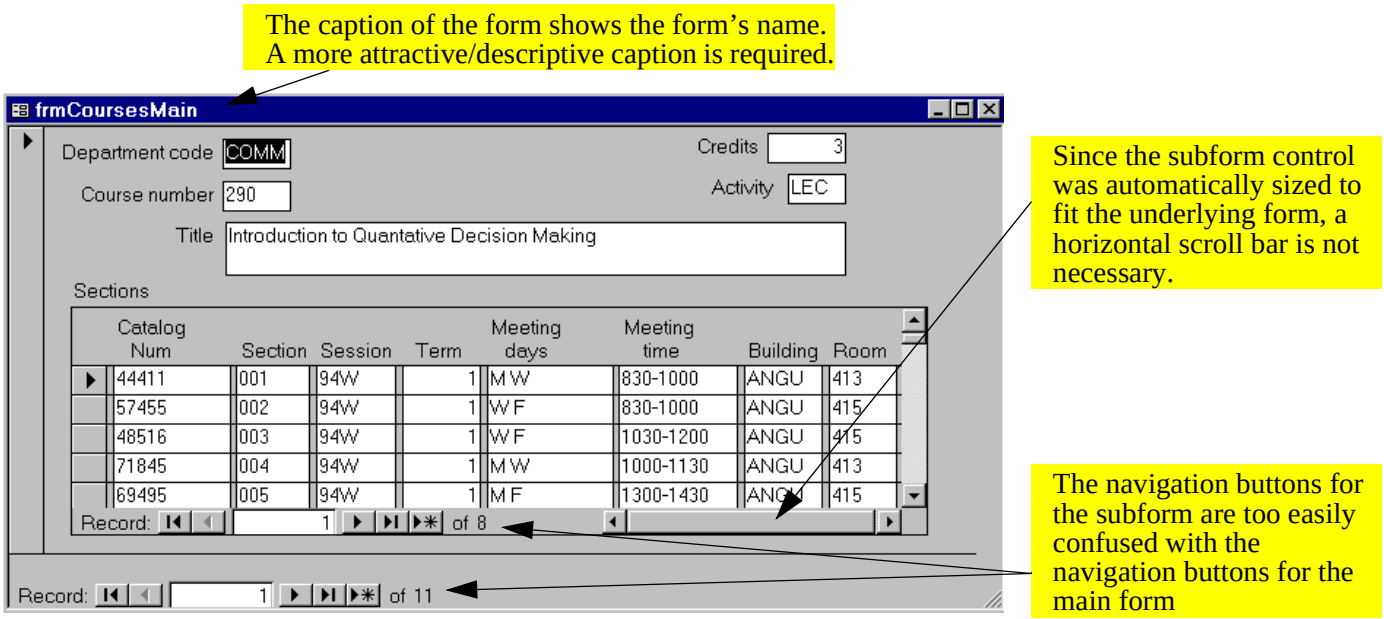
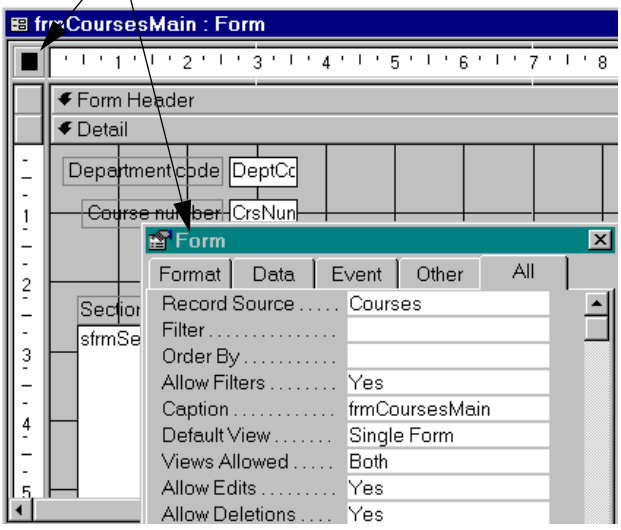


FIGURE 7.12: Select the entire form.



- Bring up the property sheet for the form and scroll down to change its *Scroll Bars* and *Navigation Button* properties, as shown in Figure 7.13.

The net result, as shown in Figure 7.14, is a more attractive, less cluttered form.

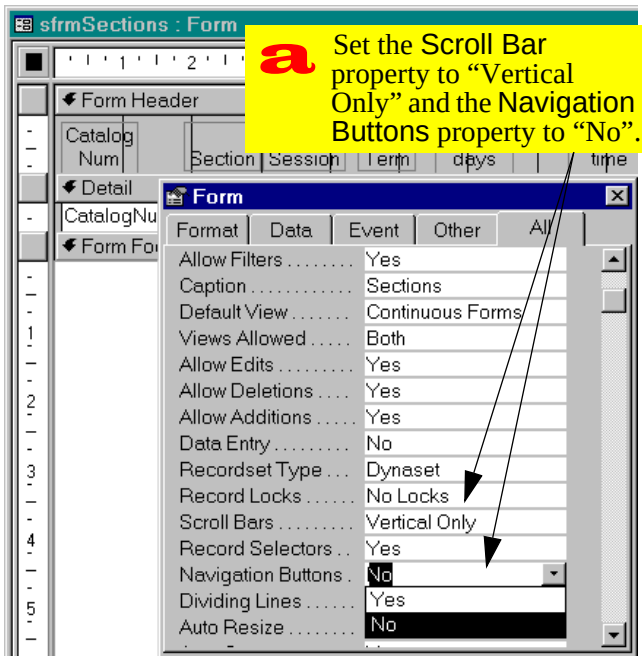
7.4 Application to the assignment

- Create a form and subform for your `Shipment` and `ShipmentDetails` information. You will use this form to record the details of shipments **from your suppliers**.

Note that both forms should be based on queries:

- the `Shipment` form should be based on a "sort" query so that the most recent shipment always shows first;
- the `ShipmentDetails` form should be based on a join query so that validation information (such as the name of the product) is shown when a product number is entered.

FIGURE 7.13: Change the scroll bars and navigation buttons of the subform.

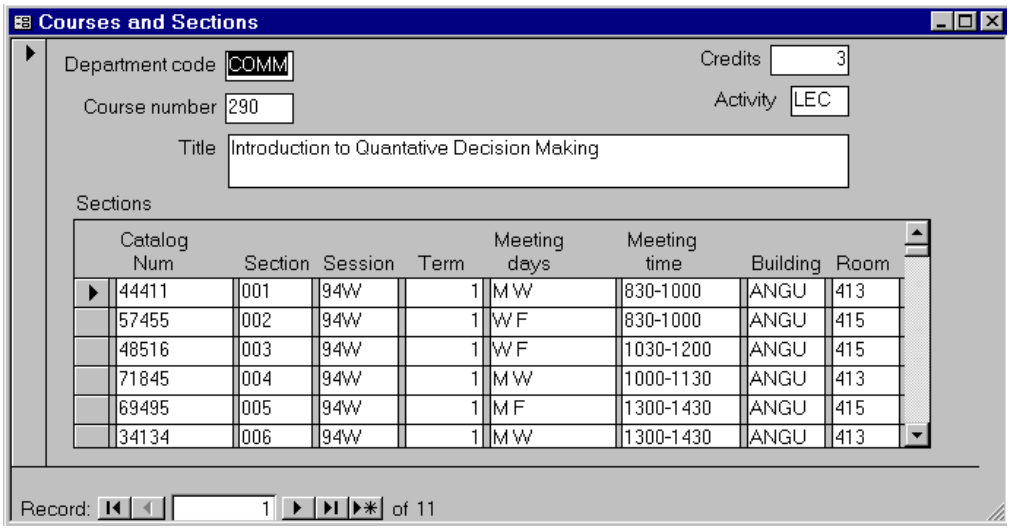


- Create a form/subform to show customer orders that have already been placed (such as the one you entered manually in Section 4.5). The top part of the form should contain information about the order plus some information about the customer; the subform should contain information about what was ordered and what was actually shipped.

? The form you created in the preceding step is used for viewing existing orders, not for adding new orders. To add new orders, the form must be more complex. For example, it has to show the quantity on hand and the back ordered quantity for each item so the user can decide how many to ship. You will create a form for order entry in the latter tutorials.

- Set the *Allow Additions* and *Allow Edits* properties of the "order viewing" form to `No`. This pre-

FIGURE 7.14: A form without subform scroll bars or navigation buttons.



vents the user from changing the details of an order that has already been invoiced or attempting to use the form for order entry.

Access Tutorial 8: Combo Box Controls

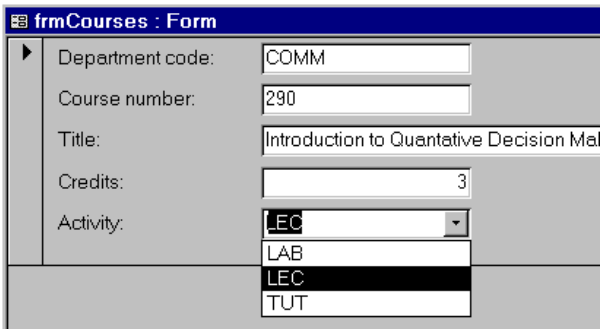
8.1 Introduction: What is a combo box?

So far, the only kind of "control" you have used on your forms has been the text box. However, Access provides other controls (such as combo boxes, list boxes, check boxes, radio buttons, etc.) that can be used to improve the attractiveness and functionality of your forms.

A combo box is list of values from which the user can select a single value. Not only does this save typing, it adds another means of enforcing referential integrity since the user can only pick values in the combo box. For example, a combo box for selecting course activities from a predefined list is shown in Figure 8.1.

Although advanced controls such as combo boxes and list boxes look and behave very differently than simple text boxes, their function is ultimately the same. For example, in Figure 8.1, the combo box is bound to the `Activity` field. When an item in the combo box is selected, the string (e.g., "LEC") is copied into the underlying field exactly as if you had typed the letters L-E-C into a text box.

FIGURE 8.1: A combo box for filling in the `Activity` field.



Last update: 25-Aug-1997

Learning objectives

- What is tab order? How do I change it so that the cursor moves in the correct order?
- Should I put a combo box on a key field?

It is important to realize that combo boxes have no intrinsic search capability. Combo boxes change values—they do not automatically move to the record with the value you select. If you want to use a combo box for search, you have to program the procedure yourself (see Tutorial 15 for more details).

8.2 Learning objectives

- How do I create a bound combo box?
- Can I create a combo box that displays values from a different table?
- How do I show additional information in a combo box?
- How do I prevent certain information from showing in the combo box?
- Can I change the order in which the items appear in a combo box?

8.3 Tutorial exercises

- Open your `frmCourses` form in design mode.
- Ensure the toolbox and field list are visible (recall Figure 6.3).

8.3.1 Creating a bound combo box

Although Access has a wizard that simplifies the process of creating combo boxes, you will start by building a simple combo box (similar to that shown in Figure 8.1) with the wizard turned off. This will give you a better appreciation for what the wizard does and provide you with the skills to make refinements to wizard-created controls.

- Delete the existing `Activity` text box by selecting it and pressing the *Delete* key.

- The wizard toggle button ( ) in the toolbox allows you to turn wizard support on and off. Ensure the button is out (wizards are turned off).
- Click on the combo box tool ( ). The cursor turns into a small combo box.
- With the combo box tool selected, drag the `Activity` field from the field list to the desired location on the form's detail section, as shown in Figure 8.2.

The process of selecting a tool from the toolbox, and then using the tool to drag a field from the field list ensures that the control you create (text box, combo box, etc.) is bound to a field in the underlying table or query.

If you forget to drag the field in from the field list, you will create an unbound combo box, as shown in Figure 8.3. If you accidently create an unbound combo box, the easiest thing to do is to delete it and try again.

FIGURE 8.3: An unbound combo box (not what you want).

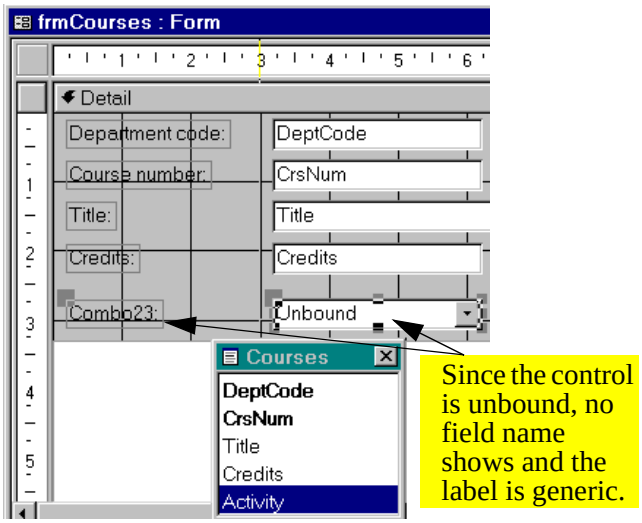
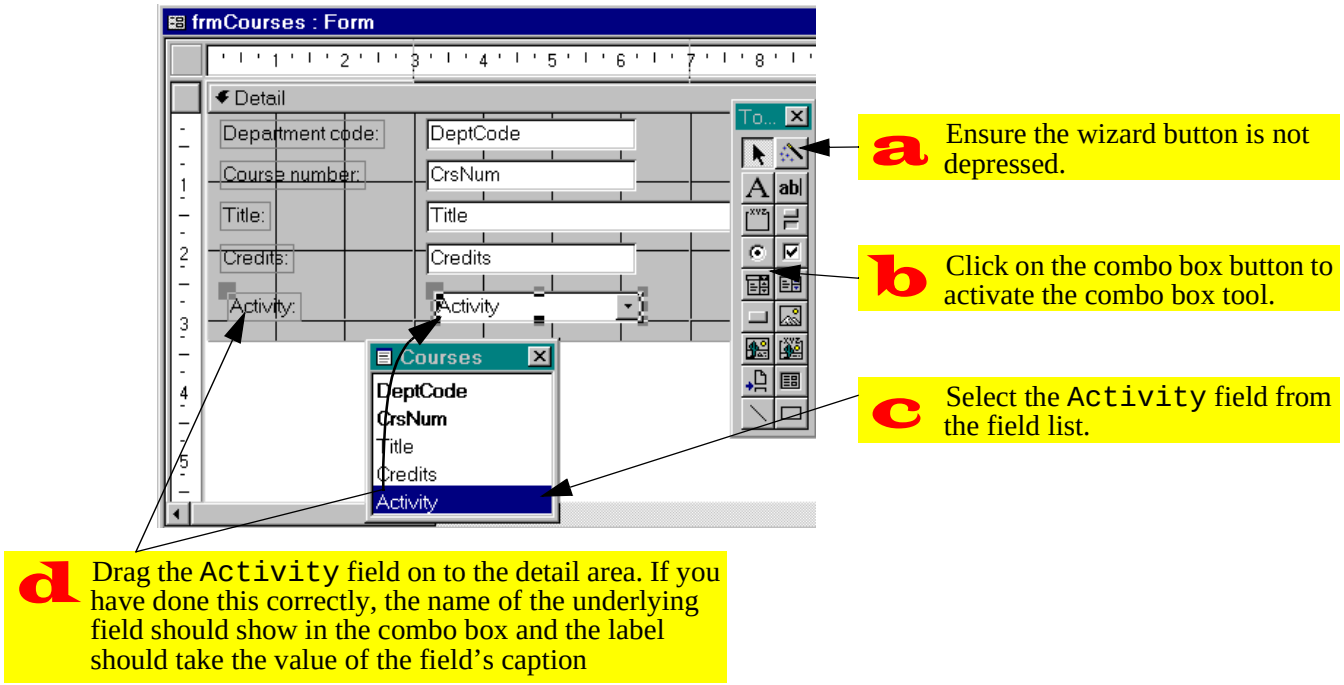


FIGURE 8.2: Create a bound combo box.



8.3.2 Filling in the combo box properties

In this section, you will tell Access what you want to appear in the rows of new combo box.

- Switch to form view and test the combo box.

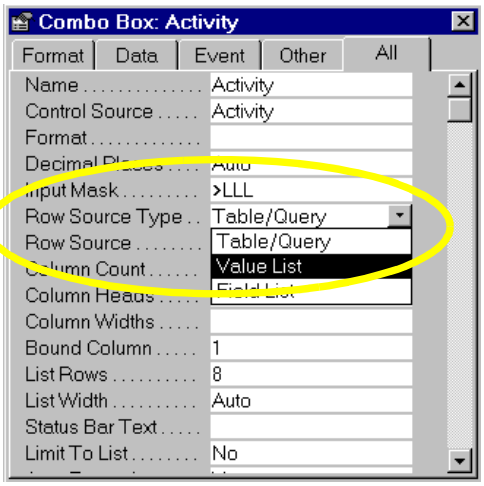
At this point, the combo box does not show any list items because we have not specified what the list items should be. There are three methods of specifying what shows up in the combo box list:

1. enter a list of values into the combo box's *Row Source* property;
2. tell Access to get the value from an existing table or query;
3. tell Access to use the names of fields in an existing table (you will not use this approach).

Although the second method is the most powerful and flexible, you will start with the first.

- Bring up the property sheet for the `Activity` combo box.
- Change the *Row Source Type* property to `Value List` as shown in Figure 8.4. This tells Access to expect a list of values in its *Row Source* property.

FIGURE 8.4: Set the *Row Source Type* property.



- Enter the following into the *Row Source* property: `LAB;LEC;TUT`
- Set the *Limit To List* property to `Yes`.

If the *Limit To List* property is set to `No`, the user can ignore the choices in the combo box and simply type in a value (e.g., "SEM"). In this particular situation, you want to limit the user to the three choices given.

- Switch to form view and experiment with the combo box.

Notice that the combo box has some useful built-in features. For example, if you choose to type values rather than select them with a mouse, the combo box anticipates your choice based on the letters you type. Thus, to select "TUT", you need only type "T".

8.3.3 A combo box based on another table or query

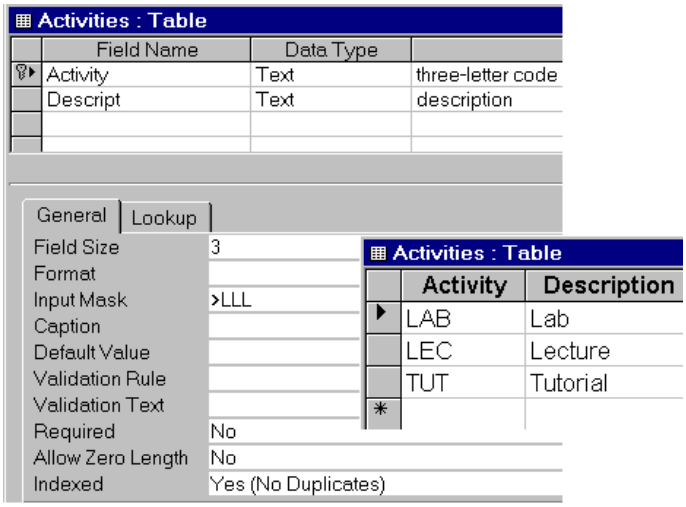
An obvious limitation of the value-list method of creating combo boxes is that it is impossible to change or update the items that appear in the list without knowing about the *Row Source* property.

A more elegant and flexible method of populating the rows of a combo box is to have Access look up the values from an existing table or query. Although the basic process of setting the combo box properties remains the same, it is more efficient to rely on the wizard when building this type of combo box.

Before you can continue, you need a table that contains appropriate values for course activities.

- Switch to the database window and create a new table called `Activities`.
- The table should consist of two fields: one called `Activity` and the other called `Descript`, as shown in Figure 8.5.

FIGURE 8.5: Create a table containing course activities.



- Populate the table with the same values used in Section 8.3.2.

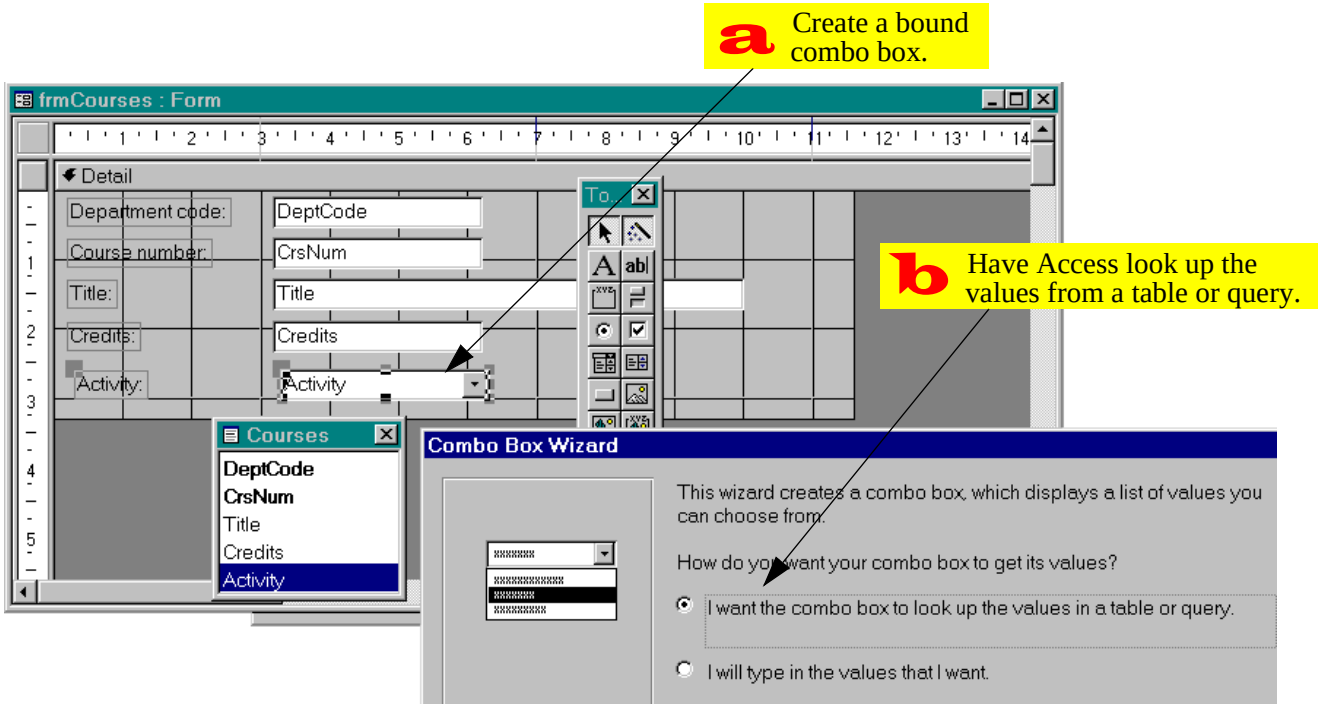
The result is a table containing all the possible course activities and a short description to explain the meaning of the three-letter codes. You can now return to creating a combo box based on these values.

- Delete the existing `Activity` combo box.
- Ensure the wizard button ( ) in the toolbox is depressed (wizards are activated).
- Repeat the steps for creating a bound combo box (i.e., select the combo box tool and drag the `Activity` field from the field list on to the detail section). As shown in Figure 8.6, this activates the combo box wizard.

The wizard asks you to specify a number of things about the combo box:

1. the table (or query) from which the combo box values are going to be taken;

FIGURE 8.6: Create a combo box using the combo box wizard.



2. the field (or fields) that you would like to show up as columns in the in the combo box;
3. the width of the field(s) in the combo box (see Figure 8.7);
4. the column from the combo box (if more than one field is showing) that is inserted into the underlying field; and,
5. the label attached to the field (see Figure 8.8).

When you are done, the combo box should look similar to that shown in Figure 8.1. However, updating or changing the values in the combo box is much easier when the combo box is based on a table.

- Add “SEM” (Seminar) to the `Activities` table.
- Return to the form, click on the Activity combo box, and press *F9* to **requery** the combo box.
- Verify that “SEM” shows up in combo box.

Access creates the rows in a combo box when the form is opened. If the values in the source table or query change while the form is open these changes are not automatically reflected in the combo box rows. As a consequence, you have to either (a) close and reopen the form, or (b) requery the form. Although you can automate the requery process, we will rely on the *F9* key for the time being.

8.3.3.1 Showing more than one field in the combo box

One problem the combo boxes created so far is that they are not of much use to a user who is not familiar with the abbreviations “TUT”, “SEM”, and so on. In this section, you will use the `Descript` field of the `Activities` table to make the combo box more readable, as shown in Figure 8.9.

- Delete the existing combo box and start again.

FIGURE 8.7: Fill in the combo box wizard dialog sheets.

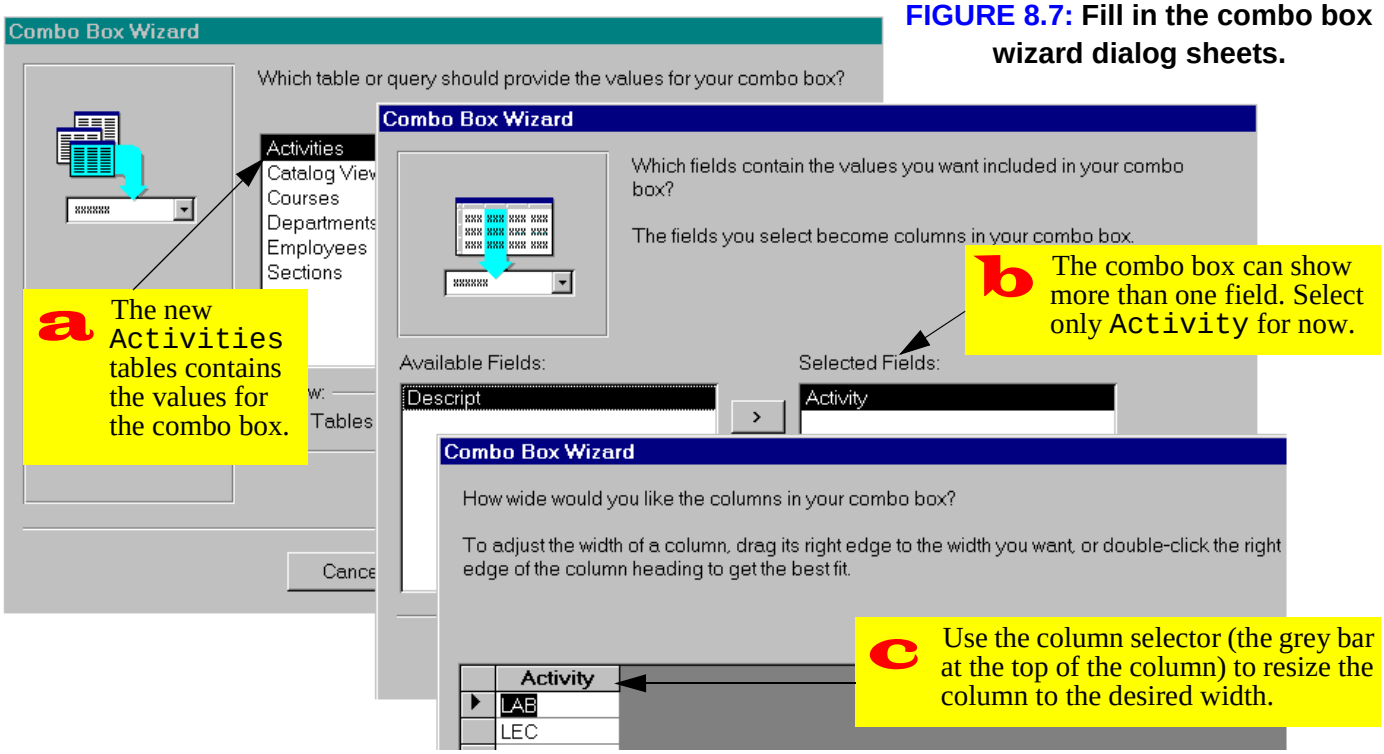


FIGURE 8.8: Fill in the combo box wizard dialog sheets (continued).

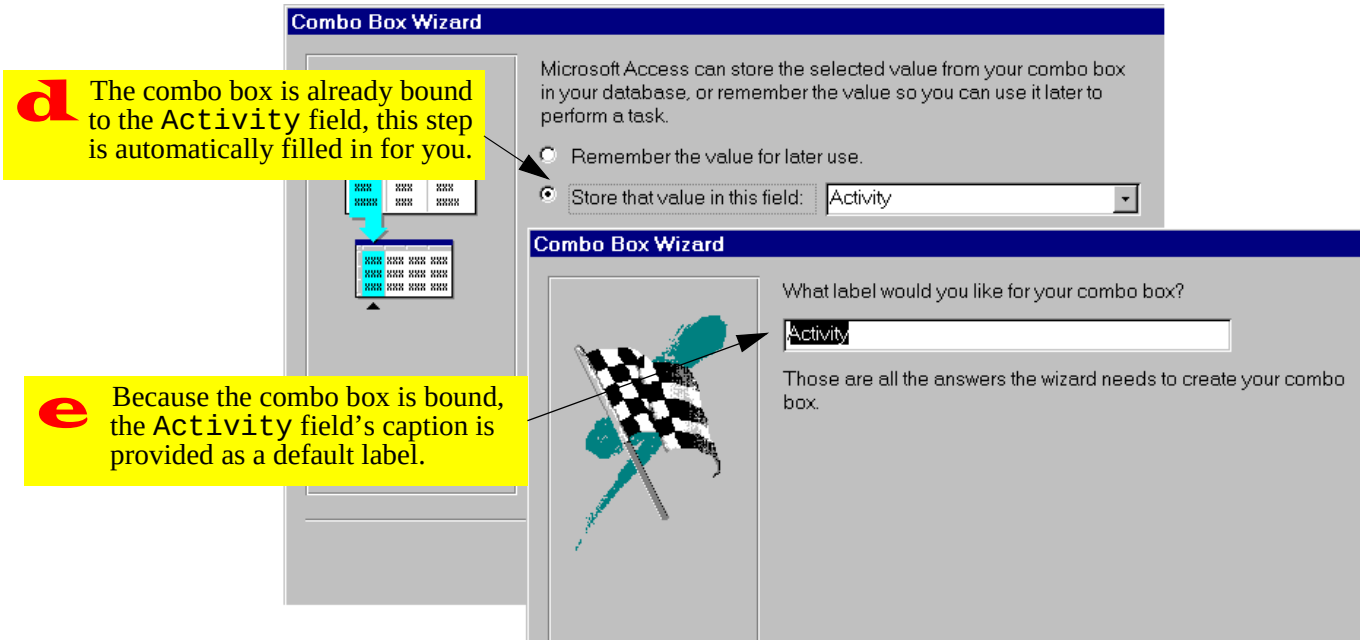
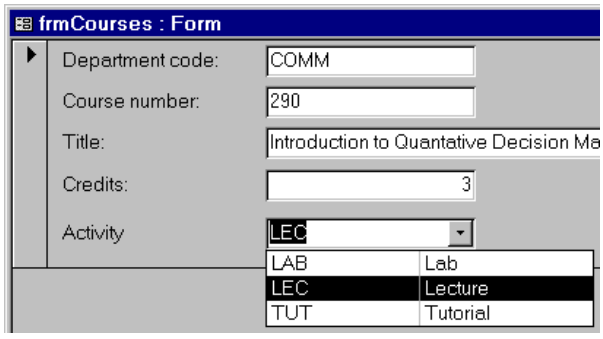


FIGURE 8.9: A combo box that shows two fields from the source table or query.



- Fill in the wizard dialog sheets as in Section 8.3.3 but make the changes shown in Figure 8.10.
- Verify that your combo box resembles Figure 8.9.

8.3.3.2 Hiding the key field

Assume for a moment that you, as a developer, do not want users to even see the three-letter abbreviations and want them to select a course activity value based solely on the `Descript` field.

In such a case, you could include only the `Descript` column in the combo box. However, this would not work because the `Activity` field of the `Courses` table expects a three-letter abbreviation. As such, the combo box would generate an error when it tried to stuff a long description into the relatively short field to which it is bound.

In this section, you will create a combo box identical to that shown in Figure 8.9 except that the key column (`Activity`) will be hidden from view. Despite its invisibility, however, the `Activity` column will still be bound to the `Activity` field of the underlying table and thus the combo box will work as it should.

- Delete the existing combo box and start again using the combo box wizard.

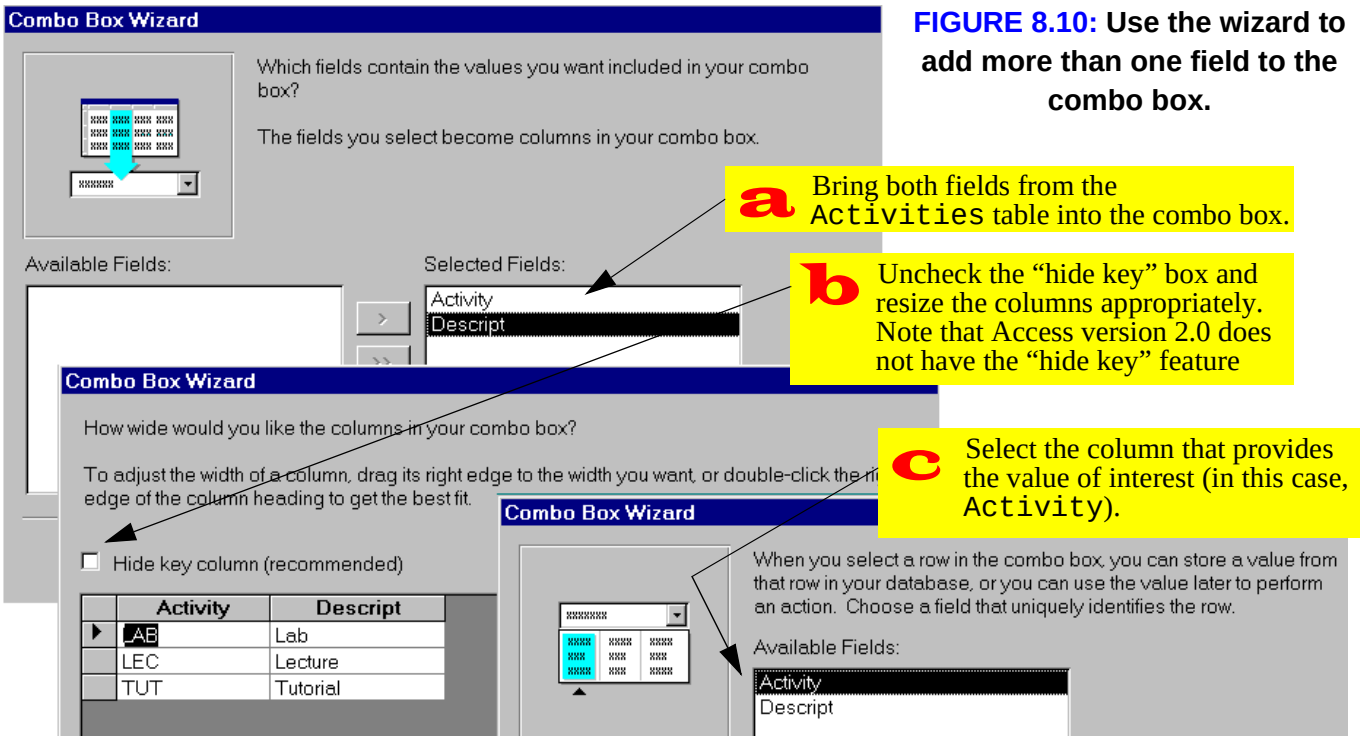
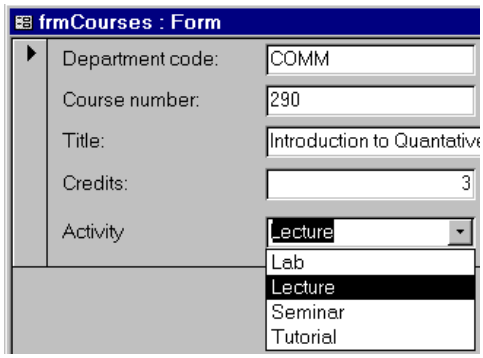


FIGURE 8.10: Use the wizard to add more than one field to the combo box.

- Include both the `Activity` and `Descript` fields in the combo box.
- Resize the Activity column as shown in Figure 8.11. Note that users of version 7.0 can simply leave the "hide key" box checked—the result is the same.
- Ensure that the *Input Mask* property for the combo box (which is inherited from the field's *Input Mask* property) is blank.
- Verify that the resulting combo box resembles that shown in Figure 8.12.

Combo boxes with hidden keys can be confusing. The important thing to remember is that even though the description (e.g., "Lecture") now shows in the combo box, what is really stored in the underlying field is the hidden key (e.g., "LEC").

FIGURE 8.12: A combo box with a hidden key.

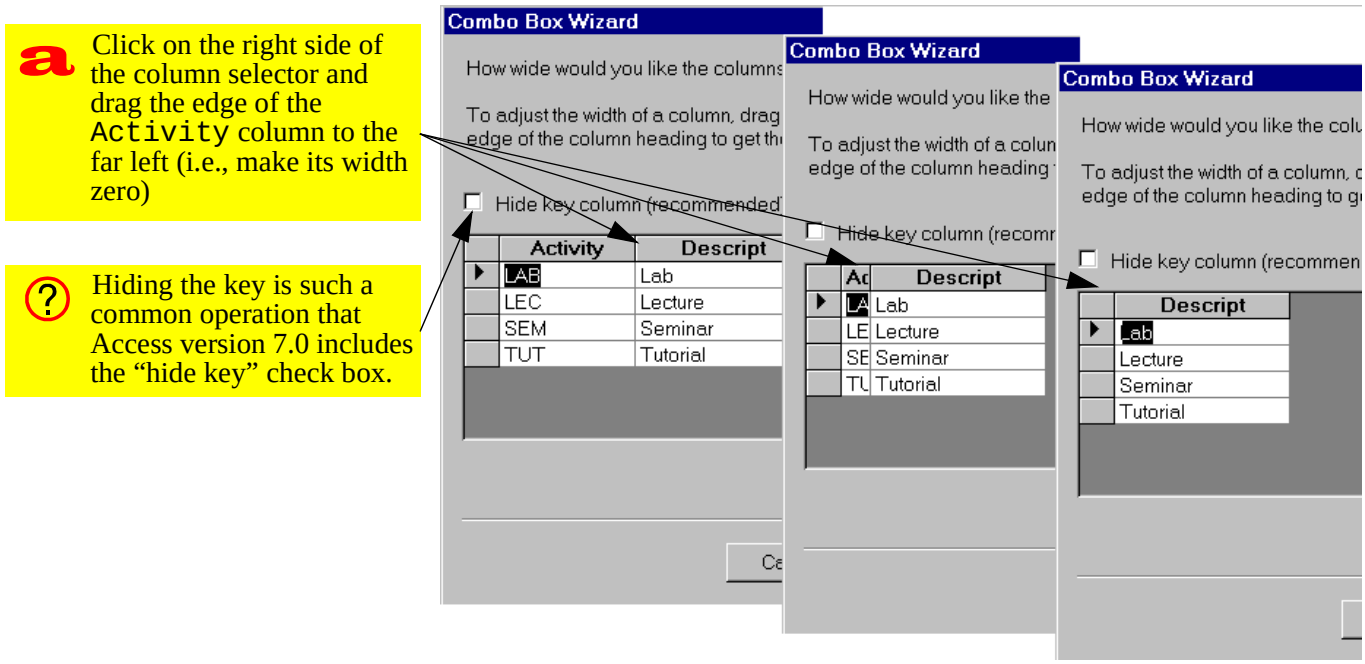


8.3.3.3 Changing the order of items in the combo box

A combo box based on a table shows the records in one of two ways:

1. If the table does not have a primary key, the records are shown in their natural order (that is, in the order they were added to the database).

FIGURE 8.11: Resize the columns to hide the key.



2. If the table does have a primary key, then the records are sorted in ascending order according to the key.

It may be, however, that you want a different order within the rows of the combo box. To achieve this, you can do one of two thing:

1. Create a stand-alone query (in which the sort order is specified) and use this query as the source for the combo box.
2. Modify the "ad hoc" query within the *Row Source* property of the combo box.

If you intend to make several major changes to the basic information in the underlying table (e.g., joins, calculated fields), it is usually better to create a stand-alone query. In this way, the same query can be used by many combo boxes.

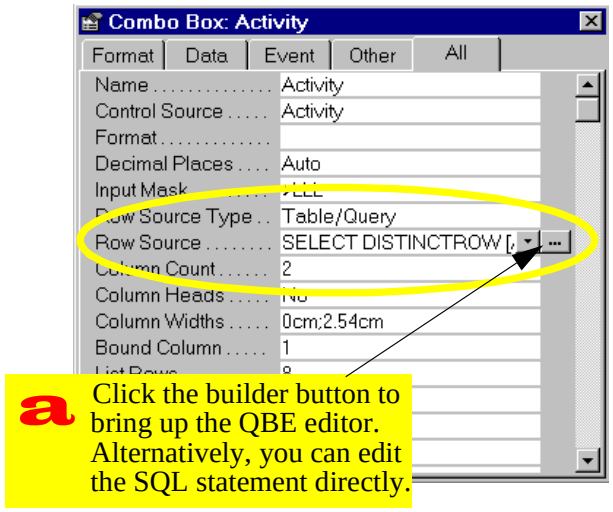
If the changes are quite minor (for instance, sorting the records in a different order), you may prefer to modify the *Row Source* property.

In Section 8.3.2, you set the *Row Source* property to equal a list of values. When the combo box is based on values from a table or a query, however, the *Row Source* is an SQL statement (recall Tutorial 5) rather than a list of values. You can either edit the SQL statement directly or invoke the QBE editor.

In this section, you will order the items in you combo box according to the length of the `Descript` field (this is done merely for illustrative purposes).

- Bring up the property sheet for the `Activity` combo box.
- Put the cursor in the *Row Source* property. As shown in Figure 8.13, a builder button (…) appears.
- Press the builder button to enter the "SQL builder" (i.e., the QBE editor).

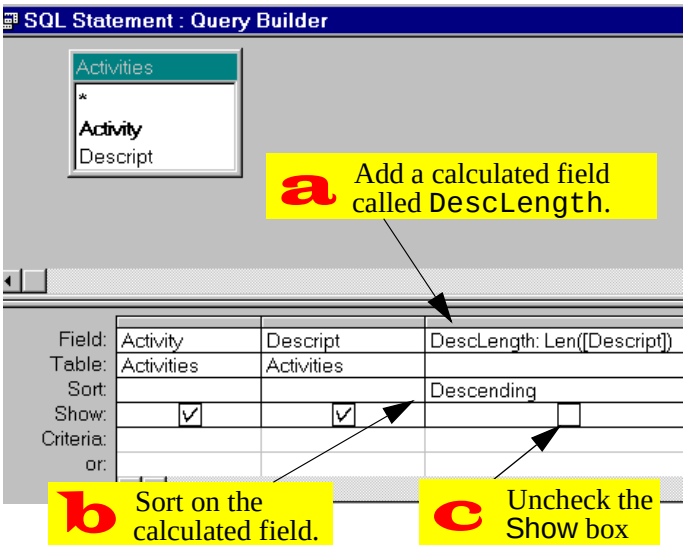
FIGURE 8.13: Invoke the builder for the *Row Source* property.



- Create a calculated field called `DescLength` using the following expression:
  `DescLength: Len([Descript])`
  (`Len()` is a built-in function that returns the length of a string of characters).
- Sort on `DescLength` in descending order.
- Switch to datasheet view to ensure the query is working as it should.
- Ensure the *Show* box for the field is unchecked, as shown in Figure 8.14.
- Instead of saving the query in the normal way, simply close the QBE box using the close button (×).

If you save the query, it will be added to your collection of saved queries (the ones that are displayed in the database window). However, if you simply close the QBE window, the *Row Source* property will be updated and no new database object will be created.

FIGURE 8.14: Use the QBE editor to modify the *Row Source* property.



8.3.4 Changing a form's tab order

A form's **tab order** determines the order in which the objects on a form are visited when the *Tab* or *Enter* (or *Return*) keys are pressed. Access sets the tab order based on the order in which objects are added to the form. As a result, when you delete a text box and replace it with a combo box or some other control, the new control becomes the last item in the tab order regardless of its position on the form.

To illustrate the problem, you are going to create a combo box for the `DeptCode` field.

- Delete the `DeptCode` text box and replace it with a combo box based on the `Departments` table.
- Switch to form view. Notice that the focus starts off in the `CrsNum` field instead of the `DeptCode` field.
- Press tab to move from field to field. Notice that after `DeptCode` is left, the focus returns to the `CrsNum` field of the next record.

- To fix the problem, return to form design mode and select *View > Tab Order* from the main menu.

In Access version 2.0, the menu structure is slightly different. As such, you must select *Edit > Tab Order*.

- Perform the steps in Figure 8.15 to move `DeptCode` to the top of the tab order.

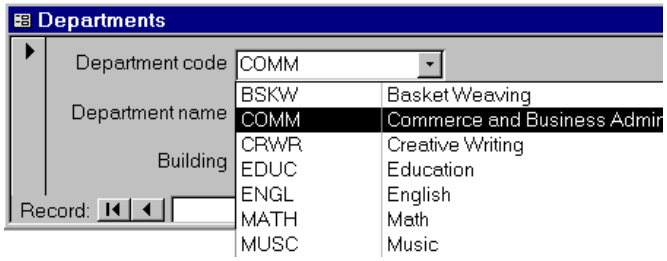
8.4 Discussion

8.4.1 Why you should never use a combo box for a non-concatenated key.

A mistake often made once new users learn how to make combo boxes is to put a combo box on everything. There are certain situations, however, in which the use of a combo box is simply incorrect.

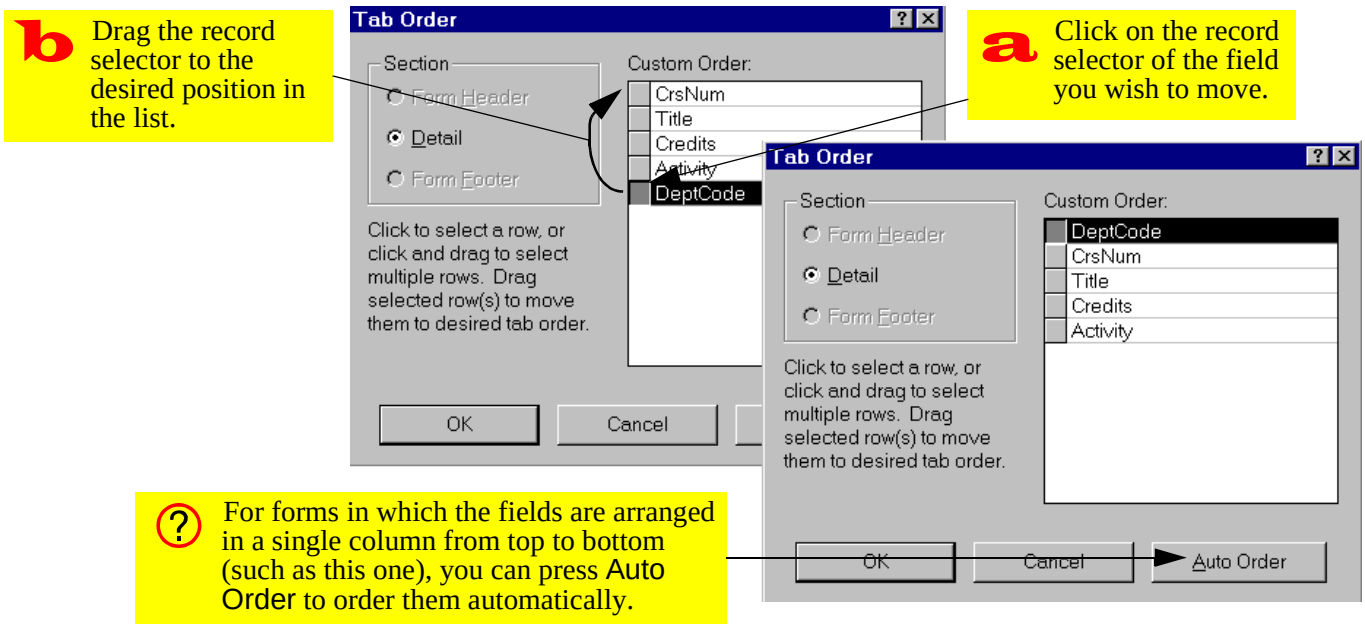
For example, it never makes sense to put a combo box on a non-concatenated primary key. To illustrate this, consider the `Departments` form shown in Figure 8.16. On this form, the `DeptCode` text box has been replaced with a combo box that draws its values from the `Departments` table.

FIGURE 8.16: A combo box bound to a key field.



This combo box appears to work. However, if you think about it, it makes no sense: The form in Figure 8.16 is a window on the `Departments` table. As such, when the `DeptCode` combo box is used,

FIGURE 8.15: Adjust the tab order of fields on a form.



one of two things can occur depending on whether a new record is being created or an existing record is being edited:

1. **A new record is being created** — If a new record is being created (i.e., a new department is being added to the information system), a unique value of `DeptCode` must be created to distinguish the new department from the existing departments. However, the combo box only shows `DeptCode` values of *existing* departments. If the *Limit To List* property is set to `Yes`, then the combo box prevents the user from entering a valid `DeptCode` value.
2. **An existing record is being edited** — It is important to remember that a combo box has no intrinsic search capability. As such, selecting "CPSC" in the `DeptCode` combo box does not result in a jump to the record with "CPSC" as its key value. Rather, selecting "CPSC" from the combo box is identical to typing "CPSC" over whatever is currently in the `DeptCode` field. This causes all sorts of problems; the most obvious of these is that by overwriting an existing value of `DeptCode`, a "duplicate value in index, primary key, or relationship" error is generated (there is already a department with "CPSC" as its `DeptCode`).

Note that a combo box may make sense when the key is concatenated. An example of this is the `DeptCode` combo box you created in Section 8.3.4.

8.4.2 Controls and widgets

Predefined controls are becoming increasingly popular in software development. Although Microsoft includes several predefined controls with Access (such as combo boxes, check boxes, radio buttons, etc.), a large number of more compex or specialized controls are available from Microsoft and other vendors. In addition, you can write your own custom controls using a language like Visual C++ or Visual Basic and use them in many different forms and applications.

An example of a more complex control is the calendar control shown in Figure 8.17. A calendar control can be added to a form to make the entry of dates easier for the user. Microsoft calls such components "ActiveX controls" (formerly known as "OLE controls"). Non-microsoft vendors provide similar components but use different names, such as "widgets".

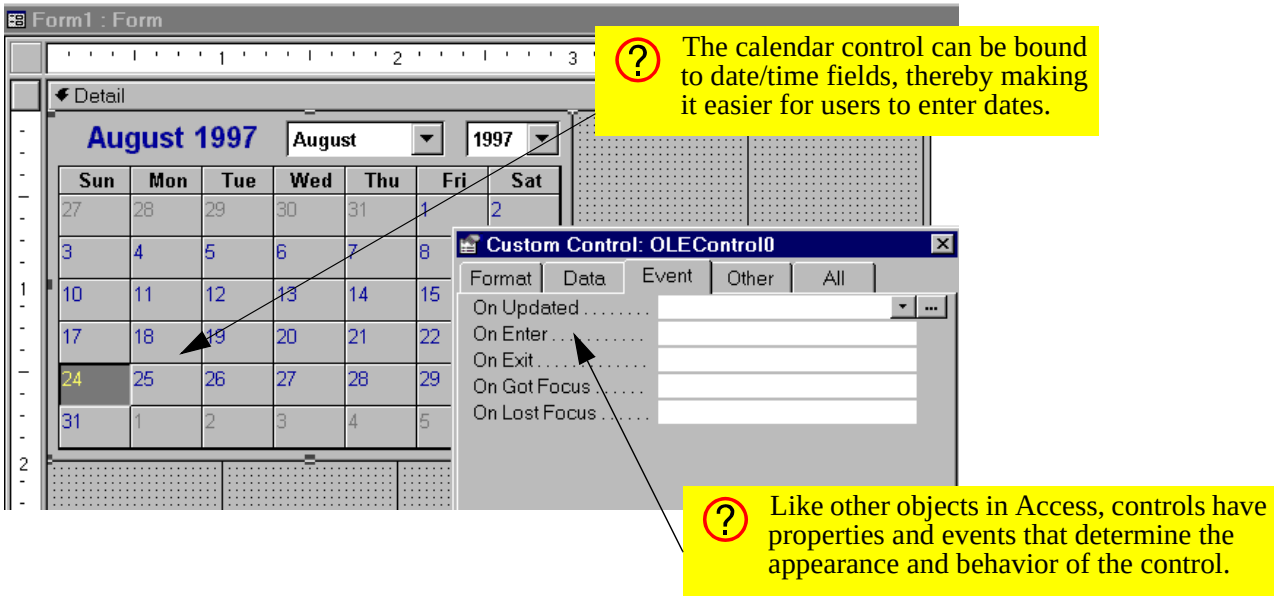
There are two main advantages of using controls. First, they cut down on the time it takes to develop an application since the controls are predefined and pre-tested. Second, they are standardized so that users encounter the same basic behavior in all applications.

8.5 Application to the assignment

There are a number of forms in your assignment that can be greatly enhanced by combo boxes.

- Create a combo box on your order form to allow the user to select customers by name rather than `CustID`. Since your `CustID` value is a counter, it has no significance beyond its use as a primary key. Generally, such keys should be hidden from view.
- Create a combo box in your order details subform to allow the user to select products. Since the `ProductID` values are used by both you and your customers, they have some significance beyond the information system. As such, `ProductID` should be visible in all combo boxes. In addition, the items in the product list should be sorted by `ProductID`. This makes it easier to select a product by typing the first few numbers.
- Create combo boxes on other forms as required.

FIGURE 8.17: A calendar control on a form.



Access Tutorial 9: Advanced Forms

9.1 Introduction: Using calculated controls on forms

It is often useful to show summary information from the subform on the main form. The classic example of this is showing the subtotal from a list of order details on the main order form.

In this tutorial, you are going to explore one means of implementing this feature using calculated controls. A calculated control is an unbound control for which the *Control Source* property is set to an expression that Access can evaluate.

Clearly, calculated controls have a great deal in common with the calculated query fields you created in Section 4.3.3. Although there are no hard-and-fast rules that dictate when to use a one over the other, pushing your calculations to the lowest level (i.e., performing calculations in the query) is usually the best course of action. However, as you will see in the context of subtotals, this is not always possible.

9.2 Learning objectives

- How do I create a calculated text box?
- What is the expression builder? When is it used?
- Where can put an intermediate result of a calculation on a form so that it does not show?

9.3 Tutorial exercises

9.3.1 Creating calculated controls on forms

In this section, you are going to create a simple calculated text box to translate the `Credits` field into a dichotomous text variable [`full year`, `half year`]. Recall that you have already implemented this feature in Section 4.3.3.2 using a calculated query field.

- Perform the steps shown in Figure 9.1 to create an unbound text box on your `fmrCoursesMain` form.
- Set the *Control Source* property of the text box using the syntax:
  `= <expression>`
  In this case, the expression should be an "immediate if" function (see Section 4.3.3.2).

By default, Access interprets text in the *Control Source* property field as the name of a variable (i.e., the name of a field or another control). As such, you must remember to include the equals sign when setting this property.

- Test your form. Note that you are prevented from editing the calculated field. If, however, you change the value of `Credits`, the value of `txtCourseLength` changes accordingly when you leave the `Credits` field.

9.3.2 Showing a total on the main form

In this section, you will create a calculated text box that displays the number of sections associated with each course. The primary motivation for this exercise is to illustrate some of the limitations of calculated controls (as they are implemented in Access) and to provide an opportunity to explore an interesting work-around.

- Create a text box call `txtNumSections` on the main form as shown in Figure 9.2.

The logical next step is to set the *Control Source* of the field to an expression that includes the `Count()` function. However, Access has a limitation in this

FIGURE 9.1: Create an unbound text box on your main form.

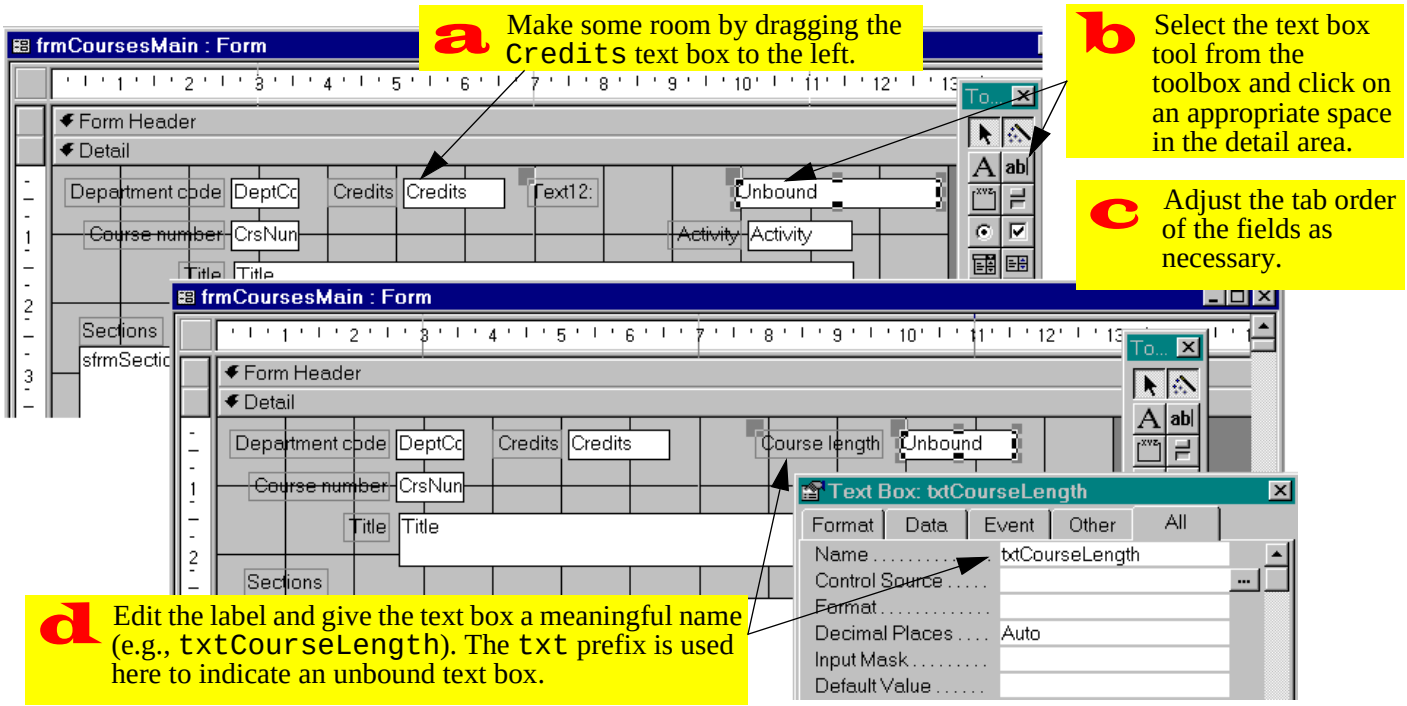
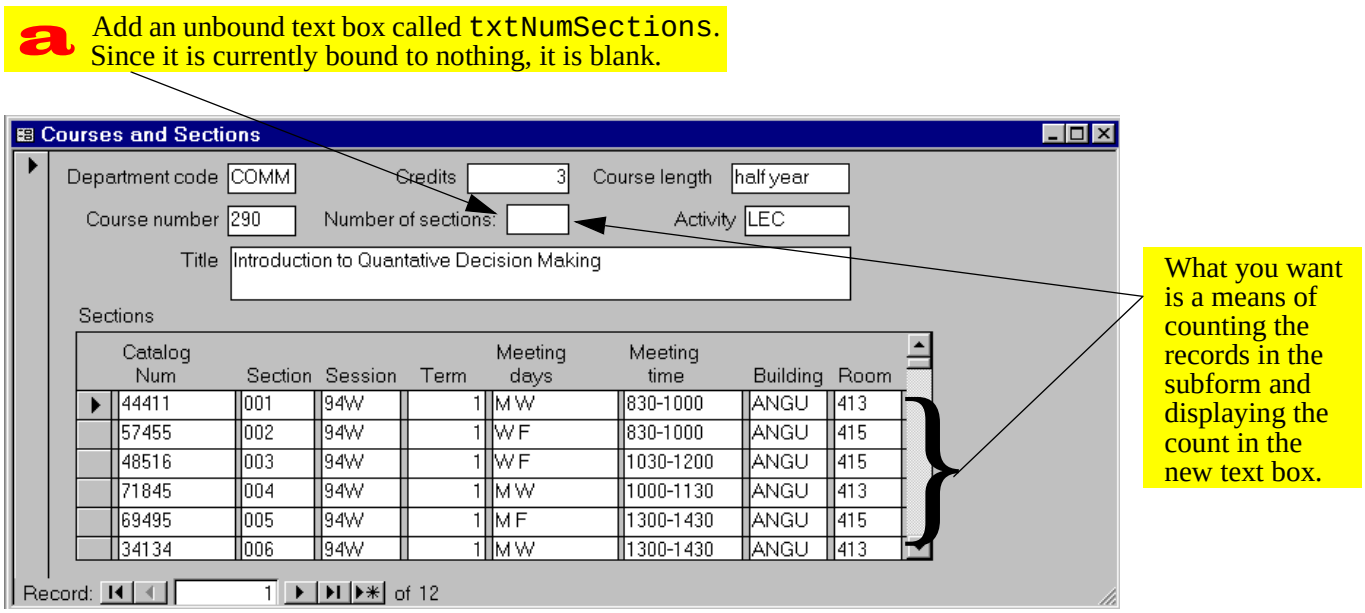


FIGURE 9.2: Create an unbound text box to show the number of sections associated with each course.



regard: you cannot use an **aggregate function** (`Sum()`, `Avg()`, `Count()`, etc.) on a main form that refers to a field in a subform. As a consequence, you have to break the calculation into two steps:

1. use the aggregate function to create a calculated text box on the subform (i.e., a "dummy" field to hold an intermediate result);
2. create a calculated control on the main form that references the dummy text box created in the first step.

It is important that you realize that this procedure does not involve any immutable, fundamental information systems knowledge. Rather, it is merely an example of the type of work-around (hack, kludge, etc.) that is routinely used when using a tool like Access to create a custom application.

9.3.2.1 Calculating the aggregate function on the subform

- Create an unbound text box on the subform as shown in Figure 9.3.
- Save the subform but do not close it.
- Return to the main form and set the *Control Source* of `txtNumSections` to equal the value of `txtNumSectionsOnSub`. Since the naming conventions for objects on forms and subforms can be tricky, use the **expression builder** (as shown in Figure 9.4) to build the name for you.

The expression builder organizes all the elements of the database environment into a hierarchical structure. You build an expression by "drilling down" to the element you need and double-clicking to copy its name into the text area.

The expression builder takes some practice. One problem is that it is easy to double-click

FIGURE 9.3: Perform the count on the subform.

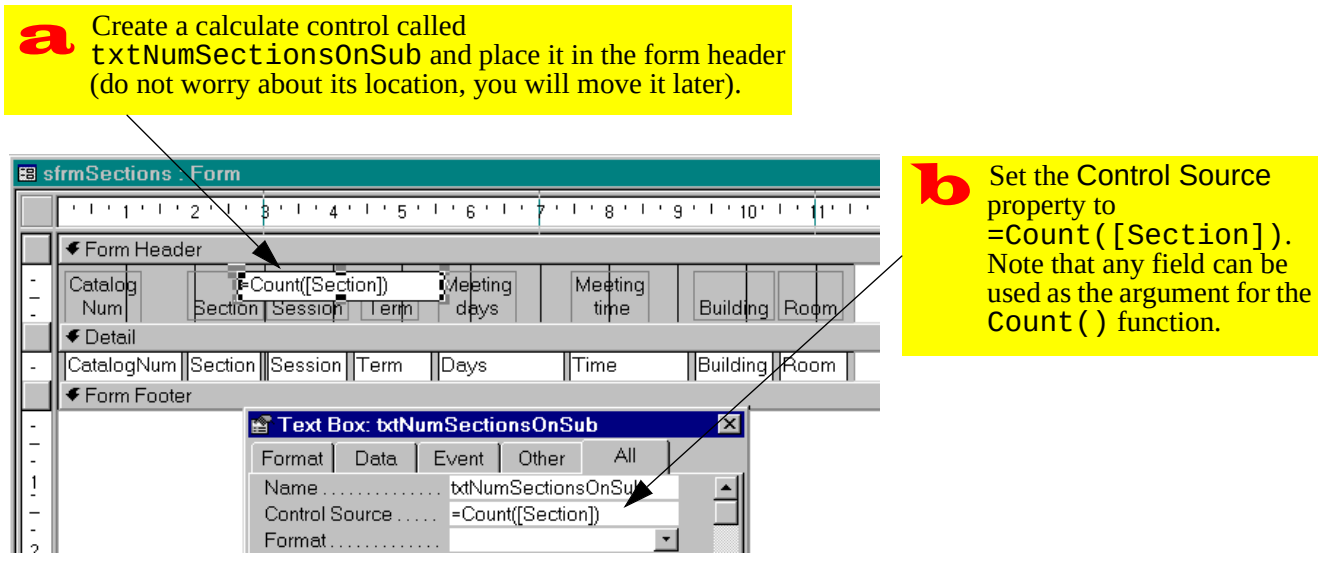
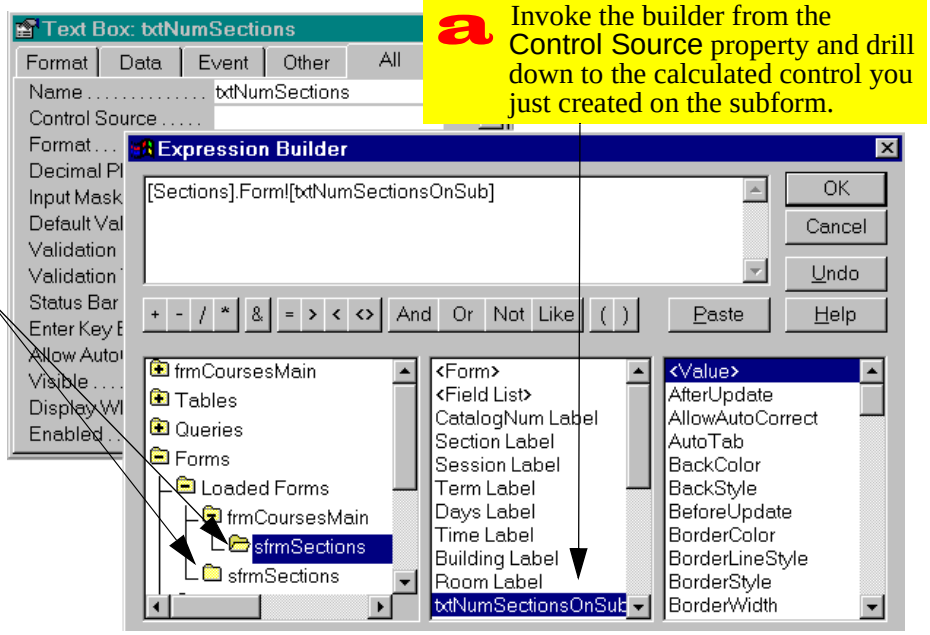


FIGURE 9.4: Use the builder to drill down to the calculated control on the subform.



Note that when the main form and the subform are both open, the subform appears twice in the builder: once as a "stand-alone" form (under "Loaded Forms") and once *as a component* of the main form (press the + sign on the `frmCoursesMain` folder). You want to use the latter (you will never access the subform in stand-alone mode).

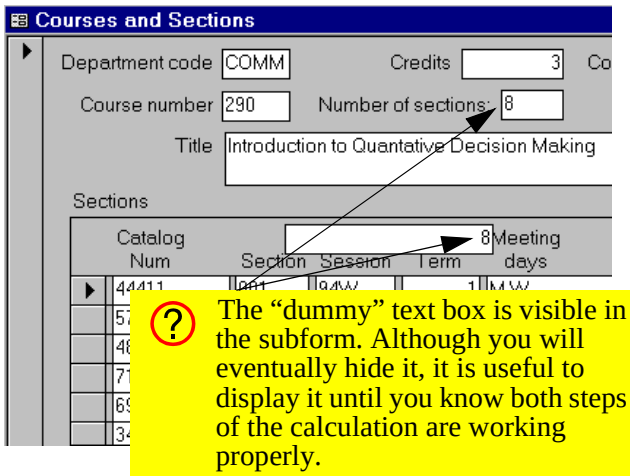
Invoke the builder from the Control Source property and drill down to the calculated control you just created on the subform.

on the wrong thing. Another problem is that Access attempts to guide you by inserting «Expr» place-holders all over the place. The solution to both problems is to click on the text window and make liberal use of the *Delete* key.

The point made about "stand-alone" and "component" subforms in Figure 9.4 is extremely important. The reason you use the `sfrm` prefix is so you know that the form is designed to be a component of another form. If you select the stand-alone version the form in the builder, the name created by the builder will be incorrect and an error will result.

- Close the subform (in version 7.0 and 8.0, the main form and subform cannot be open at the same time).
- Test the form. The value of `txtNumSections` and `txtNumSectionsOnSub` should be identical, as shown in Figure 9.5.

FIGURE 9.5: The number of sections on the main form.



The "dummy" text box is visible in the subform. Although you will eventually hide it, it is useful to display it until you know both steps of the calculation are working properly.

9.3.2.2 Hiding the text box on the subform

The obvious problem in Figure 9.5 is that the dummy text box shows on the subform. There are at least two ways to get around this: one is to set the *Visible* property of the text box to `No`; a slightly more elegant approach is to use the **page header** or **page footer** to hide the text box.

The page header and footer are areas on the form that only show when the form is printed. Since you will never print a form (reports are used for printed material), these areas can be used to hide intermediate results, etc.

- In design mode, select *View > Page Header/Footer* from the menu.

In version 2.0, the menu structure is slightly different. As such, you must select *Format > Page Header/Footer*.

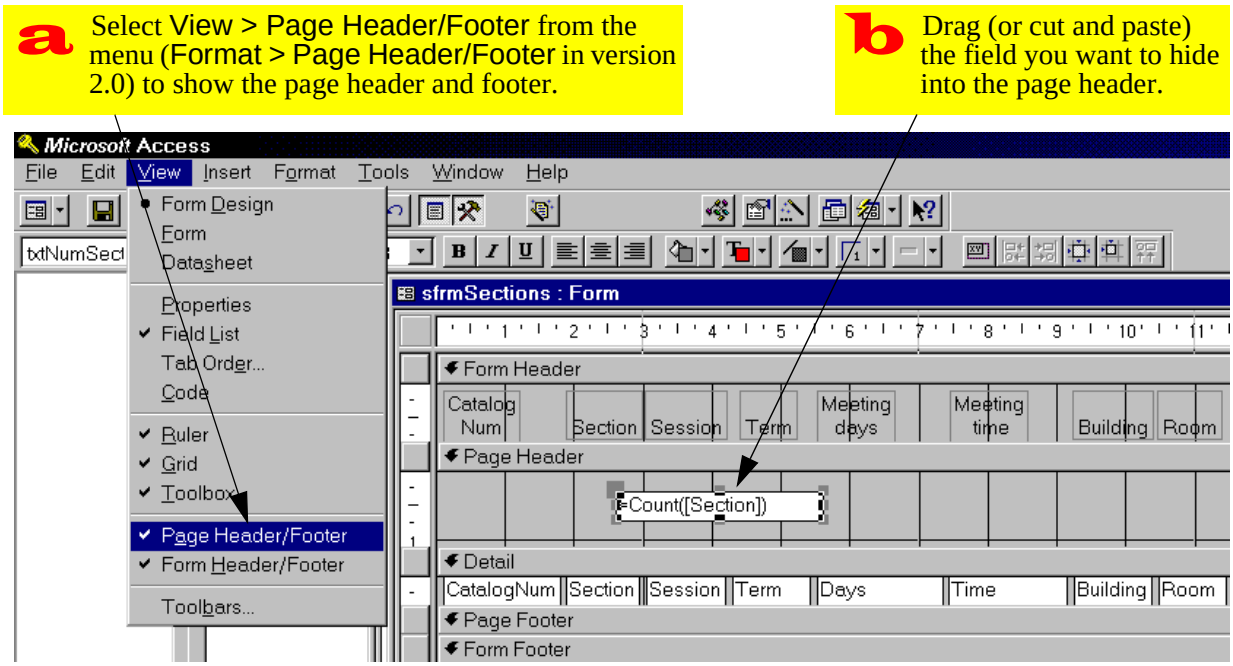
- Drag (or cut and paste) `txtNumSectionsOnSub` from the form header to the page header, as shown in Figure 9.6.
- Test the result.

9.4 Discussion

In Section 4.3.3.2 and Section 9.3.1, you accomplished the same thing (showing `half year` or `full year`) using different techniques. The advantage of implementing this as a calculated query field is that you can use this field repeatedly in other forms. On the other hand, if you do the transformation on the form, you have to repeat the calculation on every form that requires the calculated field.

In the case of the aggregate function, the situation is slightly different. Although you can use the **totals** feature of QBE (see on-line help) to count the number of sections for a particular course within a query, the resulting recordset is non-updatable (and hence

FIGURE 9.6: Hide the intermediate result in the page header.



not much use for editing course names, etc.). As a result, you are forced to do the calculation on the form rather than in the query.

9.5 Application to the assignment

To show the subtotal, tax, and grand total on your order form, you use the same techniques illustrated here. The only difference is that you use the `Sum()` function instead of the `Count()` function to get the subtotal for the order.

- Create a dummy field on your `OrderDetails` subform to calculate the subtotal for the order.
- Calculate the tax (G.S.T. only for wholesale) and grand total on the main form (traditionally, this information is located near the bottom of the form—but *not* in the form footer).

Access Tutorial 10: Parameter Queries

The last few tutorials have been primarily concerned with interface issues. In the remaining tutorials, the focus shifts to transaction processing.

10.1 Introduction: Dynamic queries using parameters

A **parameter query** is a query in which the criteria for selecting records are determined when the query is executed rather than when the query is designed.

For example, recall the select query shown in Figure 4.6. In this query, the results set is limited to records that satisfy the criterion `DeptCode = "COMM"`. If you wanted a different set of results, you would have to edit the query (e.g., change the criterion to `"CPSC"`) and rerun the query.

However, if a variable (parameter) is used for the criterion, Access will prompt the user for the value of the variable before executing the query. The net result is that parameters can be used to create extremely flexible queries.

When the concepts from this tutorial are combined with action queries (Tutorial 11) and triggers (Tutorial 13), you will have a the skills required to create a simple transaction processing system without writing a line of programming code.

10.2 Learning objectives

- ❒ What is a parameter query? How do I create one?
- ❒ How do I prompt the user to enter parameter values?
- ❒ How do I create a query whose results depend on a value on a form?

10.3 Tutorial exercises

10.3.1 Simple parameter queries

- If you do not already have a `qryCourses` query like the one shown in Figure 4.6, create one now and save it under the name `pqryCourses`.
- Replace the literal string in the criteria row (`"COMM"`) with a variable (`[X]`).

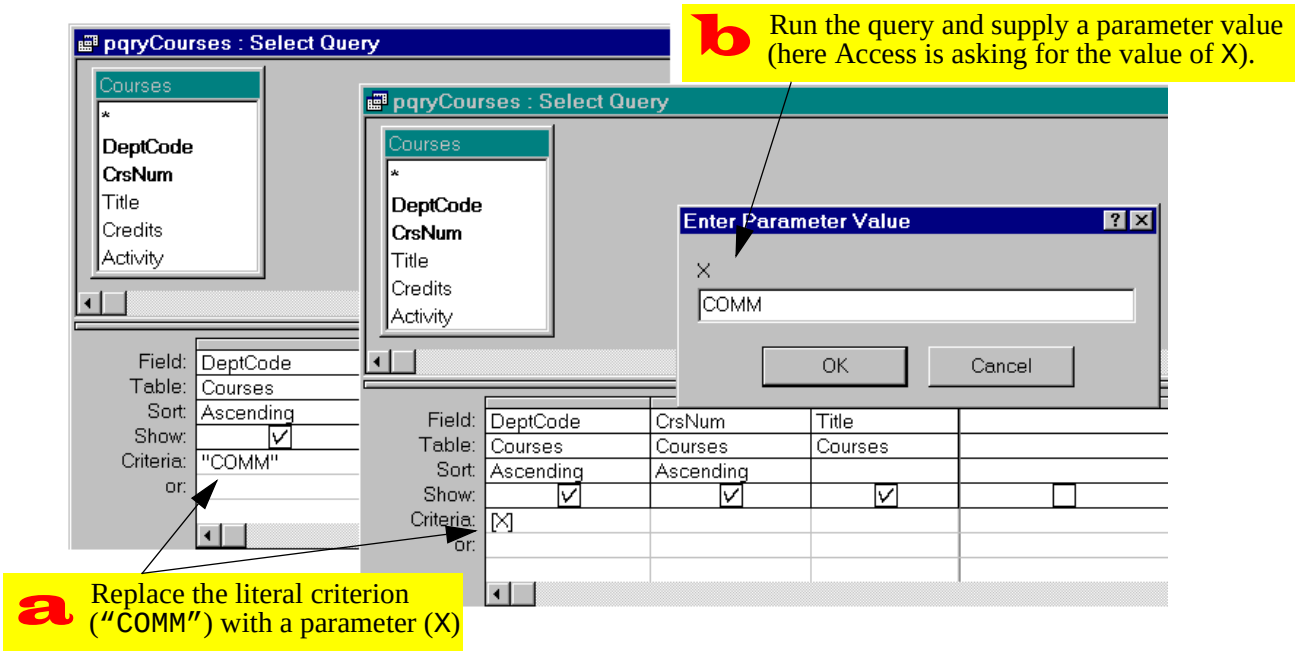
By default, Access expects criteria to be literal strings of text. As a result, it automatically adds quotation marks to text entered in the criteria row. To get around this, place your parameter names inside of square brackets.

- Execute the query as shown in Figure 10.1.

When Access encounters a variable (i.e., something that is not a literal string) during execution, it attempts to bind the variable to some value. To do this, it performs the following tests:

1. First, Access checks whether the variable is the name of a field or a calculated field in the query. If it is, the variable is bound to the current value of the field. For example, if the parameter is named `[DeptCode]`, Access replaces it with the current value of the `DeptCode` field. Since `X` is not the name of a field or a calculated field in this particular query, this test fails.
2. Second, Access attempts to resolve the parameter as a reference to something within the current environment (e.g., the value on an open form). Since there is nothing called `X` in the current environment, this test fails.
3. As a last resort, Access asks the user for the value of the parameter via the "Enter Parameter Value" dialog box.

FIGURE 10.1: Convert a select query into a parameter query.



Note that the spelling mistakes discussed in Section 4.3.4 are processed by Access as parameters.

10.3.2 Using parameters to generate prompts

Since the name of the parameter can be anything (as long as it is enclosed in square brackets), you can exploit this feature to create quick and easy dialog boxes.

- Change the name of your `DeptCode` parameter from `[X]` to `[Courses for which department?]`.
- Run the query, as shown in Figure 10.2.

10.3.3 Values on forms as parameters

A common requirement is to use the value on a form to influence the outcome of a query. For instance, if the user is viewing information about departments, it may be useful to be able to generate a list of courses offered by the department currently being viewed. Although you could use a creatively-named parameter to invoke the "Enter Parameter Value" dialog, this requires the user to type in the value of `DeptCode`.

A more elegant approach is to have Access pull the value of a parameter directly from the open form. This exploits the second step in the operation of a parameter query (Access will attempt to resolve a parameter with the value of an object within the current environment). The basic idea is shown in Figure 10.3.

The key to making this work is to provide a parameter name that correctly references the form object in which you are interested. In order to avoid having to remember the complex naming syntax for objects on forms, you can invoke the expression builder to select the correct name from the hierarchy of database objects.

FIGURE 10.2: Select a parameter name that generates a useful prompt.

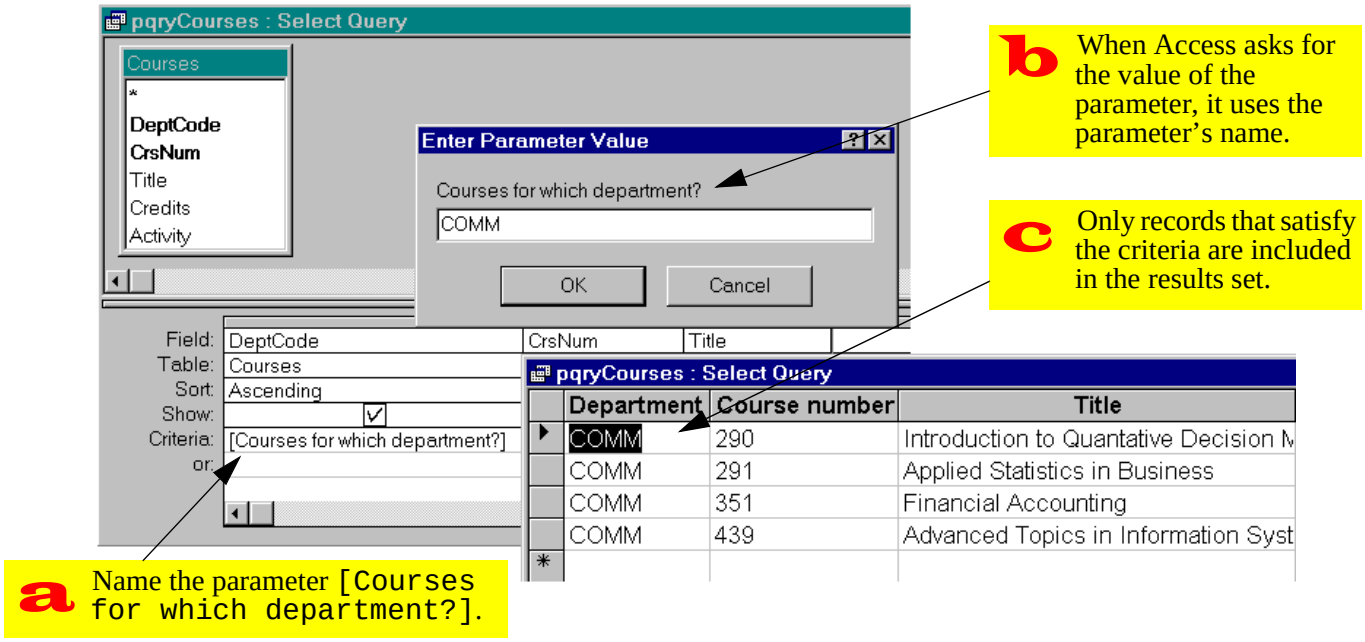
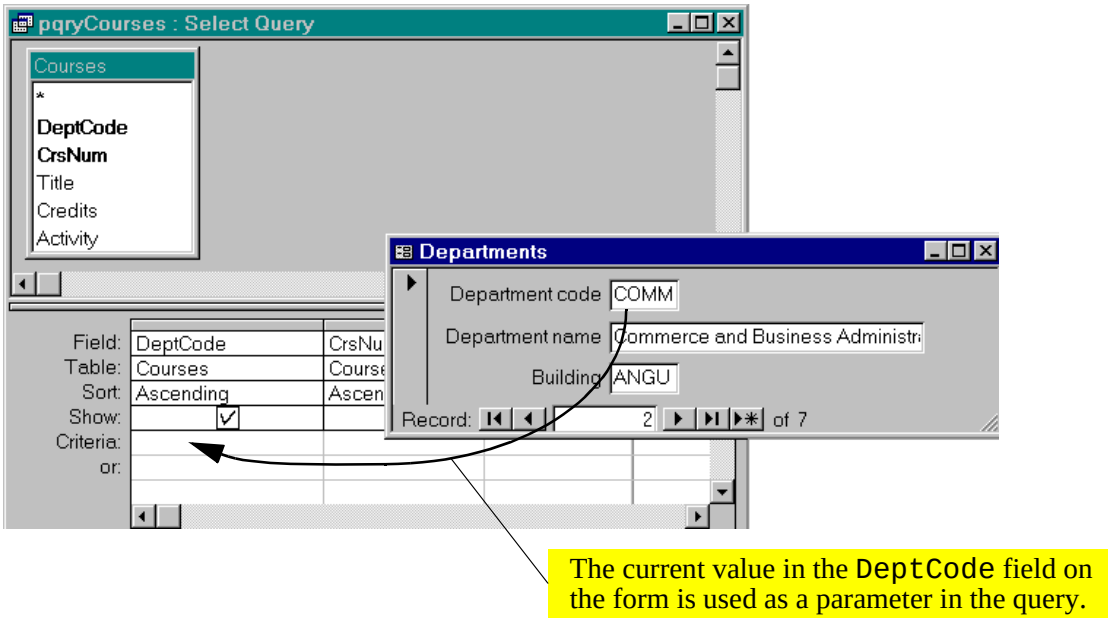


FIGURE 10.3: Using the value on an open form as a parameter in a query.



- Create a very simple form based on the `Departments` table and save it as `frmDepartments`.
- Leave the form open (in form view or design mode, it does not matter).
- Open `pqryCourses` in design mode, place the cursor in the criteria row of the `DeptCode` field, and invoke the expression builder as shown in Figure 10.4.
- Perform the steps shown in Figure 10.5 to create a parameter that references the `DeptCode` field on the `frmDepartments` form.
- Run the query. The results set should correspond to the department showing in the `frmDepartments` form.
- Move to a new record on the form. Notice that you have to requery the form (*Shift-F9*) in order for the new parameter value to be used (see Figure 10.6).

Although the naming syntax of objects in Access is tricky, it is not impossible to comprehend. For example, the name `Forms![frmDepartments]![DeptCode]` consists of the following elements: `Forms` refers to a collection of Form objects; `[frmDepartments]` is a specific instance of a Form object in the Forms collection; `[DeptCode]` is a Control belonging to the form. See Tutorial 14 for more information on the hierarchy of objects used by Access.

10.4 Application to the assignment

You will use parameter queries as the basis for several **action queries** (see Tutorial 11) that process transactions against master tables. For now, simply create the parameter queries that take their criteria values from forms you have already created.

FIGURE 10.4: Invoke the builder to build a parameter.

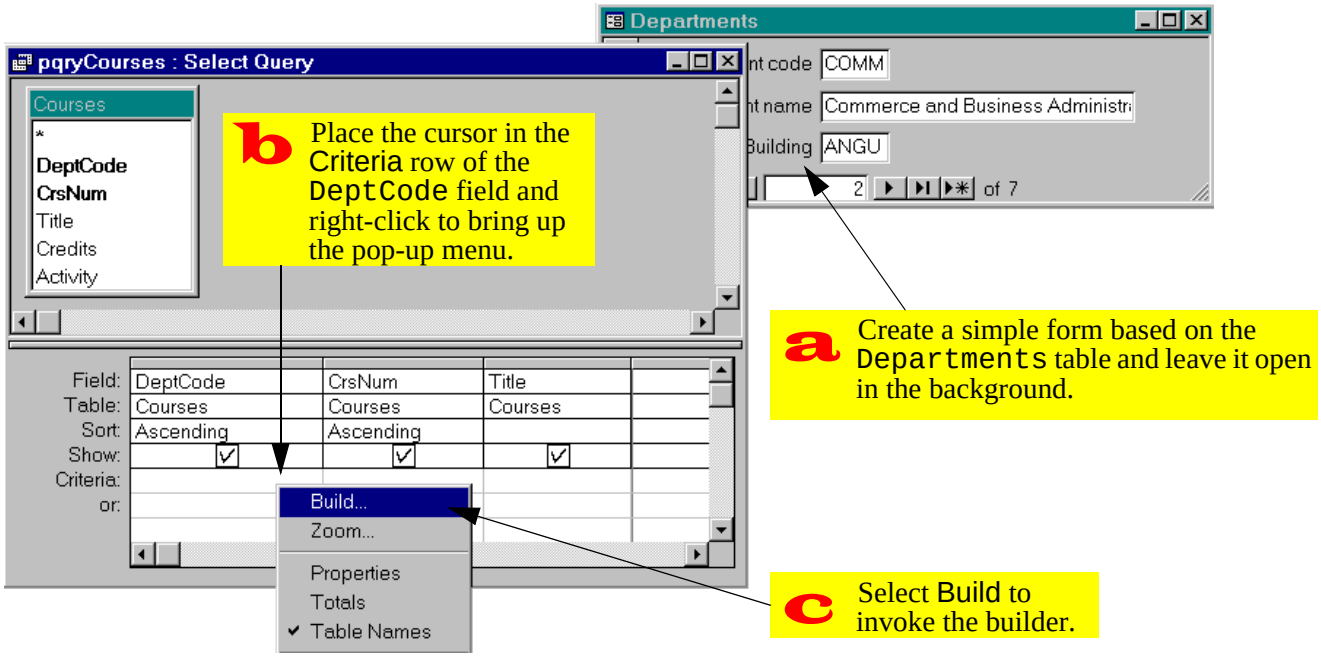


FIGURE 10.5: Use the builder to select the name of the object you want to use as a parameter.

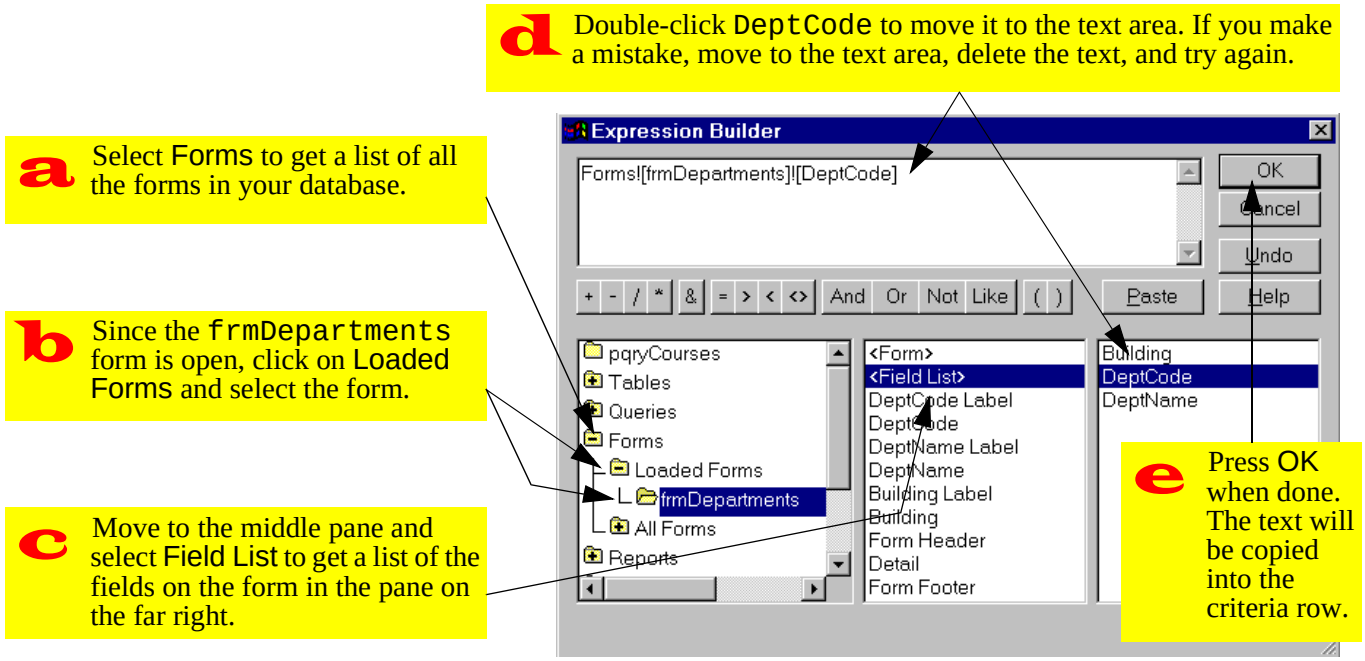
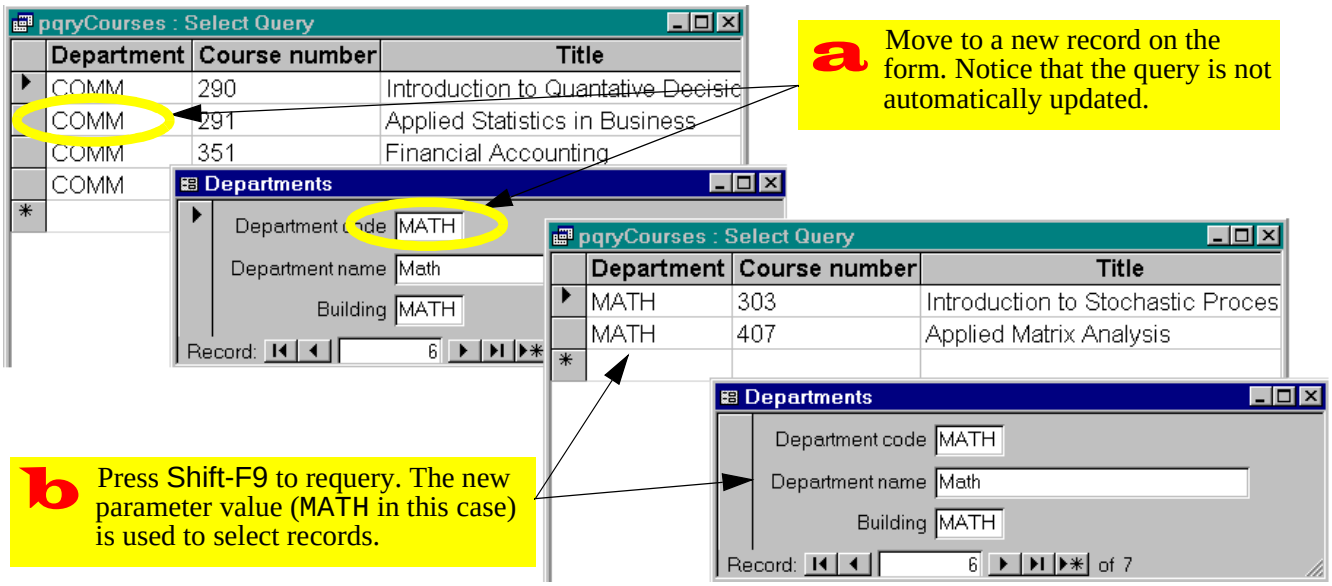


FIGURE 10.6: Requery the results set to reflect changes on the form.



- Create a parameter query to show all the order details for a particular order.
- Create a second parameter query to show all the shipment details for a particular shipment.

Each order may result in a number of changes being made to the `BackOrders` table. For some items in the order, more product is ordered than is actually shipped (i.e., a backorder is created). For other items, more product is shipped than is ordered (i.e., a backorder is filled).

In Tutorial 15, you are supplied with a "shortcut" Visual Basic procedure that makes the changes to the `BackOrders` table for you. However, the shortcut procedure requires a query that lists the changes that must be made to the `BackOrders` table for a particular order. The requirements for this query are the following:

- The name of the query is `pqryItemsToBackOrder`
- It shows the change (positive or negative but not zero) in backorders for each item in a particular order.
- The query consist of three fields: `OrderID`, `ProductID` and a calculated field `Qty` (i.e., the change in the back order for a particular product).
- The name of the parameter is in this query is simply `[pOrderID]`. Since the value of this parameter will be set by the Visual Basic shortcut before the query is run, there is no need to set it to a value on a form.

Since the query is accessed by a program, the name of the query and all the fields must be *exactly as described above*. In other words, you are given a precise specification for a database object that fills a role in a process designed and implemented by someone else. You will not understand how the query fits in until Tutorial 15.

Access Tutorial 11: Action Queries

11.1 Introduction: Queries that change data

11.1.1 What is an action query?

All of the queries that you have created to this point have been variations of “select” queries. Select queries are used to display data but do not actually change the data in any way.

Action queries, in contrast, are used to change the data in existing tables or make new tables based on the query's results set. The primary advantage of action queries is that they allow you to modify a large number of records without having to write Visual Basic programs.

Access provides four different types of action queries:

1. **Make table** — creates a new table based on the results set of the query;
2. **Append** — similar to a make-table query, except that the results set of the query is appended to an existing table;
3. **Update** — allows the values of one or more fields in the result set to be modified; and,
4. **Delete** — deletes all the records in the results set from the underlying table.

Since the operation of all four types of action queries is similar, we will focus on update queries in this tutorial.

11.1.2 Why use action queries?

To motivate the examples in the first part of this tutorial, we are going to assume that the number of credits allocated to courses in certain departments need to be changed. For example, assume that you need to increase the number of credits for courses in the Commerce department by 1.5 times their current values. There are at least four different ways of accomplishing this task:

1. Create a calculated field called `NewCredits` that multiplies the value of `Credits` by 1.5 — The query containing the calculated field can be used in place of the `Courses` table whenever credit information is required. Of course, the values stored in the `Courses` table are still the old values. Although there might be some advantages to keeping the old values, it may cause confusion about which values to use. In addition, the use of a calculated field creates a computational load that becomes larger as the number of courses increases.
2. Go through the `Courses` table record by record and manually change all the values — This approach is tedious and error prone. Furthermore, it is simply impractical if the number of courses is large.
3. Write a Visual Basic program to automate Step 2. This is a good approach; however, it clearly requires the ability to write Visual Basic programs.
4. Create an **update query** that (a) selects only those courses that require modification and (b) replaces the value of `Credits` with `Credits * 1.5`. — This approach is computationally efficient and allows you to work with the QBE editor rather than a programming language.

11.2 Learning objectives

- ❒ What is an action query? Why would I want to use one?
- ❒ How do I make a backup copy of one of my tables?
- ❒ How to I undo (rollback) an action query once I have executed it?

Last update: 25-Aug-1997

- How do I update only certain records in a table?
- How do I create a button on a form? How do I make an action query execute when the button is pressed?

11.3 Tutorial exercises

11.3.1 Using a make-table query to create a backup

Since action queries permanently modify the data in tables, it is a good idea to create a backup of the table in question before running the query. An easy way to do this is to use a make-table query.

- Create a select query based on the `Courses` table and save it as `qryCoursesBackup`.
- Project the asterisk (*) into the query definition so that all the fields are included in the results set.
- While still in query design mode, select *Query > Make Table* from the main menu and provide a name for the target table (e.g., `CoursesBackup`) as shown in Figure 11.1.
- Select *Query > Run* from the main menu to execute the action query, as shown in Figure 11.2.

Action queries do not execute until you explicitly run them. Switching to datasheet mode only provides a preview of the results set.

- Save the query. If you switch to the database window, you will notice that the new make-table query has a different icon than the select queries.

11.3.2 Using an update query to rollback changes

Having a backup table is not much use without a means of using it to restore the data in your original table. In this section, you will use an update query to

FIGURE 11.1: Use a make-table query to back up and existing table

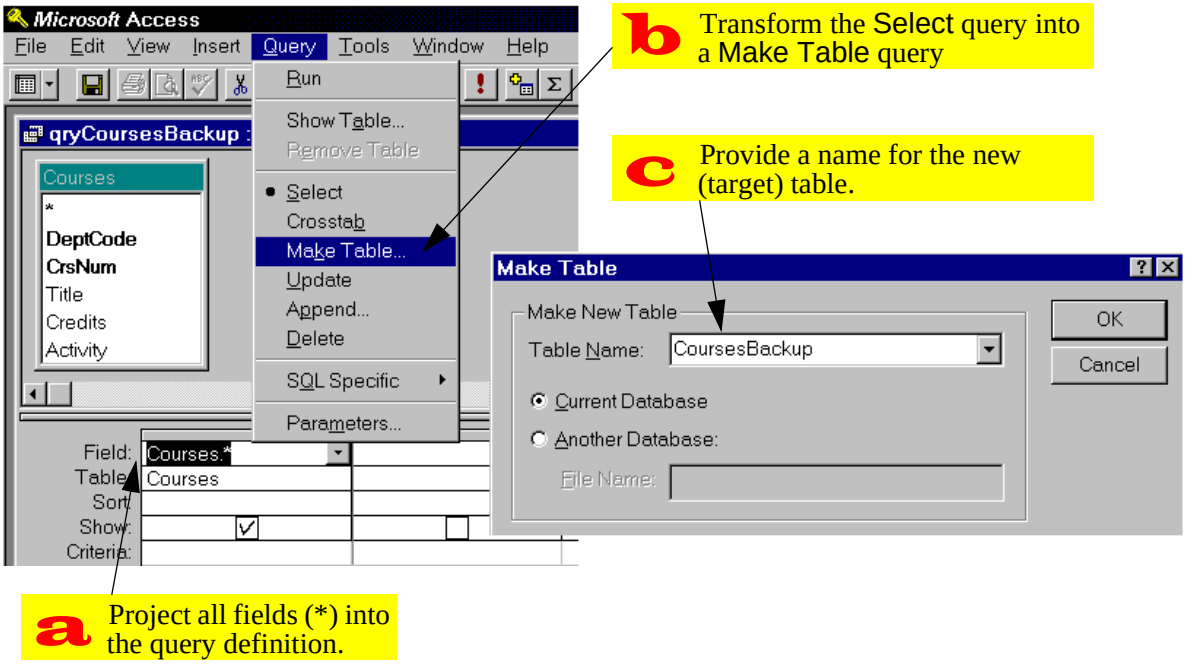
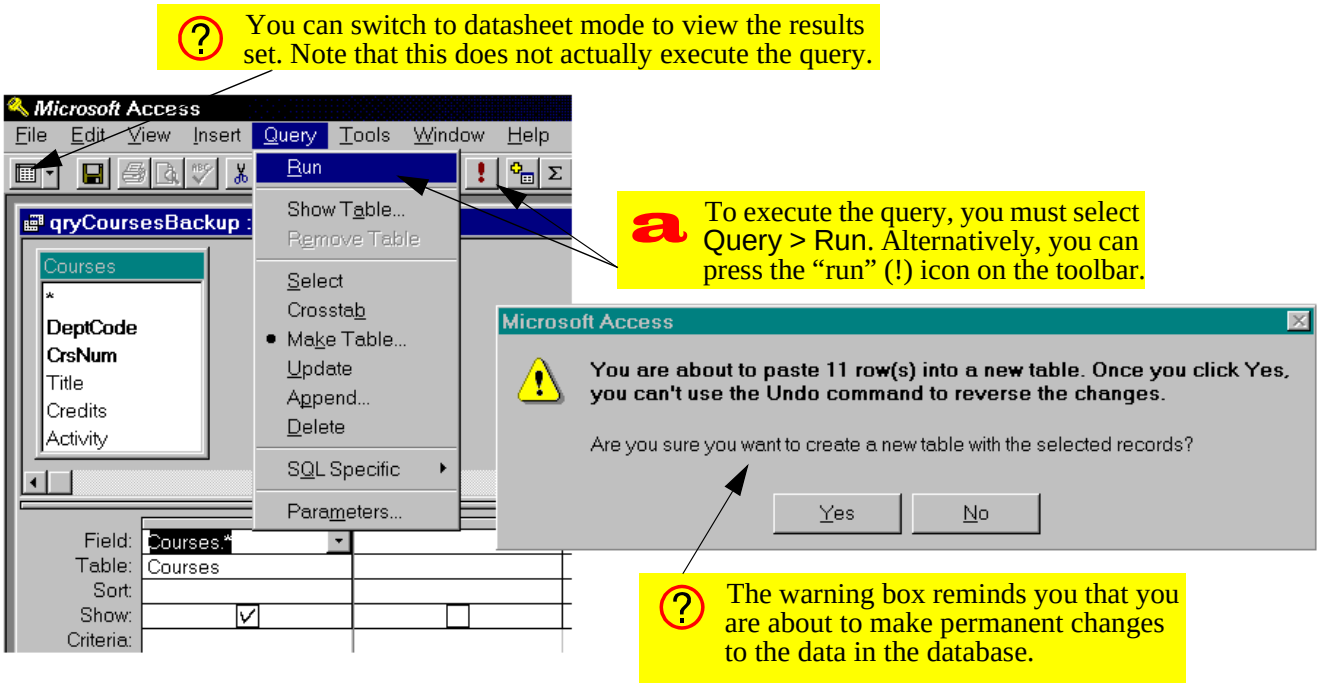


FIGURE 11.2: Run the make-table query.



replace some of the values in your `Courses` table with values from your `CoursesBackup` table.

- Create a new query based on the `Courses` and `CoursesBackup` tables.
- Since no relationship exists between these tables, create an *ad hoc* relationship within the query as shown in Figure 11.3.
- Select *Query > Update* from the main menu. Note that this results in the addition of an *Update To* row in the query definition grid.
- Project `Credits` into the query definition and fill in the *Update To* row as shown in Figure 11.4.
- Save the query as `qryRollbackCredits`.

Now is a good point to stop and interpret what you have done so far:

1. By creating a relationship between the `Courses` table and its backup, you are joining together the records from both tables that satisfy the condi-

FIGURE 11.3: Create an *ad hoc* relationship between the table and its backup.

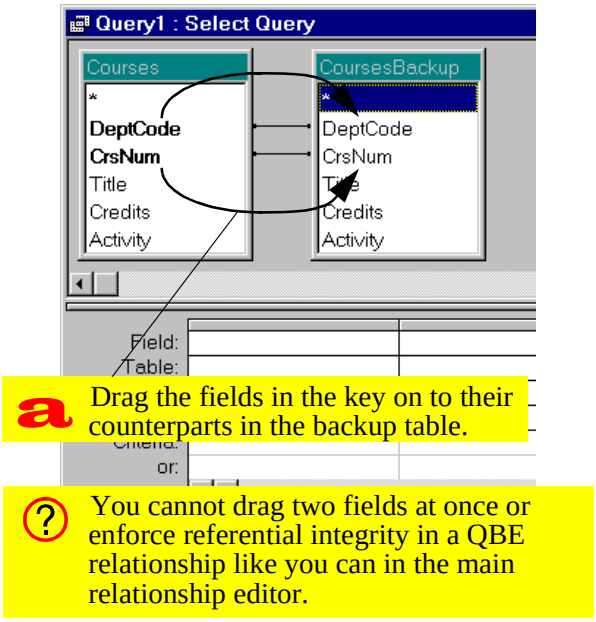
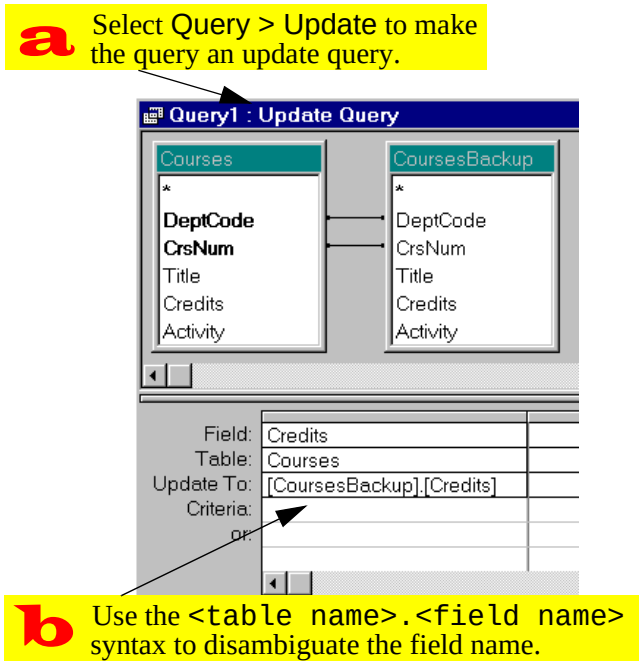


FIGURE 11.4: Fill in the *Update To* field.



tion:

```
Courses.DeptCode =
 CoursesBackup.DeptCode AND
 Courses.CrsNum =
 CoursesBackup.CrsNum.
```

2. By projecting `Courses.Credits` into the query, you are making it the target for the update. In other words, the values in `Courses.Credits` are going to be modified by the update action.
3. By setting the *Update To* field to `Courses-Backup.Credits`, you are telling Access to replace the contents of `Courses.Credits` with the contents of `CoursesBackup.Credits`.

Whenever this query is run, it will replace whatever is in the `Credits` field of all the records in the `Courses` table with values from the backup. You will use this query to "rollback" updates made later on.

11.3.3 Using an update query to make selective changes

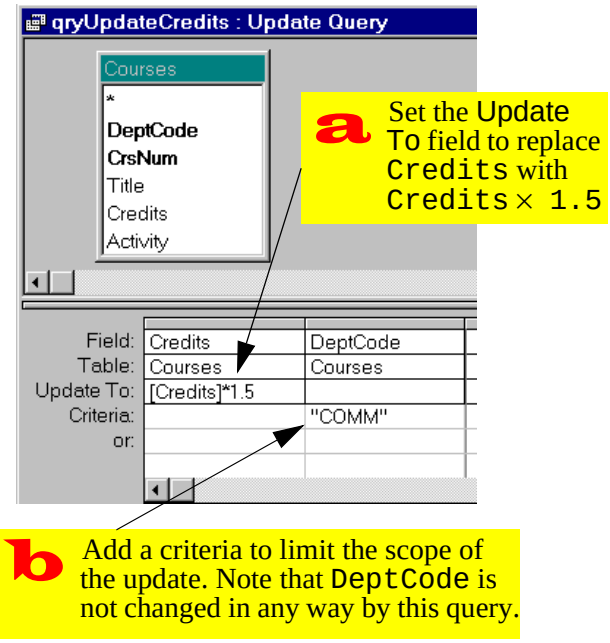
Now that you have an infrastructure for undoing any errors, you can continue with the task of updating credits for the Commerce department.

- Create an update query based on the `Courses` table and save it as `qryUpdateCredits`.
- Set the *Update To* field to `[Courses]*1.5`. Note that if you do not include the square brackets, Access will interpret `Courses` as a literal string rather than a field name.

? Since this particular query only contains one table, the `<table name>.<field name>` syntax is not required for specifying the *Update To* expression.

- Since you only want to apply the change to Commerce courses, enter a criterion for the `Dept-Code` field, as shown in Figure 11.5.

FIGURE 11.5: Create an update query that updates a subset of the records.



- Run the query and verify that update has been performed successfully.

11.3.4 Rolling back the changes

While testing the `qryUpdateCredits` query, your exuberance may have led you to execute it more than once. To return the `Courses` table to its state before any updates, all you need to do it run your rollback query.

- Run `qryRollback` credits by double-clicking its icon in the database window.

Once an action query is created, it has more in common with subroutines written in Visual Basic than standard select queries. As such, it is best to think of action queries in terms of procedures to be executed rather than virtual tables or views. Double-clicking an action query executes it.

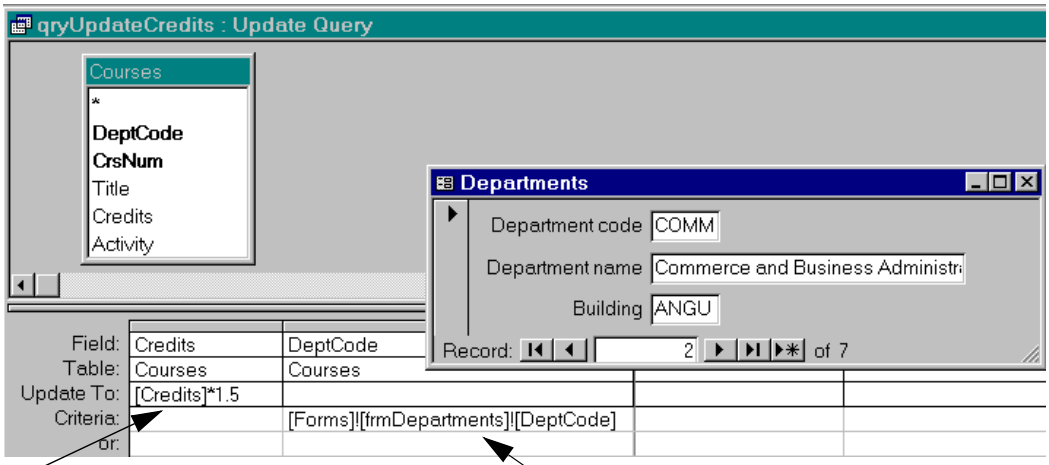
11.3.5 Attaching action queries to buttons

As a designer, you should not expect your users to understand your query naming convention, rummage through the queries listed in the database window, and execute the queries that need to be executed. As such, it is often useful to create buttons on forms and “attach” the action queries to the buttons. When the button is pressed, the query is executed.

Although we have not yet discussed buttons (or **events** in general), the button wizard makes the creation of this type of form object straightforward.

- Modify `qryUpdateCredits` so that it updates only those departments matching the `DeptCode` value in the `frmDepartments` table (see Figure 11.6).
- Save the resulting action parameter query as `pqryUpdateCredits` and close it.

FIGURE 11.6: Create an action parameter query to update `Credits` for a particular department.



- Switch to the design view of `frmDepartments` and add a button as shown in Figure 11.7.
- Attach the `pqryUpdateCredits` query to the button as shown in Figure 11.8.
- Provide a caption and a name for the button as shown in Figure 11.9.
- Switch to form view. Press the button to run the query (alternatively, use the shortcut key by pressing *Alt-U*) as shown in Figure 11.10.

11.4 Application to the assignment

11.4.1 Rolling back your master tables

As you begin to implement the transaction processing component of your system, it is worthwhile to have a means of returning your master tables to their original state (i.e., their state when you started developing the system).

- Create backup copies of your `Products` and `BackOrders` tables using make-tables queries. Save these queries but note that they only need to be run once.
- Create a rollback query that allows you to return your `Products` table to its original state.

Rolling back the `BackOrders` table is more complex than rolling back the `Products` table. This is because we are making the assumption that no products are ever added or deleted to the system. As such, all the information needed for the rollback is in the backup copy of `Products`.

In contrast, records are added to the `BackOrders` table on a regular basis. As a result, the `BackOrders` table and its backup may contain a different number of records. If so, the match-and-replace process used for rolling back `Products` is inappropriate.

FIGURE 11.7: Add a button to the form using the button wizard.

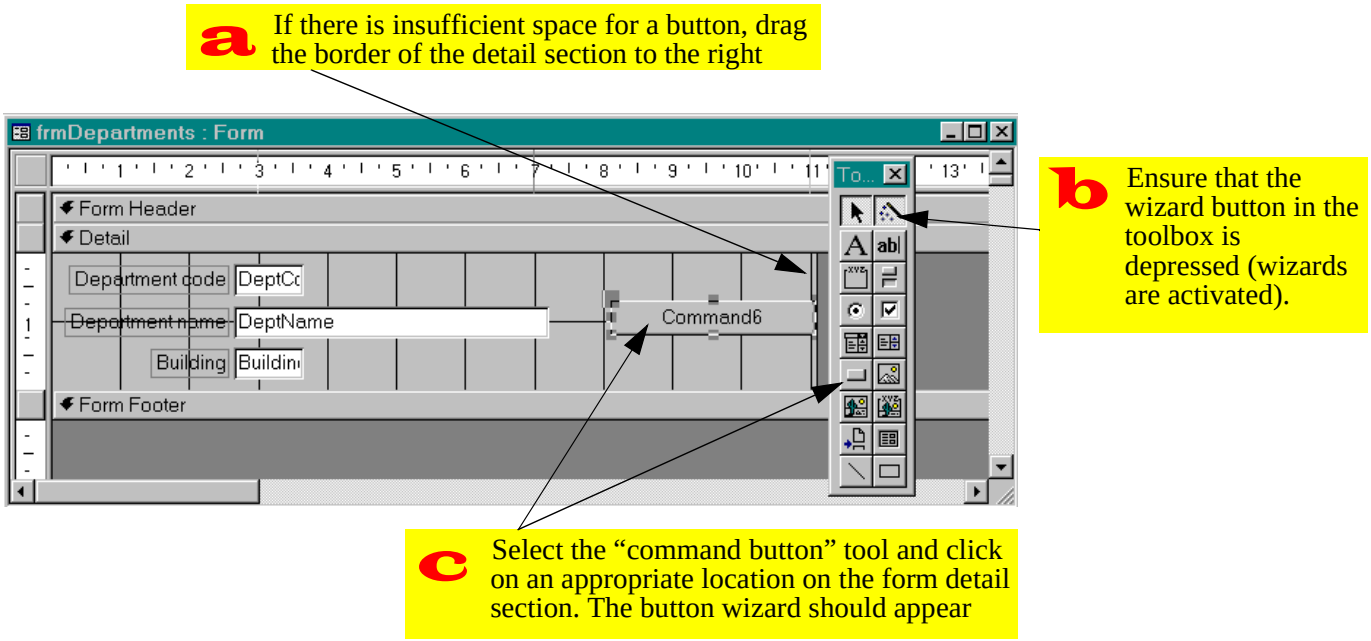
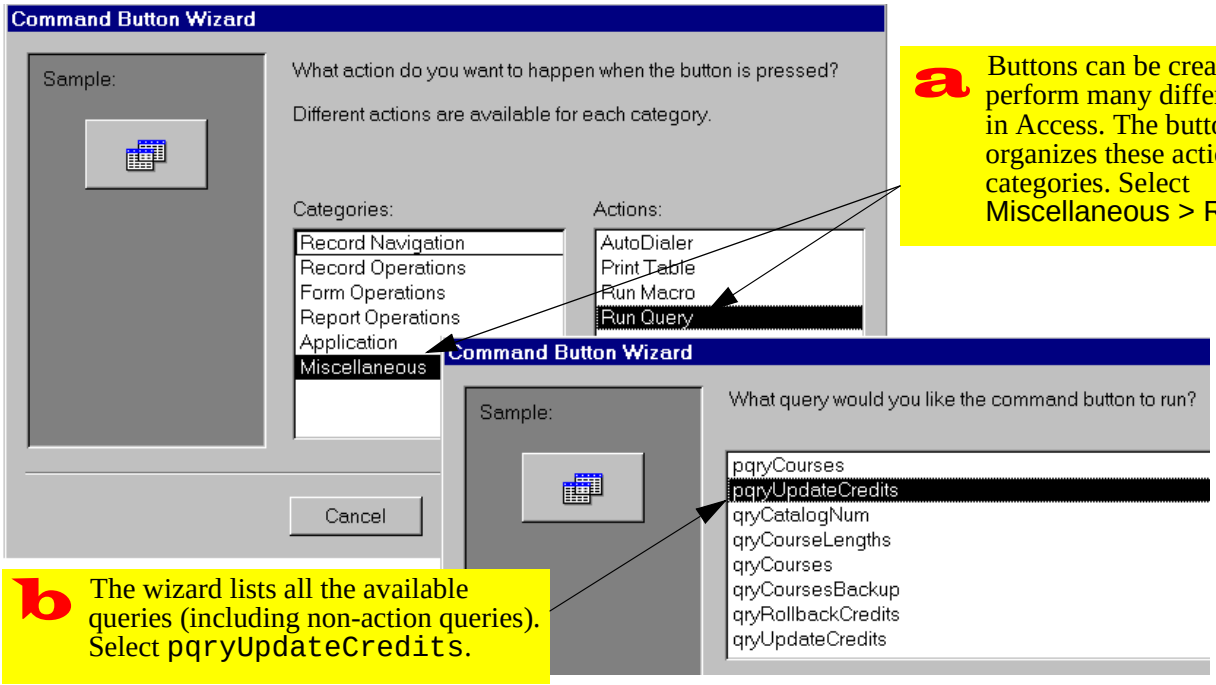


FIGURE 11.8: Use the wizard to attach an action query to the button.



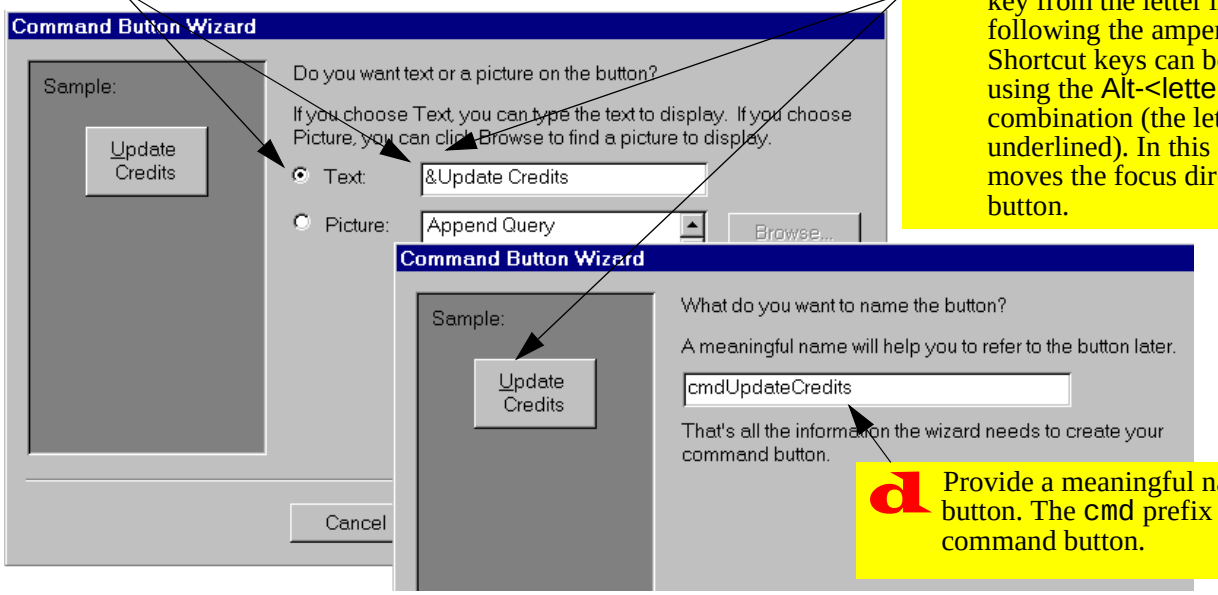
a Buttons can be created to perform many different actions in Access. The button wizard organizes these actions into categories. Select Miscellaneous > Run Query.

b The wizard lists all the available queries (including non-action queries). Select `pqryUpdateCredits`.

FIGURE 11.9: Use the wizard to attach a query to a button (continued)

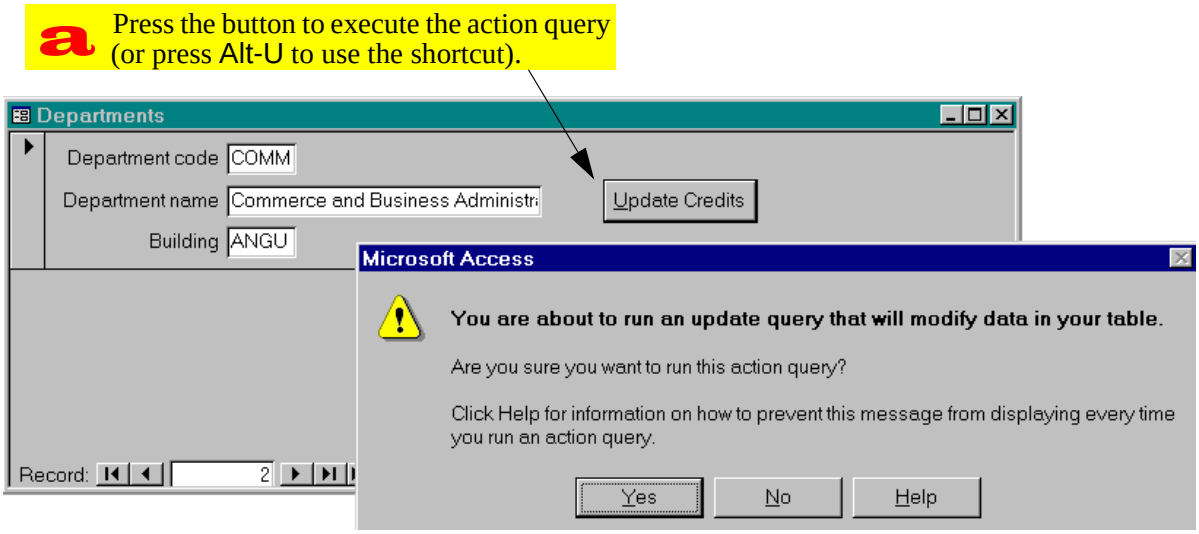
c You can show either a picture (icon) or a caption on the button. Enter a suitable caption.

? Including an ampersand (&) in the caption creates a shortcut key from the letter immediately following the ampersand. Shortcut keys can be invoked using the Alt-<letter> combination (the letter is underlined). In this case, Alt-U moves the focus directly to the button.



d Provide a meaningful name for the button. The `cmd` prefix indicates a command button.

FIGURE 11.10: Execute the action query by pressing the button.



The easiest way to rollback the `BackOrders` table is to delete all the records it contains and use an append query to replace the records from the backup.

- Open your `BackOrders` table in datasheet mode and select *Edit > Select All Records* from the menu (alternatively, press *Control-A*)
- Press the *Delete* key.
- Create an append query that adds the records in the backup table to the `BackOrders` table.

Once you learn the Access macro language or Visual Basic for Applications, you will be able to write a small procedure to execute these steps for you. For the assignment, however, this “manual rollback” is sufficient.

11.4.2 Processing transactions

You are now in a position to combine parameter queries and action queries into parameter-action queries. These queries will allow you to perform reasonably complex transaction processing operations on your master tables.

- Create an update query to add all products in a shipment to inventory.

Note that this query should only process shipment details for the current shipment (i.e., it should be based on a parameter query similar to the one you created in Section 10.4).

- Create a button on the shipments form to perform this update.
- Create an update query to subtract items from inventory when you process an order from your customers. Do not attach this query to a button at this point.

This query should only process order details from the current order.

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.

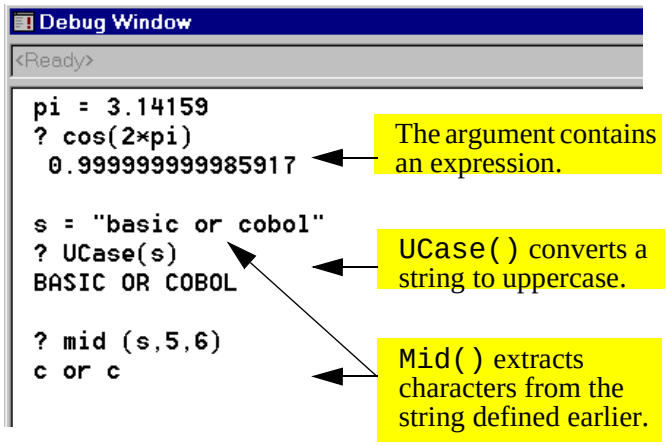
In this section, we are going to explore some basic predefined functions for working with numbers and text. The results of these exercises are shown in Figure 12.1.

- 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

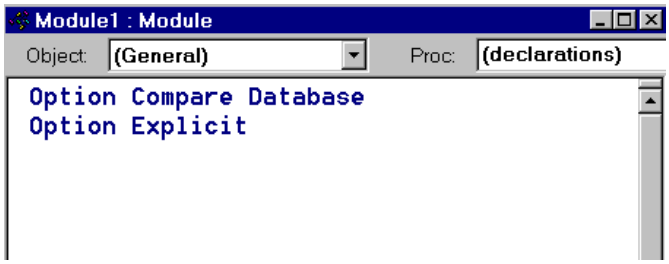
- Close the debug window so that the declaration page of the new module created in Section 12.3.3 is visible (see Figure 12.2).

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**.

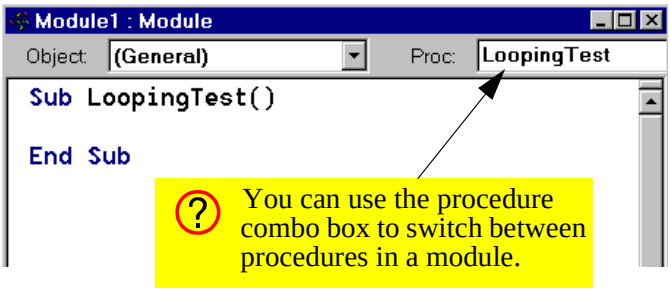
- Create a new subroutine by typing the following anywhere on the declarations page:
  `Sub LoopingTest()↵`

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

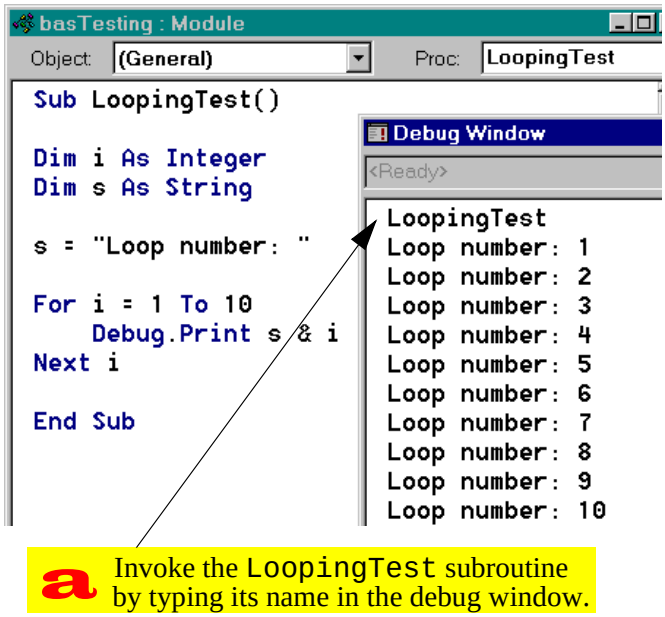
Now that you have created a subroutine, you need to run it to see that it works. To invoke a subroutine, you simply use its name like you would any statement.

- 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*.

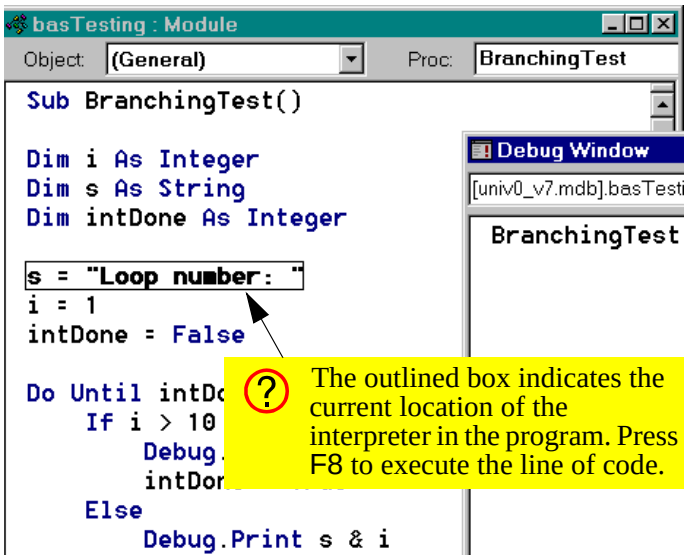
By stepping through a program line by line, you can usually find any program bugs. In addition, you can use the debug window to examine the value of variables while the program's execution is suspended.

- 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

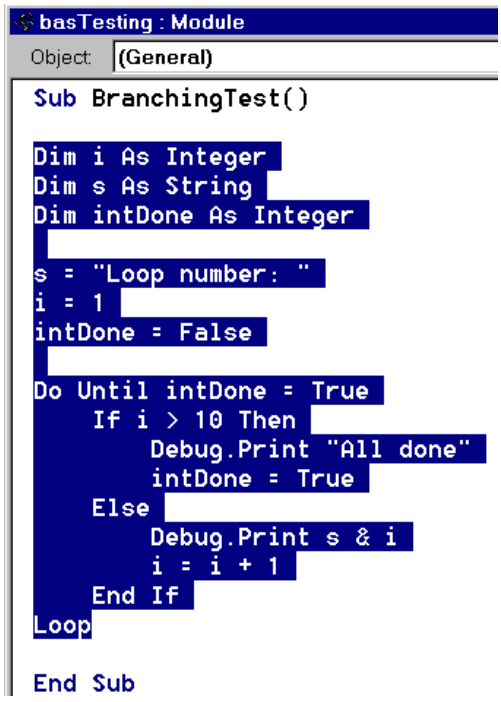
```
Sub BranchingTest(intStart as
  Integer, intStop as Integer)
```

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.

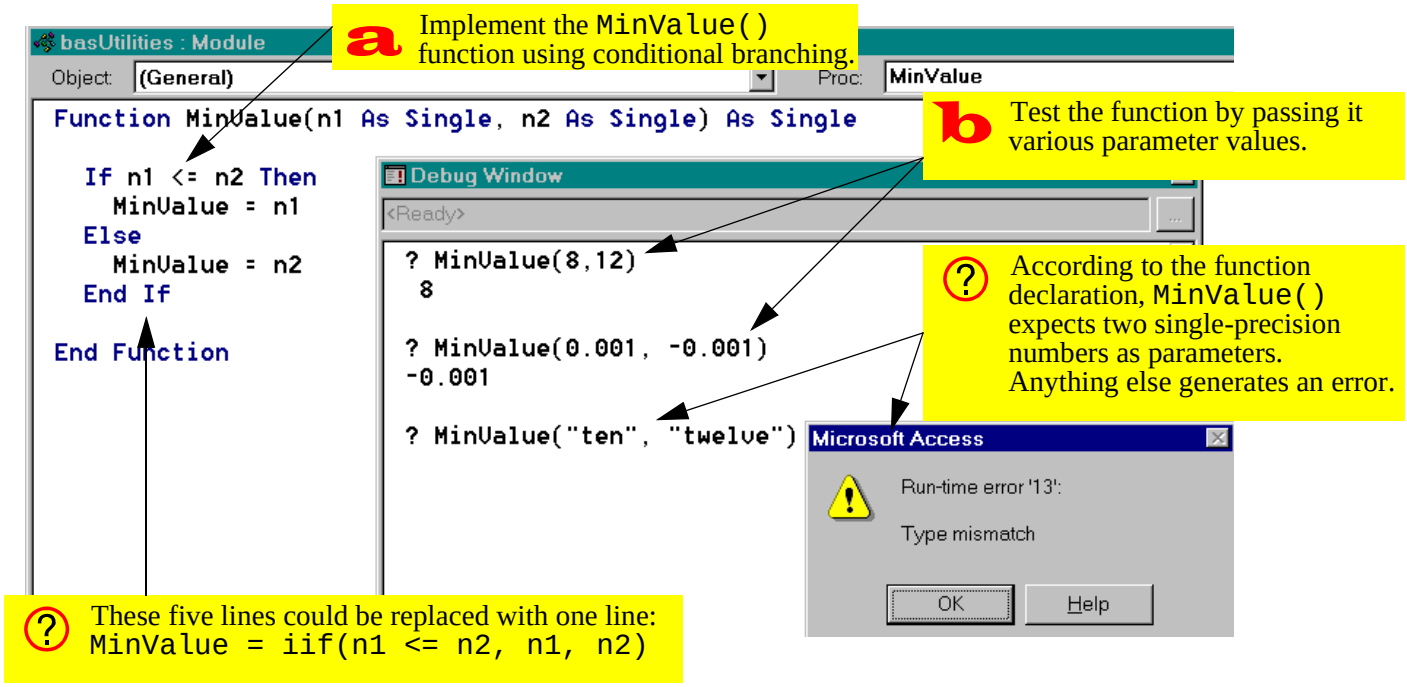
12.4 Discussion

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?

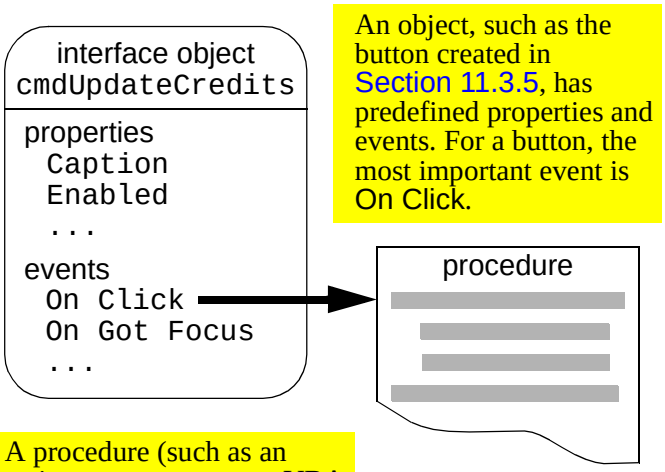
In conventional programming, the sequence of operations for an application is determined by a central controlling program (e.g., a main procedure). In **event-driven** programming, the sequence of operations for an application is determined by the user's interaction with the application's interface (forms,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

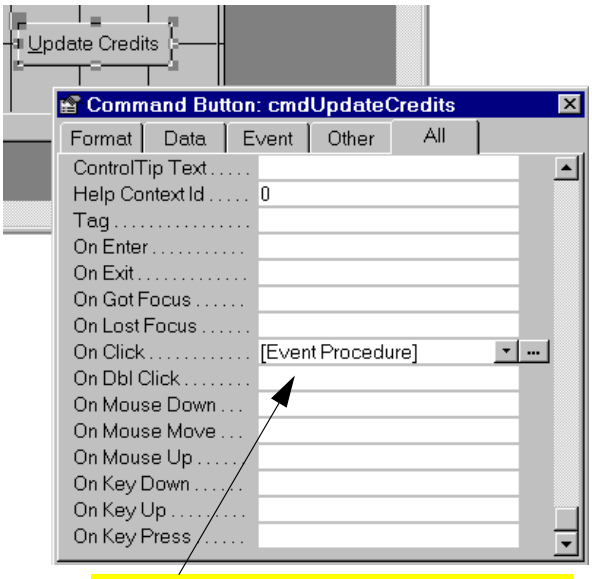


FIGURE 13.2: The macro editor.

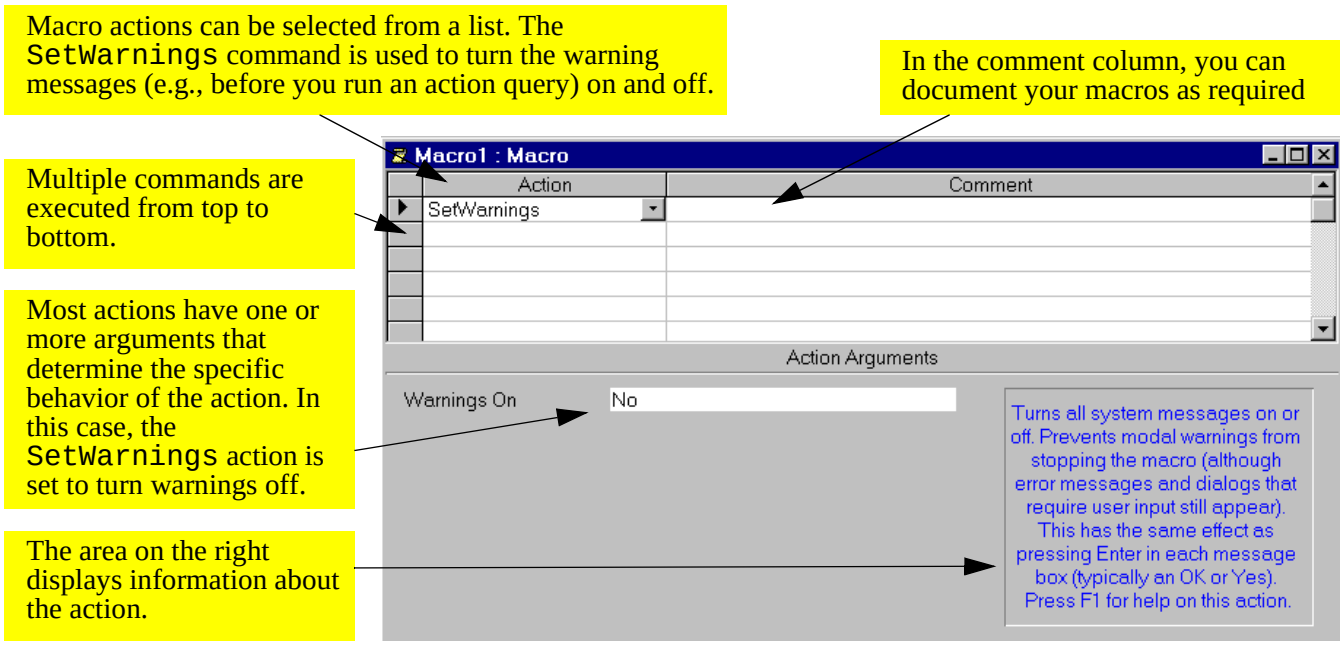
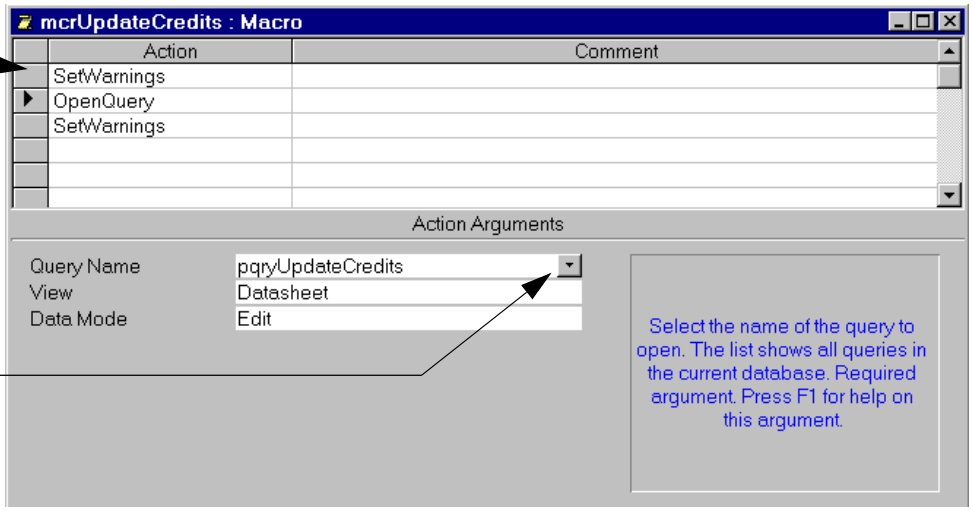


FIGURE 13.3: Create a macro that answers the "what" question.

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.

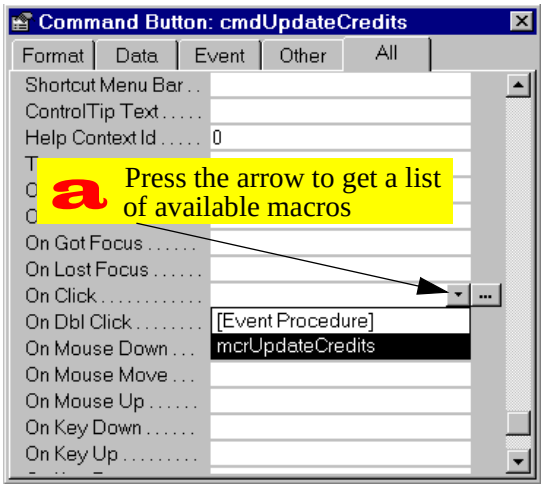


- Press the builder button (…) beside the existing procedure and look at the VBA subroutine created by the button wizard. Most of this code is for error handling.

? 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`.

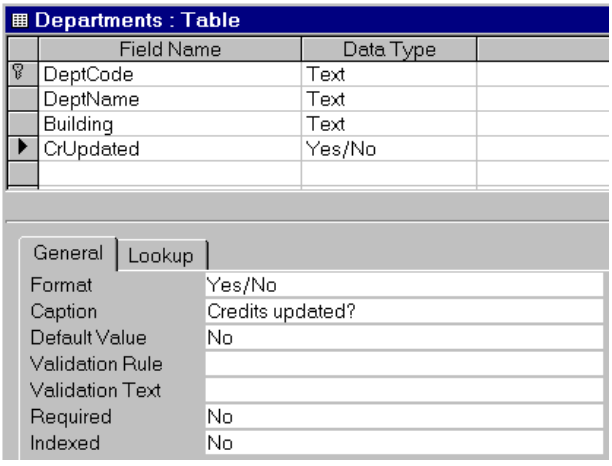
If you have an open query or form based on the `Departments` table, you will not be able to modify the structure of the table until the query or form is closed.

- 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

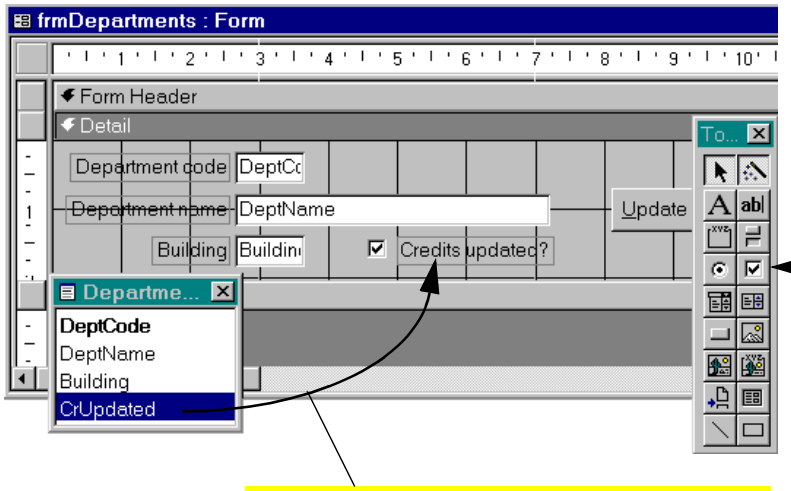
So far, you have used two commands in the Access macro language: `SetWarnings` and `OpenQuery`. In this section, you are going to use one of the most useful commands—`SetValue`—to automatically change the value of the `CrUpdated` check box.

- 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.



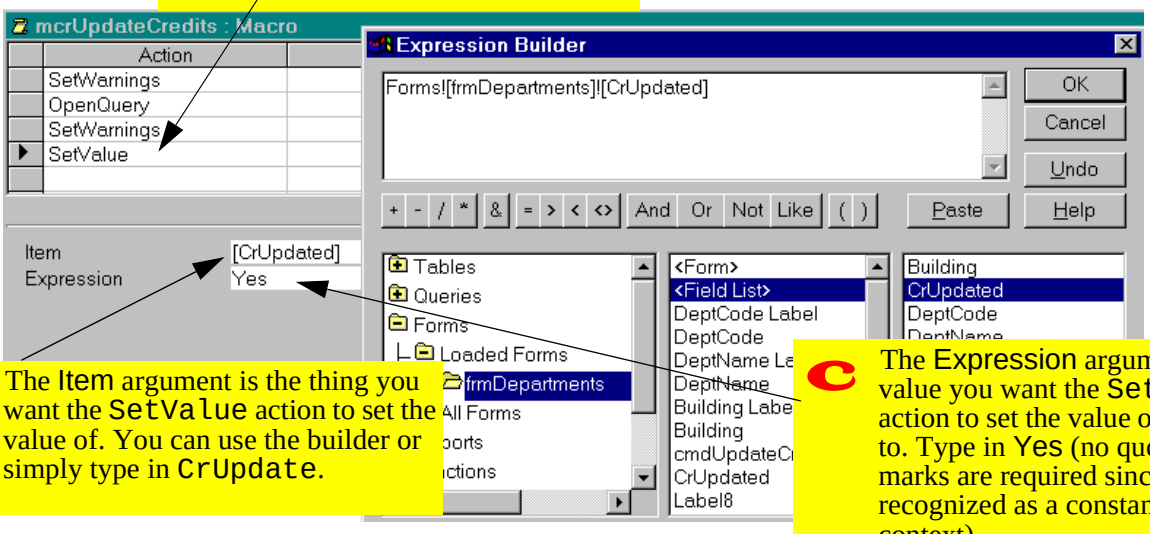
a Select the check box tool from the toolbox.

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.

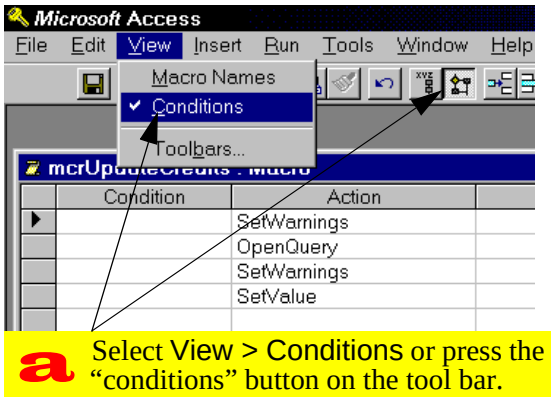


b The Item argument is the thing you want the `SetValue` action to set the value of. You can use the builder or simply type in `CrUpdate`.

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



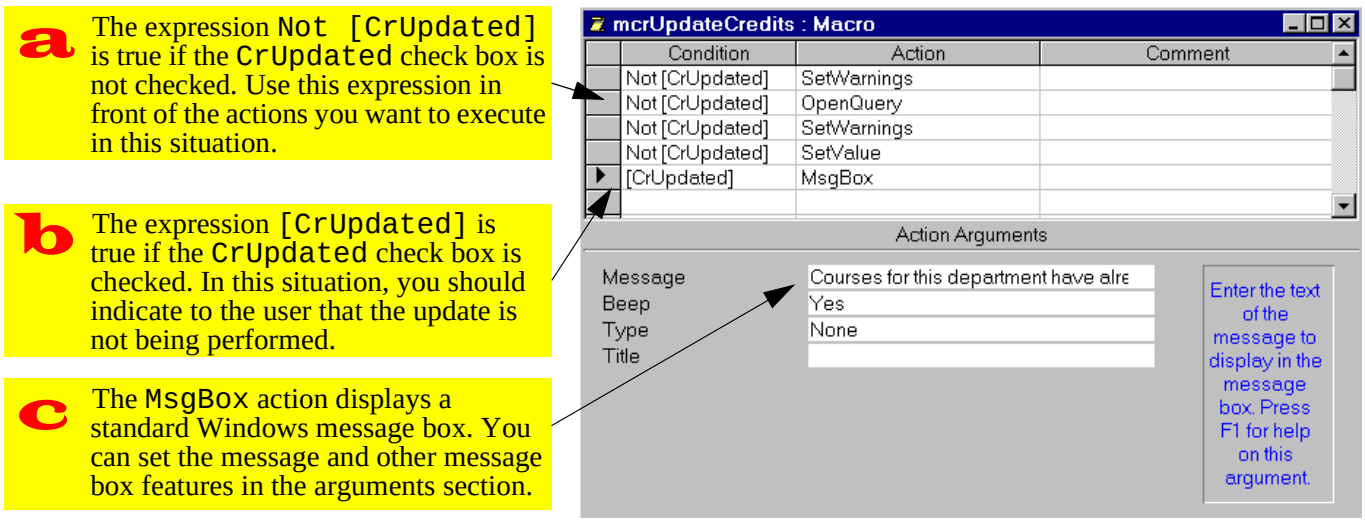
13.3.5.1 The simplest conditional macro

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]`.

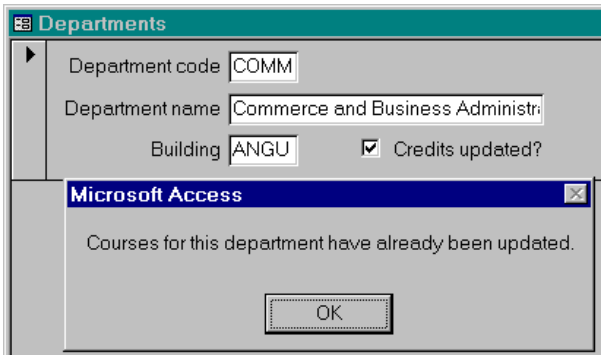
Since `CrUpdated` is a **Boolean** (yes/no) variable, you do not need to write `[CrUpdated] = True` or `[CrUpdated] = False`. The true and false parts are implied. However, if a non-Boolean data type is used in the expression, a comparison operator must be included (e.g., `[DeptCode] = "COMM"`, `[Credits]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

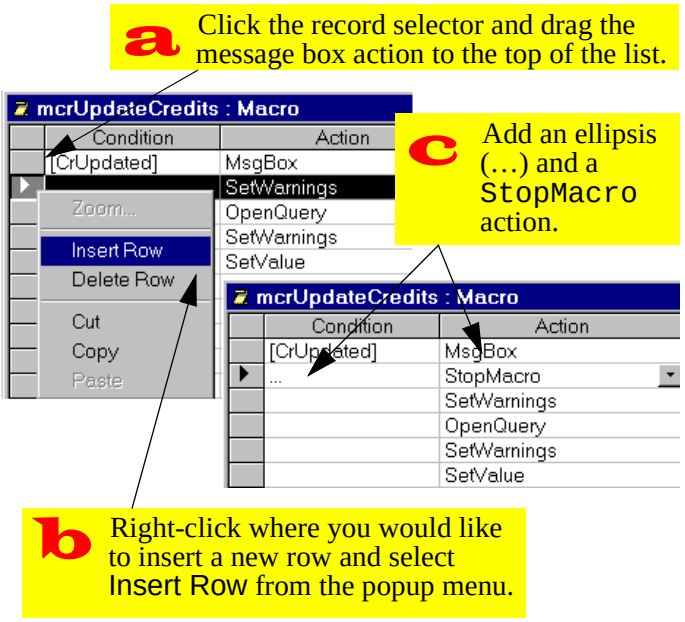
The macro shown in Figure 13.10 can be improved by using an ellipsis (…) instead of repeating the same condition in line after line. In this section, you will simplify your conditional macro slightly.

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.

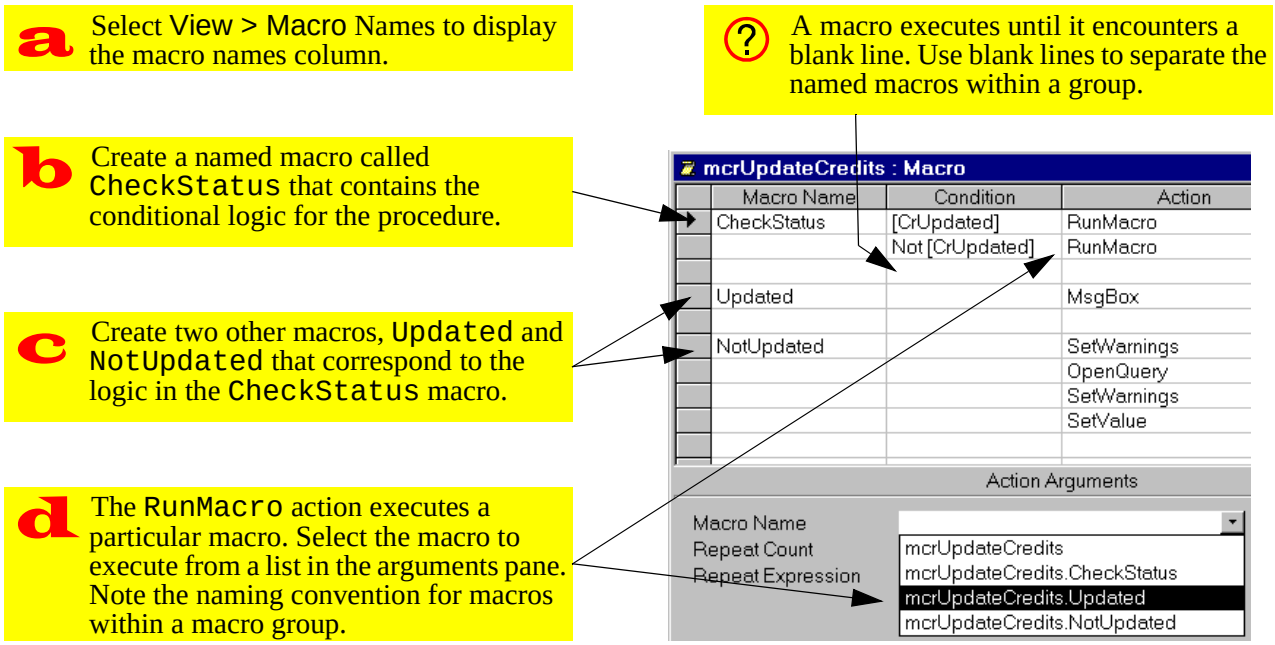
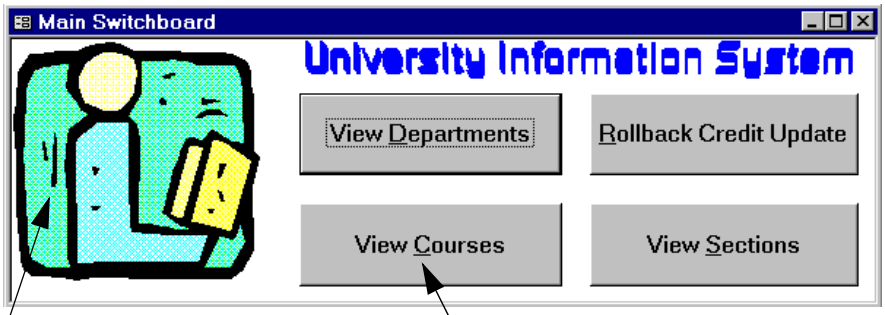


FIGURE 13.14: A switchboard interface to the application.

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

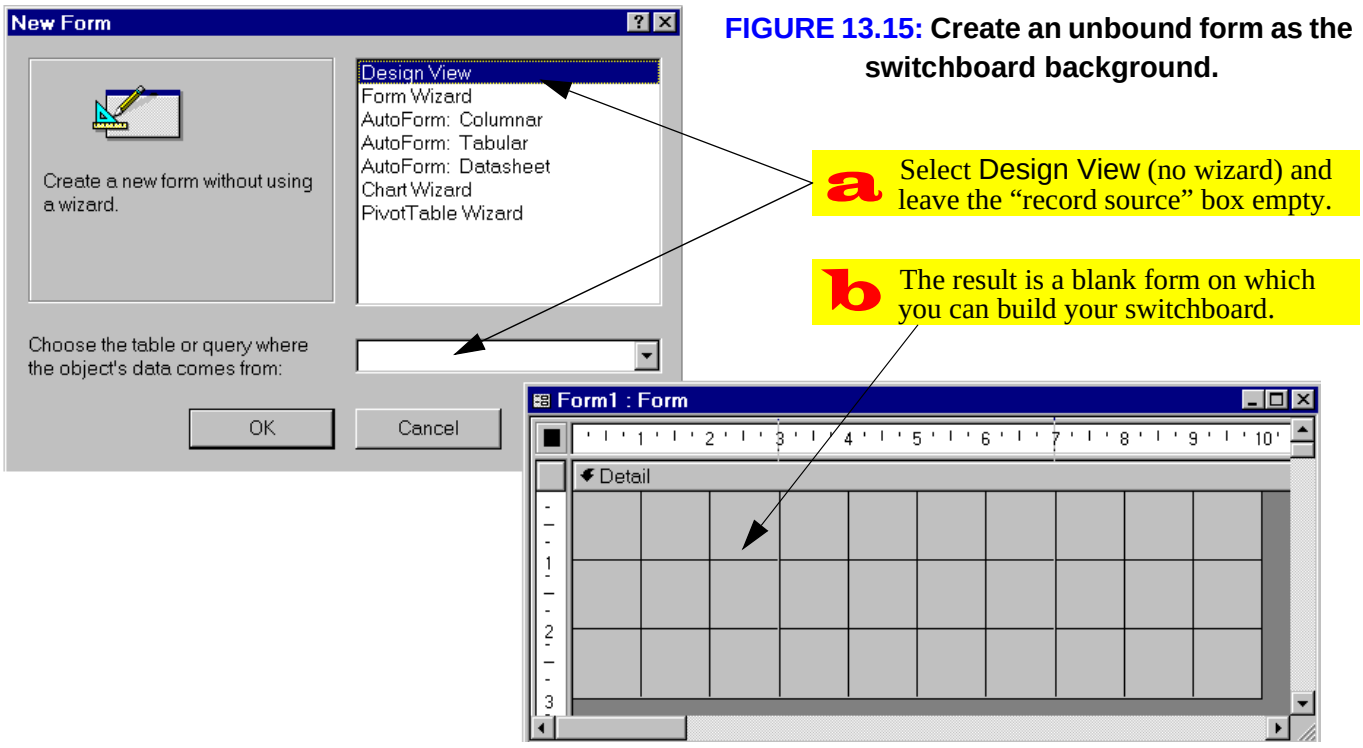
Although it is not shown here, switchboards can call other switchboards, allowing you to add a hierarchical structure to your application.



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.

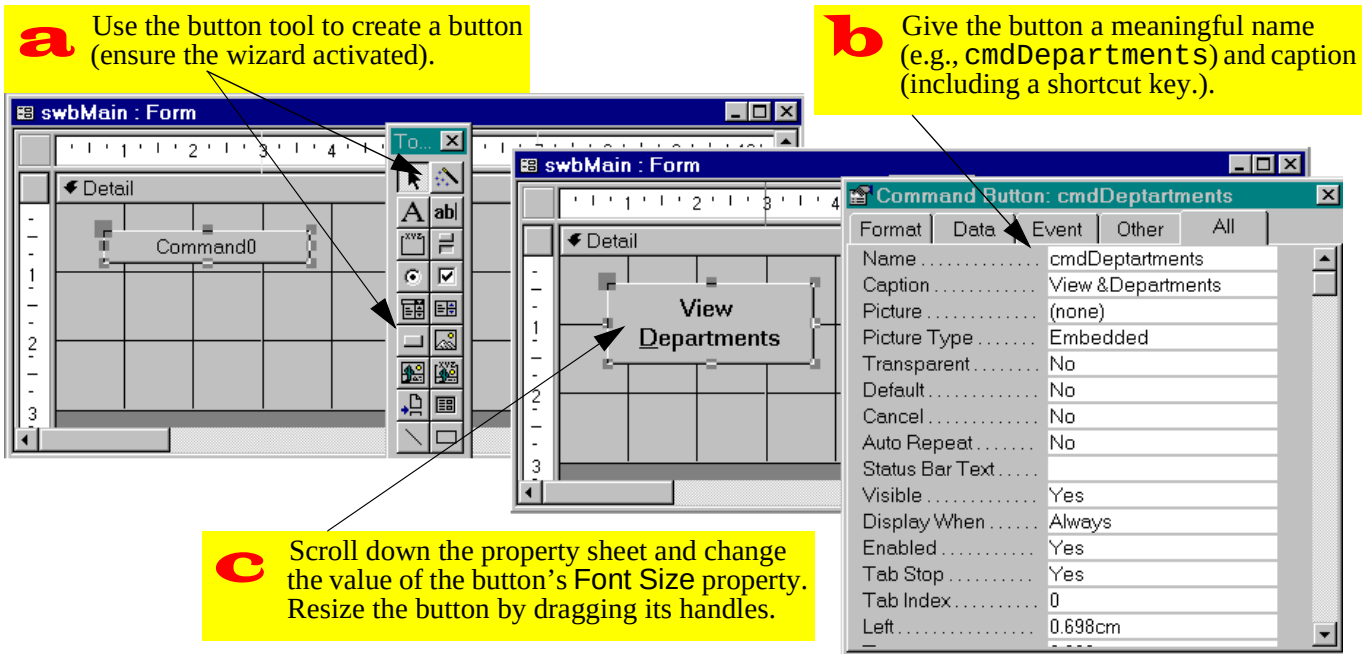
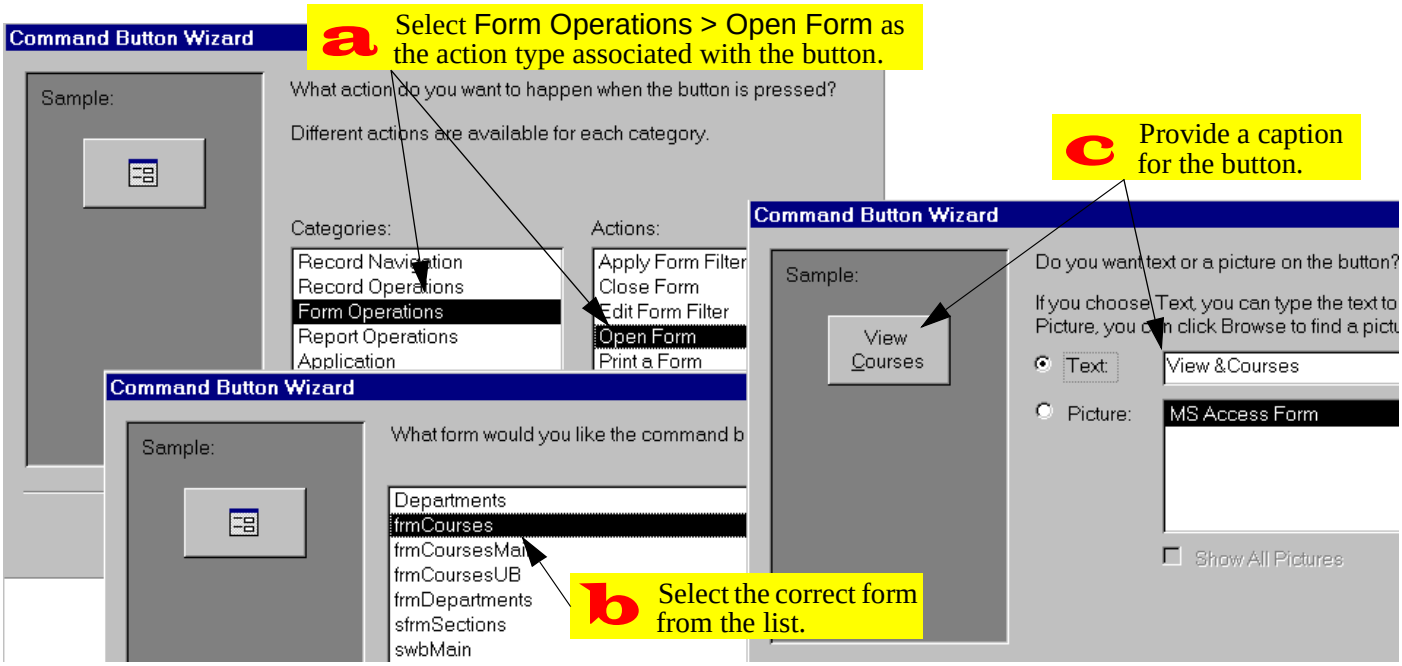


FIGURE 13.17: Use the command button wizard to create a button for the switchboard.



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.

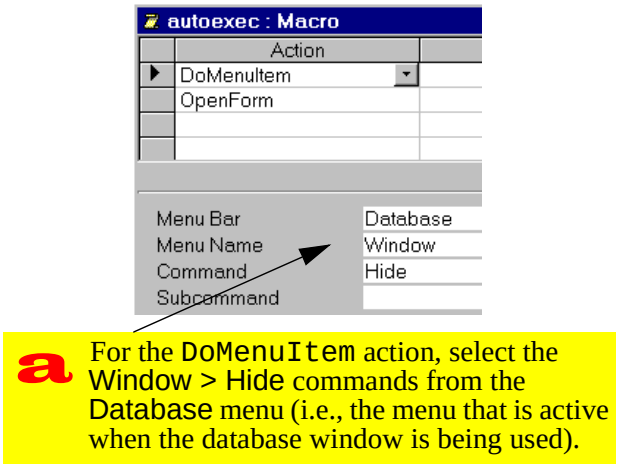
As its name suggests, the `DoMenuItem` action performs an operation just as if it had been selected from the menu system. Consequently, you need to know something about the menu structure of Access before you create your macro.

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.

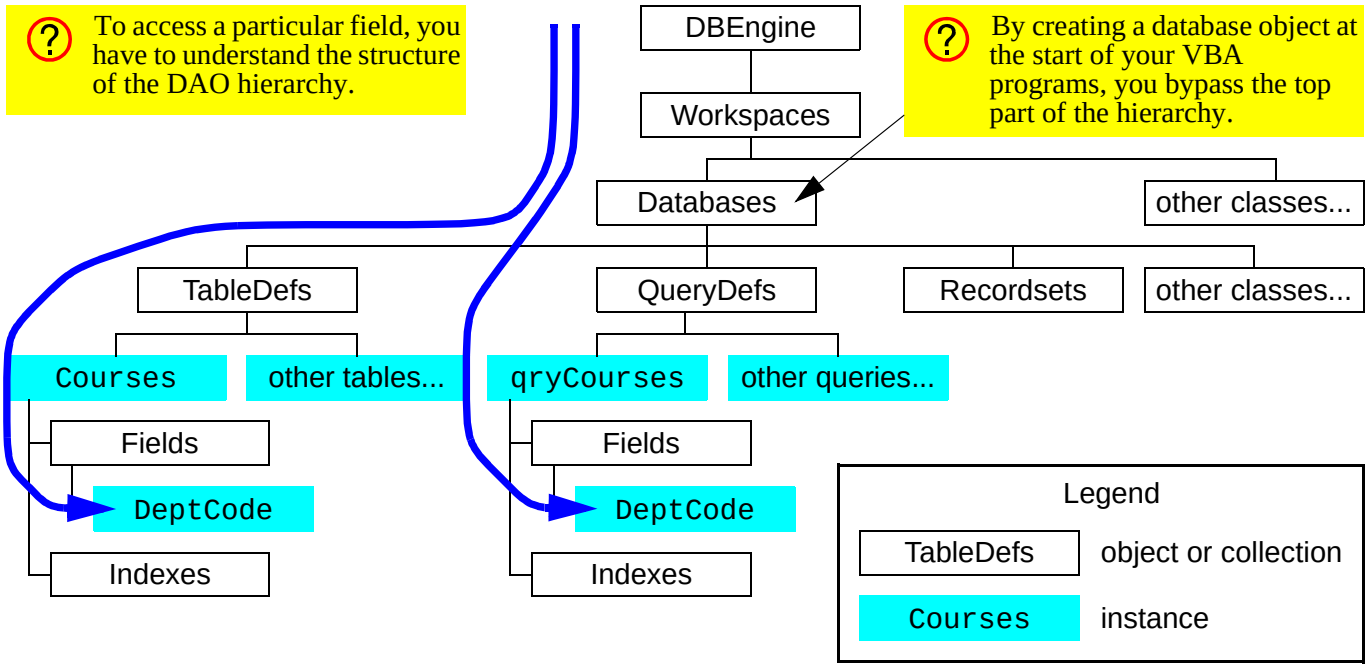
A reasonable question at this point might be: Isn't FieldSize a property of a field, not a method? The answer to this is that the implementation of DAO is somewhat inconsistent in this respect. The best policy is to look at the object summaries in the on-line help if you are unsure.

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:

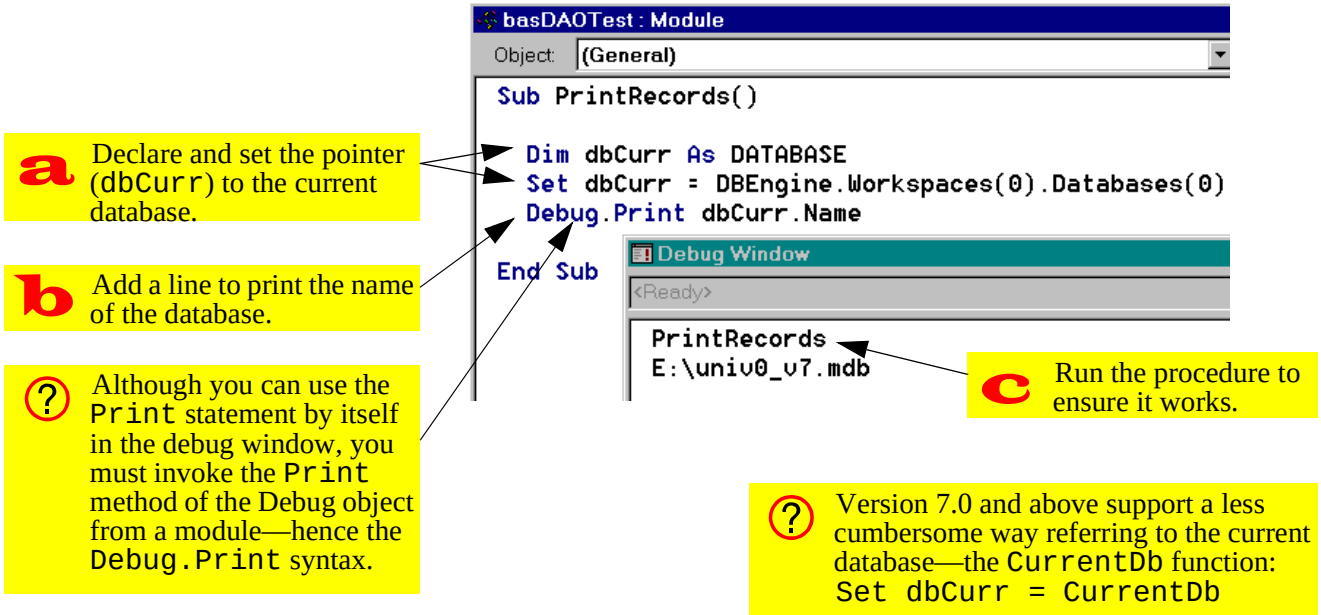
```
Dim dbCurr As DATABASE
Set dbCurr =
  DBEngine.Workspaces(0).Databases(0)
Debug.Print dbCurr.Name
```

- 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

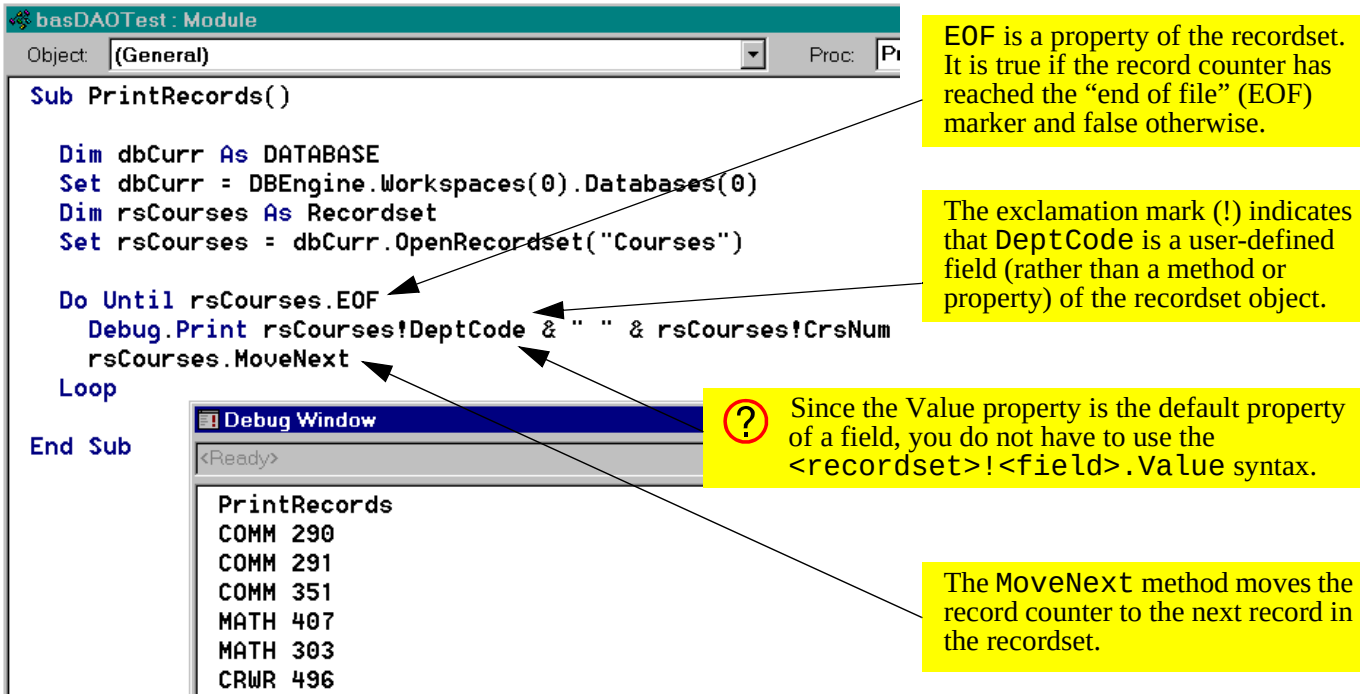
In this section, you will use some of the properties and methods of a Recordset object to print its contents.

- 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()`.

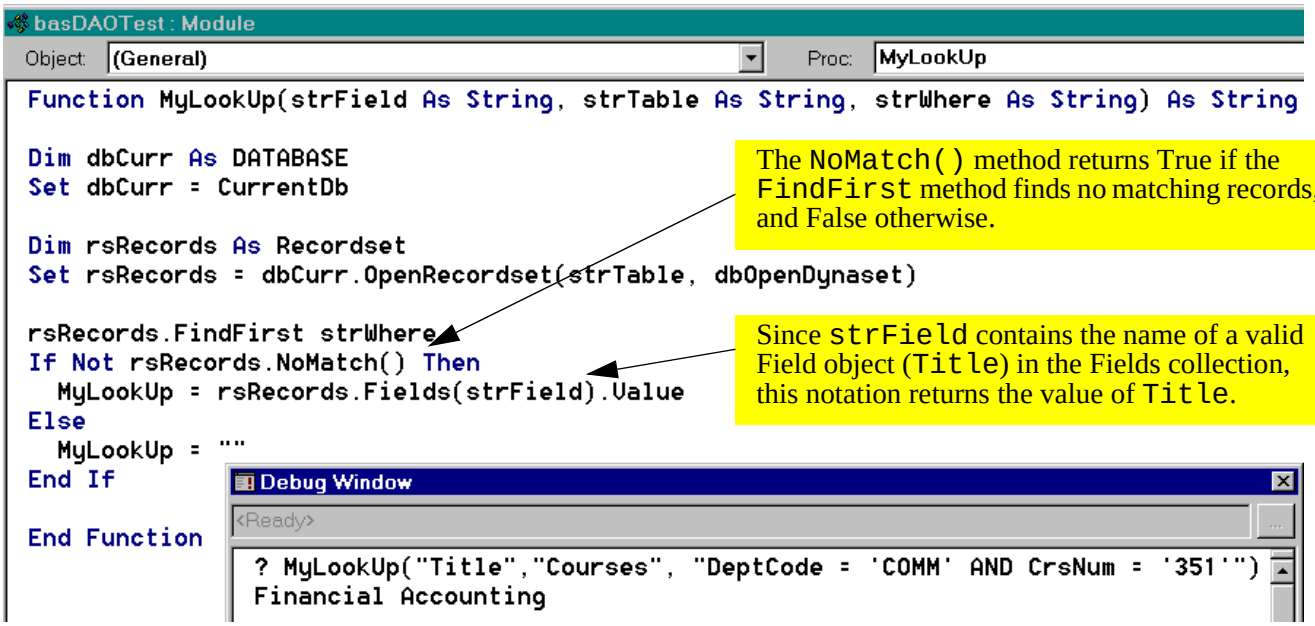
14.3.5 The `DLookUp()` function

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] & "'")
```

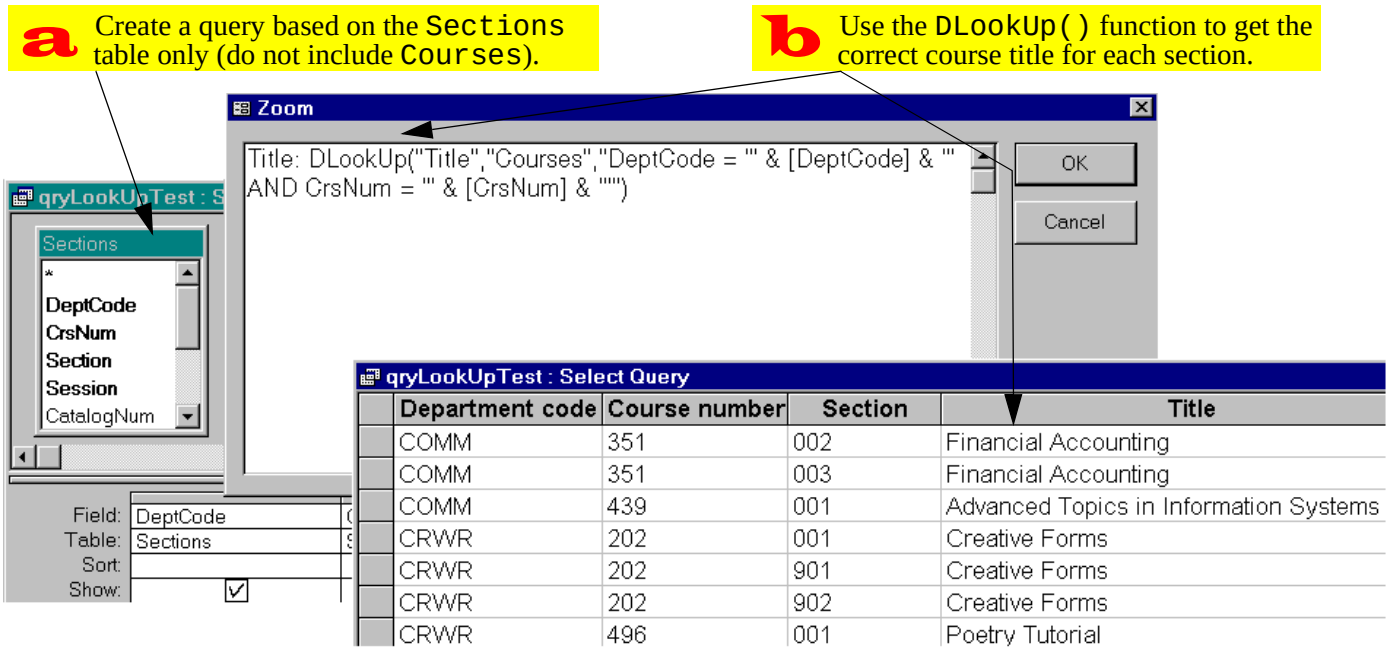
14.3.5.2 Understanding the WHERE clause

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

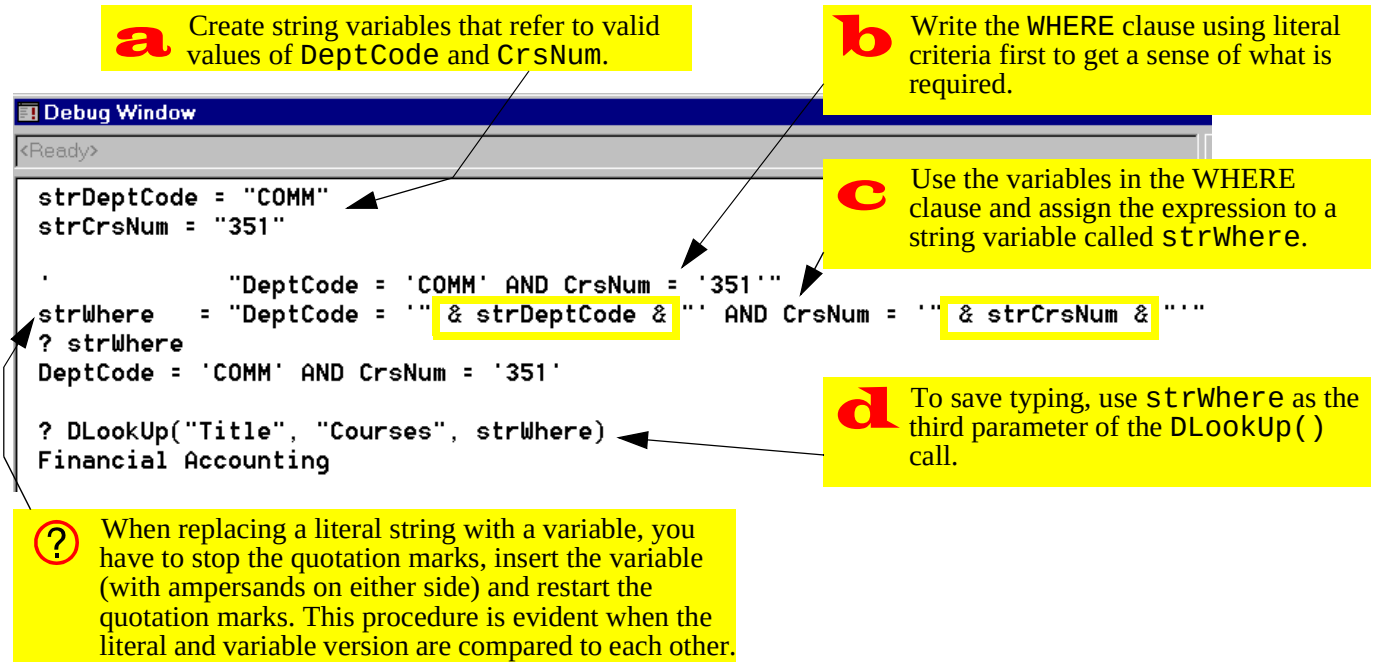
14.4.1 VBA versus SQL

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
```

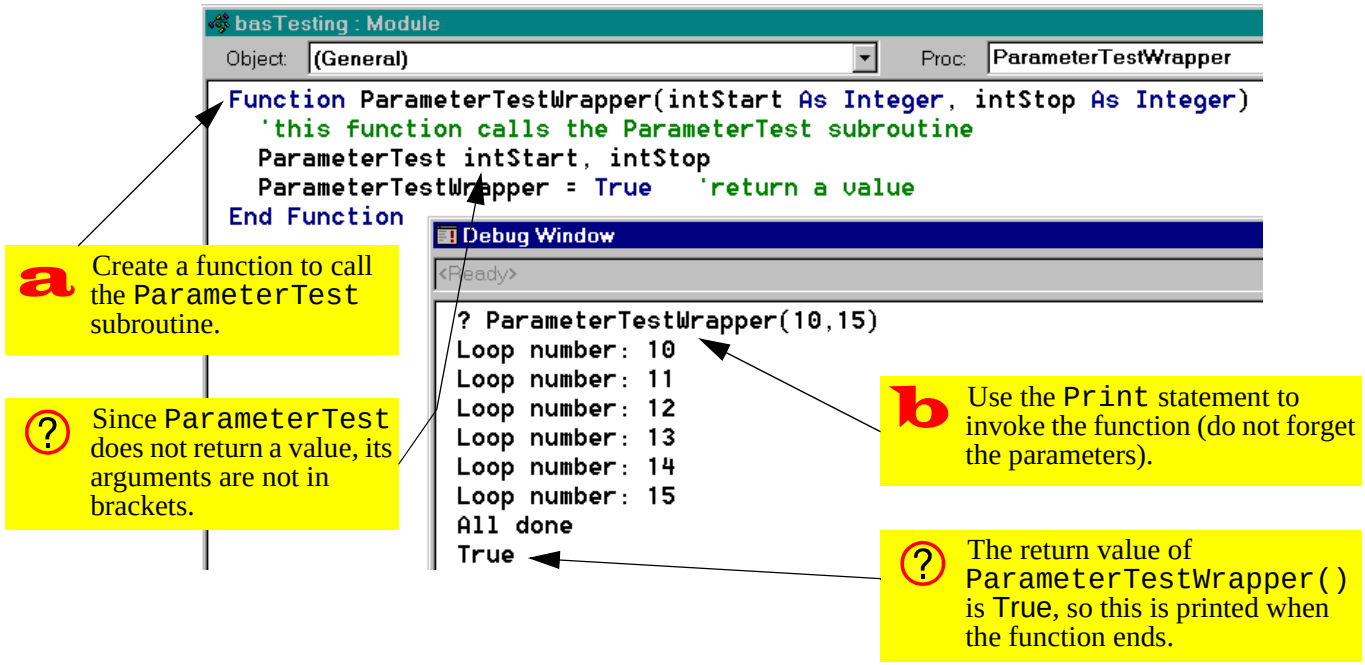
- Call the function, as shown in Figure 15.1.

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.

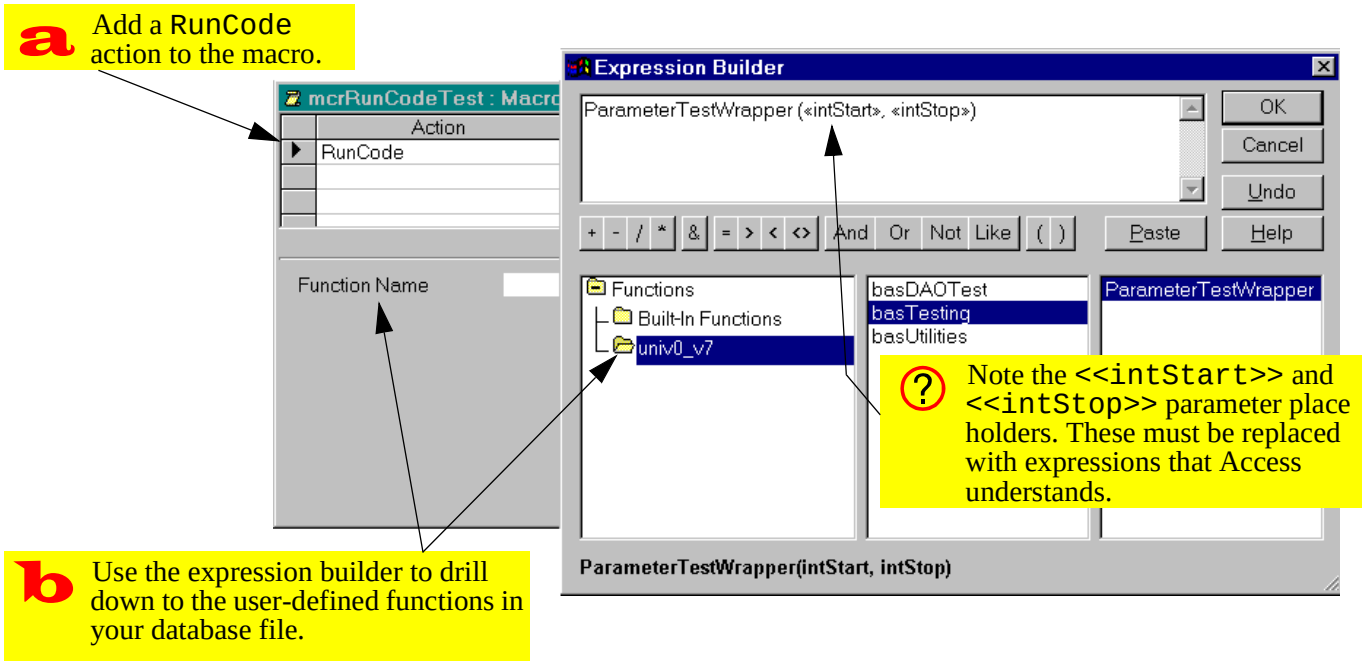
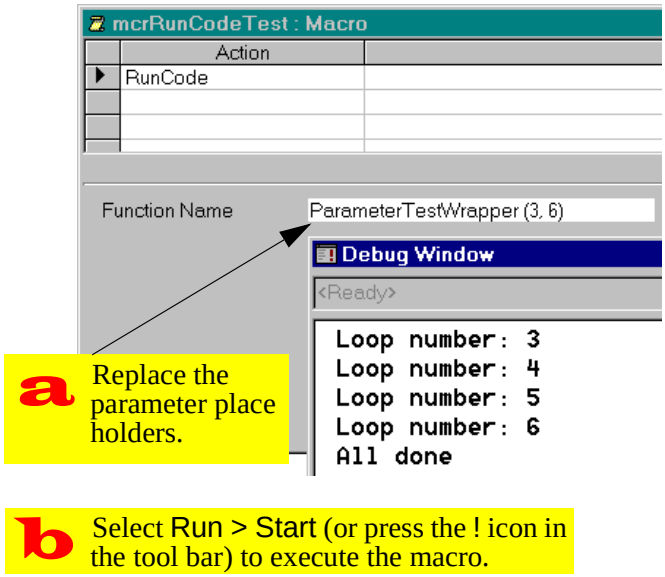


FIGURE 15.3: Execute the `RunCode` macro.



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.

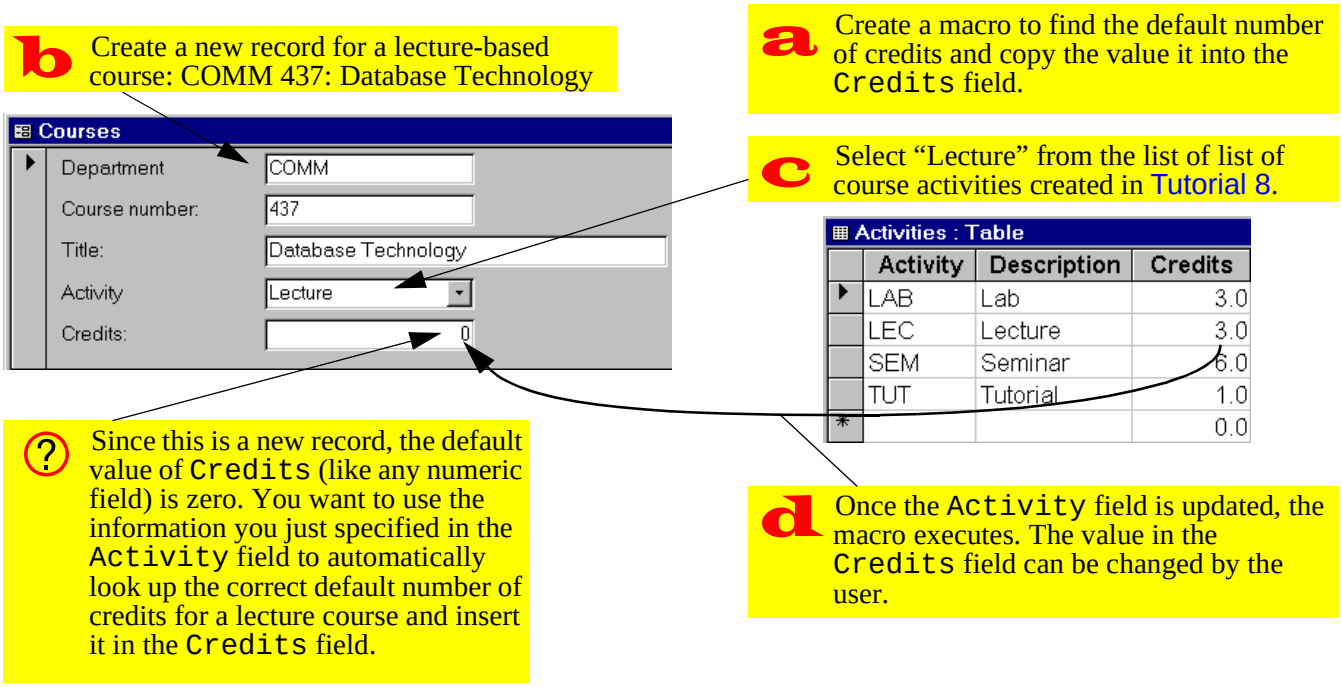
You want to use the default credit values when you create a new course or modify an existing course. However, the user may override this default if necessary for a particular course. The basic requirement is illustrated in Figure 15.4.

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

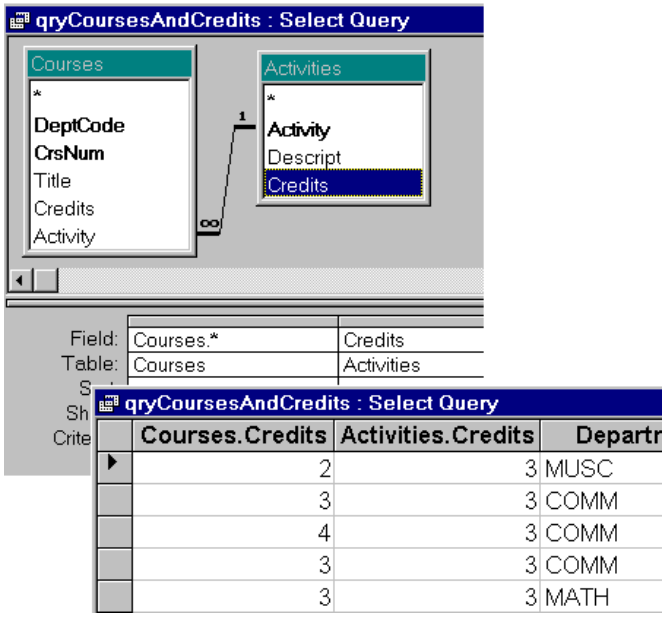
As you discovered in Section 14.3.5, Access has a `DLookUp()` function that allows you to go to the `Activities` table and find the value of `Credits` for a particular value of `Activity`. A different approach is to join the `Activities` table with the `Courses` table in a query so that the default value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

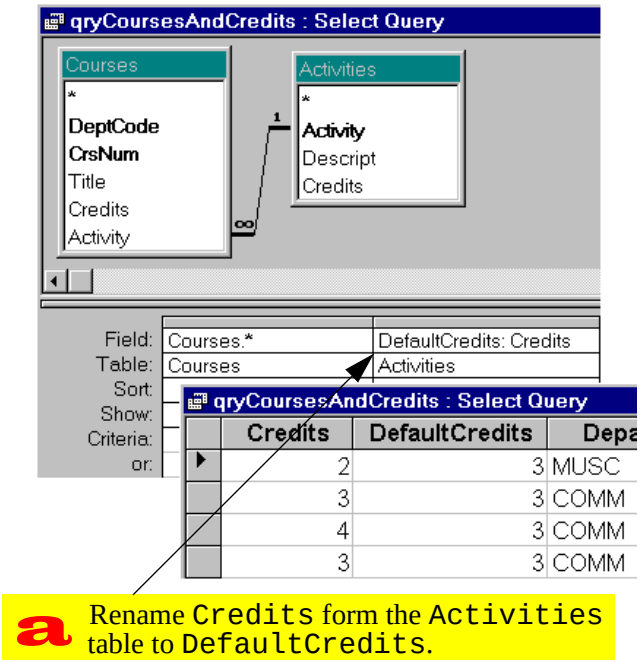
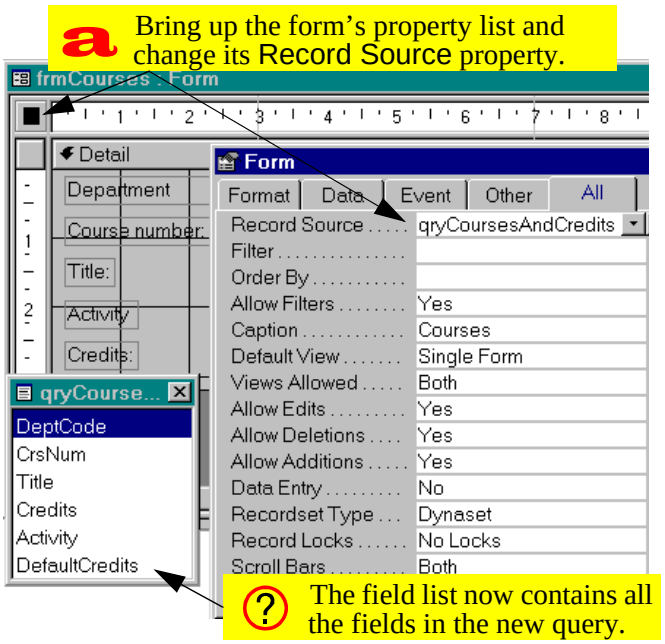


FIGURE 15.7: Change the *Record Source* property of an existing form.



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

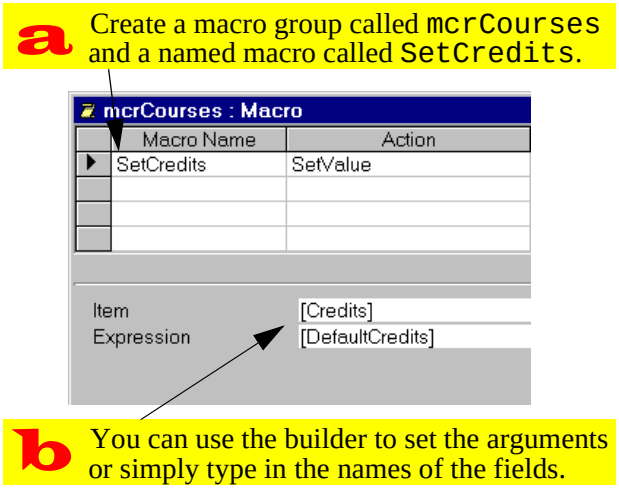
The `SetValue` macro you require here is extremely simple once you have `DefaultCredits` available within the scope of the form.

- 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.

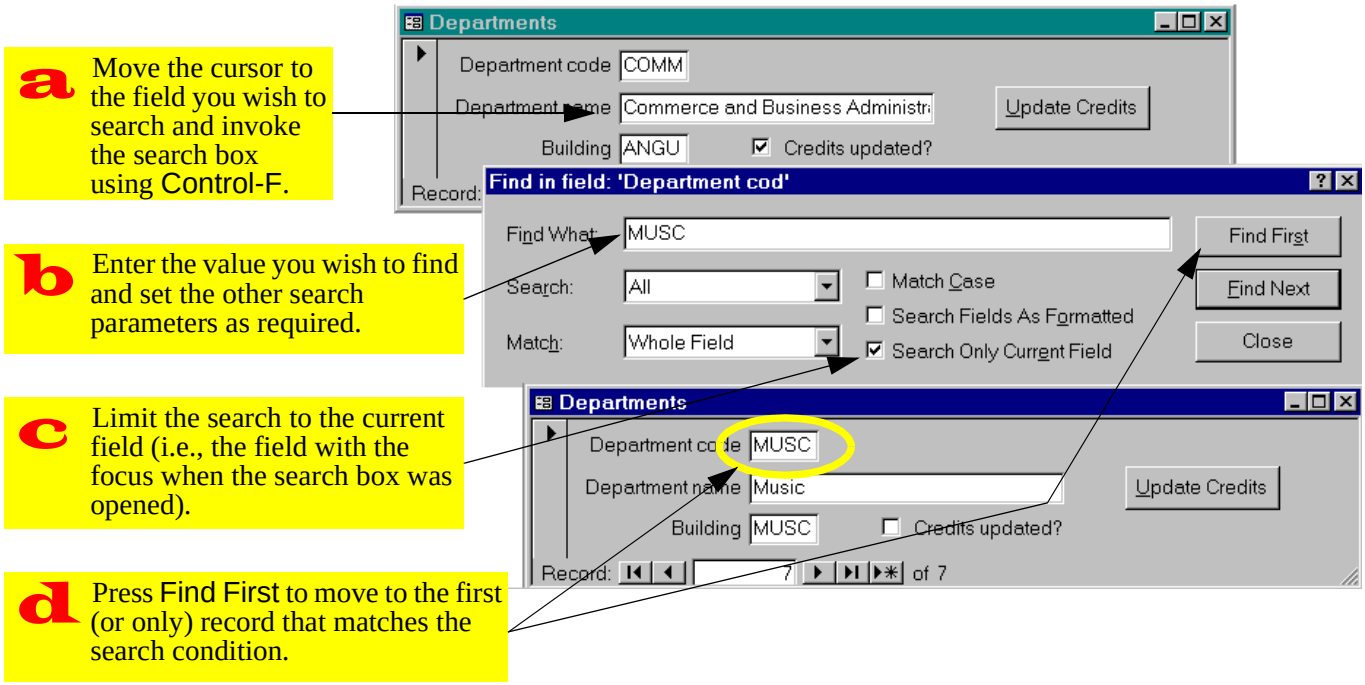


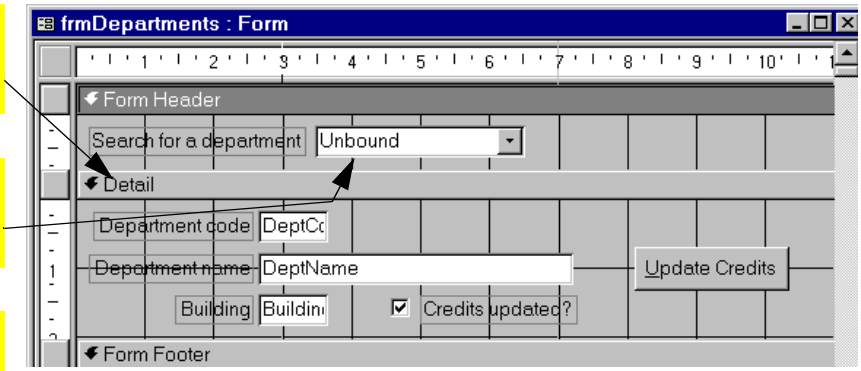
FIGURE 15.10: Create an unbound combo box.

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.



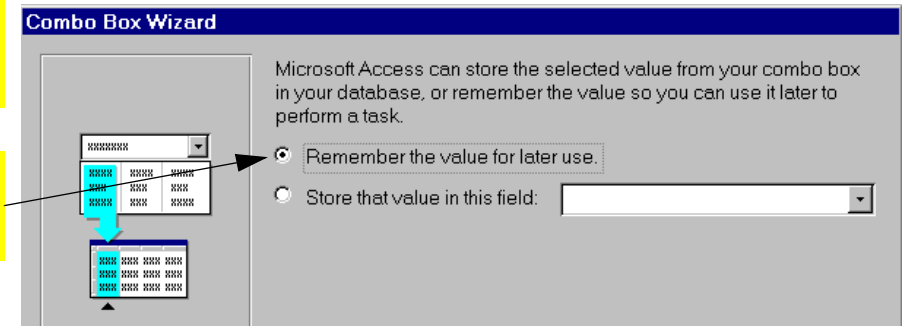
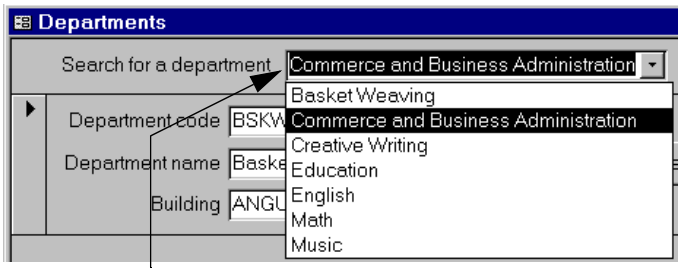


FIGURE 15.11: An unbound combo box.



? 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.

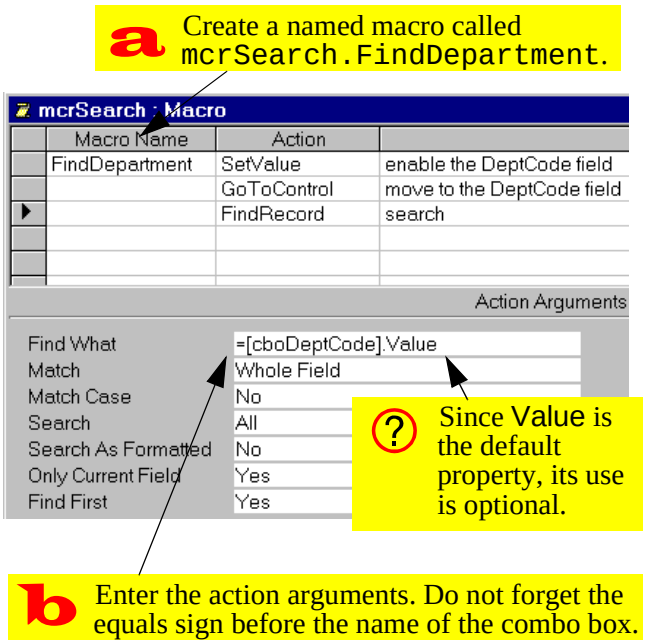
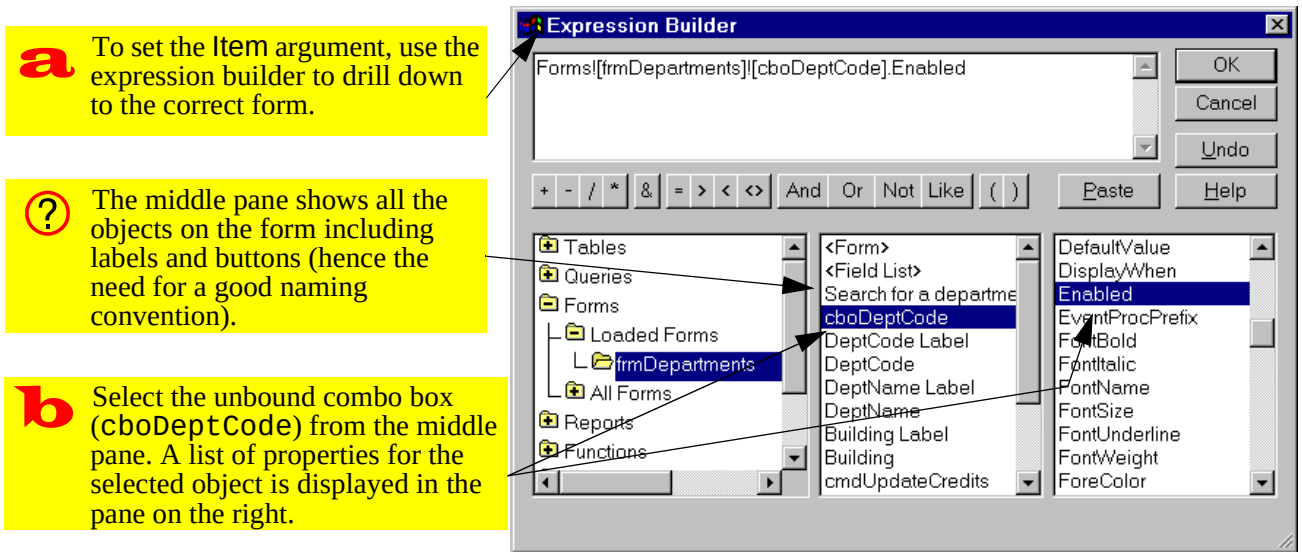


FIGURE 15.12: Use the builder to specify the name of the property to set.



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:

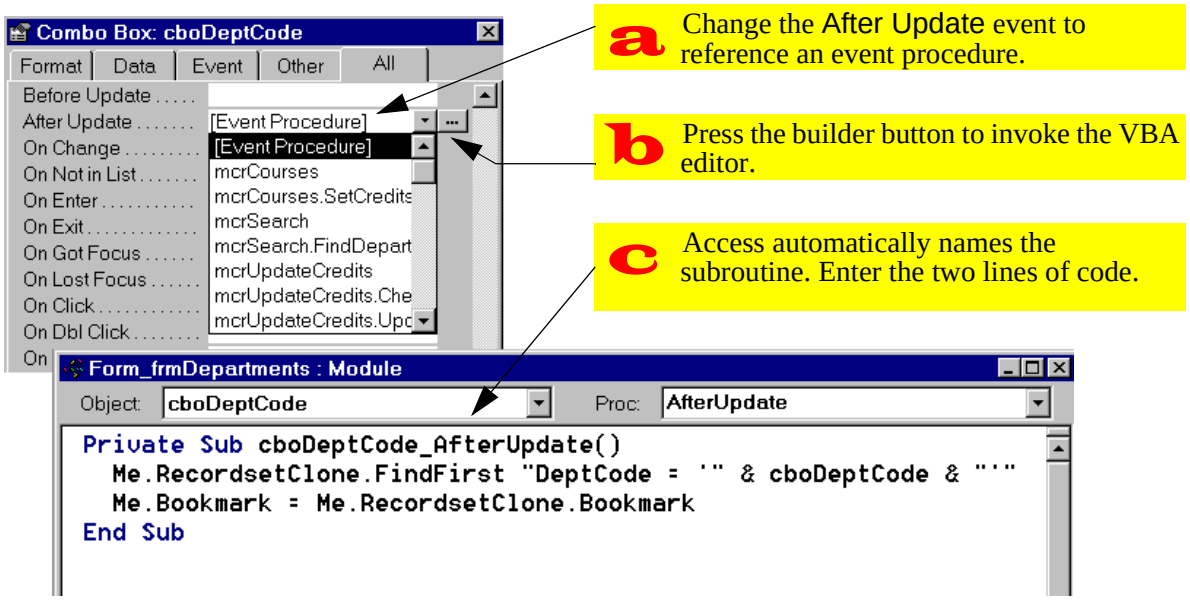
- The property `Me` refers to the current form. You can use the form's actual name, but `Me` is much faster to type.
- A form’s `RecordsetClone` property provides a means of referencing a copy of the form's underlying recordset.
- The `FindFirst` method is straightforward. It acts, in this case, on the 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()`.

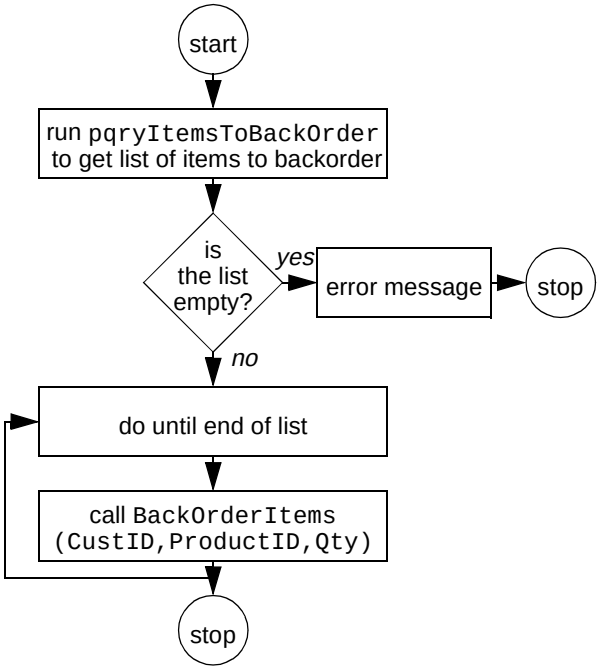
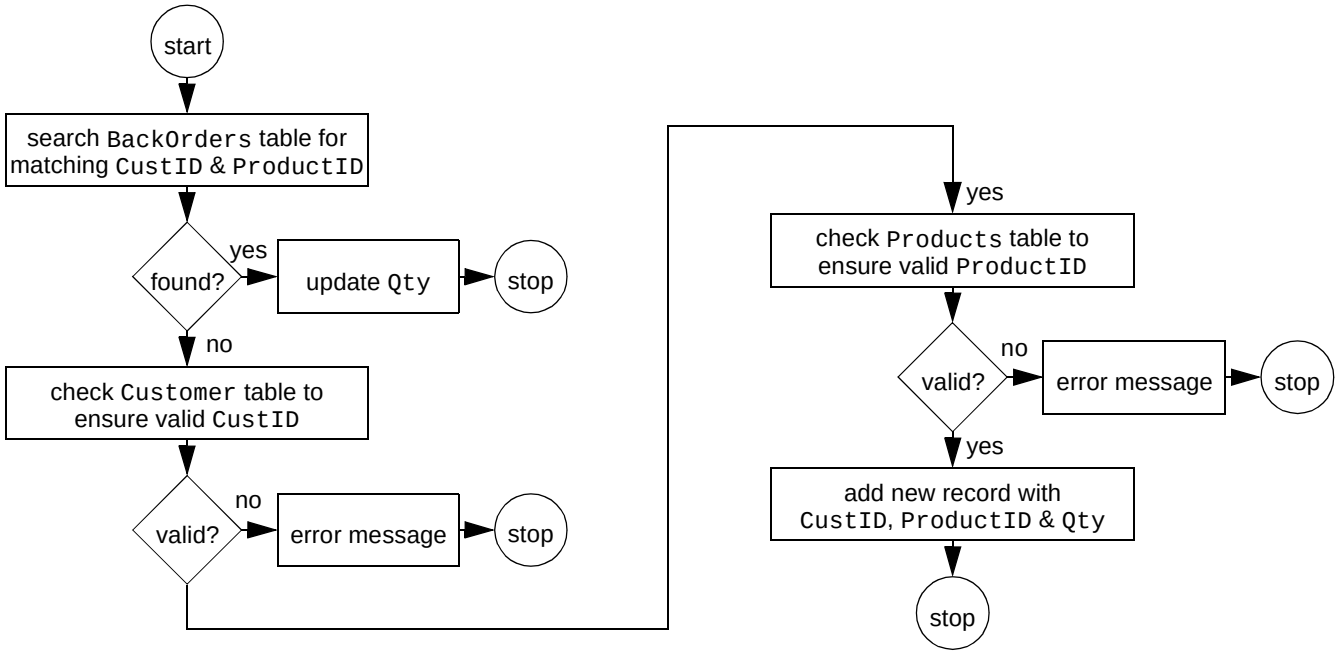


FIGURE 15.16: Flowchart for the `BackOrderItem` subroutine.



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.