

Contents:	Page
Preface	
Why should you use this Trainer?	vi
Pre-requisite knowledge	vi
Why use VBA?	vii
Other sources of information	vii
 <u>PART 1 – BASICS</u>	
1.1	1
Some Terminology and Explanations	1
1.1.1 Form, Report and Access Modules	1
1.1.2 Procedures – Sub and Function	2
1.1.3 Properties and Methods	2
1.1.4 Event-Based Programming	3
1.2	3
A First Look at a Code Module	3
1.3	5
The Help Systems	5
1.4	6
Creating and Debugging a simple Function	6
1.4.1 Create a simple function	6
1.4.2 Compiling your code	8
1.4.3 Debugging your code	8
1.4.3.1 The Debug Toolbar	8
1.4.3.2 Using the Immediate Window	8
1.4.3.3 Breakpoints, Stepping into code, watching values	10
1.5	10
Using a Function in Queries	10
1.6	12
Running a Query from a Command Button	12
1.7	14
Creating & Using Message & Question Procedures	14
1.7.1 Sending a Message to the screen	14
1.7.2 Asking a Question	17
1.7.3 Getting a Value from the screen	17
1.8	18
Documenting your Code	18
1.8.1 Using a standard form	18
1.8.2 Using a database	18
1.9	19
Exercises	19
1.9.1 myDisplayWarningMessage sub procedure	19
1.9.2 Improving the myUpdateFee function	19
1.9.3 Running an Update Query	20
 <u>PART 2 – USING EVENT CODE ON FORMS – DATA MAINTENANCE</u>	
2.1	21
Introduction	21
2.2	23
Viewing Data	23
2.2.1 Setting a form default	23
2.2.2 Using a View Command Button	24
2.2.3 Setting a default for each new record	24
2.2.4 Using a common procedure	25
2.3	25
Editing Data	25
2.4	25
Showing which is the active button	25
2.4.1 By changing text colour on command buttons	25
2.4.2 By using labels to simulate raised/sunken buttons	27
2.4.3 By using labels to look like hyperlinks	28
2.4.4 Some useful command button properties	29
2.5	30
Saving Records	30
2.5.1 Using a Save button	30
2.5.2 Using the Form_BeforeUpdate event	31
2.5.3 Using the Form_AfterUpdate event	32
2.6	32
Adding and Deleting Records	32
2.6.1 Add a New Record	32
2.6.2 Delete an Existing Record	34
2.7	35
Exercises	35
2.7.1 Buttons to Renew Membership and Change Address	35
2.7.2 Use Cancel Button on Membership Form	35
2.7.3 Stock Levels Form	35
2.7.4 Enabling/Disabling Close button	35

Continued...

Contents:	Page
<u>PART 3 – USING EVENT CODE ON FORMS – MISCELLANEOUS FEATURES</u>	36
3.1 Introduction	36
3.2 Automatic Calculations	36
3.2.1 Stock Form	36
3.2.2.1 Value of Each Stock item	36
3.2.2.2 Highlighting low stock	38
3.2.2.3 Adding Stock	38
3.2.2 Showing the Member's Age on the Membership Form	39
3.2.2.1 A myCalculateAge Function	39
3.2.2.2 Showing the Age on the Form	40
3.2.3 Showing the Category type on the Membership Form	42
3.3 Validations	43
3.3.1 Field Validations	43
3.3.2 Form Validations	45
3.3.3 Parameter Validations	46
3.3.3.1 Single parameter	46
3.3.3.2 Two parameters for a value range	49
3.4 Searching for Records	51
3.4.1 Replacing the Navigation Bar Functions	51
3.4.1.1 Next/Previous Records	51
3.4.1.2 Count of Records	51
3.4.2 Looking for a Particular Record	53
3.5 Applying a filter to the form (with a count of records)	55
3.5.1 Filtering on a text (string) field with wildcard	55
3.5.2 Filtering on a text(string) field for an exact match	57
3.5.3 Filtering on a numeric field	57
3.5.4 Filtering on a Yes/No field	58
3.5.5 Filtering on a date field	58
3.5.6 Removing a filter	58
3.5.7 Combining filters	58
3.5.8 Filtering on Null values	59
3.6 Using Combo and List Boxes on Forms	59
3.6.1 Some useful Combo and List Box properties	59
3.6.2 Changing Combo Box contents at run time	60
3.6.3 Using a list box to select records and change contents at run-time	62
3.6.3.1 Create a form based on the Class list table	62
3.6.3.2 Add two list boxes in the form footer	63
3.6.3.3 Add a text box called txtLetters near to lstMember	63
3.6.3.4 Create double-click event for lstMember	64
3.6.3.5 Check if member is already registered on the class	64
3.6.4 AddItem and RemoveItem methods	64
3.6.4.1 Using AddItem	65
3.6.4.2 Using RemoveItem	65
3.7 Exercises	65
3.7.1 Implement Receive Stock function, with validations	65
3.7.2 Show the total stock value on the stock form	65
3.7.3 Function to calculate the number of years between any two given dates.	66
3.7.4 Filter on Sporting interests	66
3.7.5 Combine Filters	66
3.7.6 Filter by Sex	66
3.7.7 Check for missing required fields	66
3.7.8 Create myIsAlphabetic function	67
3.7.9 Check for existing bookings (prevent double bookings)	68
3.7.10 Show booking history per member	68
<u>PART 4 – USING EVENT CODE ON FORMS - MENUS</u>	69
4.1 Introduction	69
4.2 Creating and using a Main Menu	69
4.2.1 Starting a menu – dynamic date & time, day of week	69
4.2.2 Improve the menu appearance	70

Continued...

Contents:	Page	
4.2.3	Command button to load a form	71
4.2.4	Open menu automatically on start-up	71
4.2.5	Exiting the application	71
4.2.6	Control tips	72
4.2.7	Accelerator keys	72
4.3	Data Maintenance via a Sub Menu	72
4.3.1	Create a sub menu	72
4.3.2	Exiting from the sub menu	73
4.3.3.	Amend Membership form	73
4.3.3	Editing Membership records	73
4.3.5	Viewing Membership records	74
4.3.6	Adding new Membership records	74
4.3.7	Deleting existing Membership records	74
4.3.8	Try this	74
4.3.9	Opening the form with a particular record	75
4.4	Exercises	75
4.4.1	'Are You Sure?' procedure on exit	75
4.4.2	Sub Menu buttons for Renew Membership and Change Address	75
4.4.3	Show count of records added	76
4.4.4	Open SubMembership form for a particular member number	76
4.4.5	Provide data maintenance facilities for Stock via a sub menu	76
4.4.6	Using a system heading in a table	76
<u>PART 5 – USING EVENT CODE ON REPORTS</u>		79
5.1	Introduction	79
5.2	Report of members who joined 10 or more years ago	79
5.3	Changing the appearance of a field at run-time	81
5.4	Calculating and printing totals	82
5.5	Empty reports	83
5.5.1	Using the Report_NoData event and cancelling the report	83
5.5.2	Printing an 'empty' report	83
5.6	Using a query criteria parameter at run time	84
5.7	Changing the sort order at run-time	85
5.8	Suppressing detail lines	86
5.9	Exercises	86
5.9.1	Member report	86
5.9.2	Stock report	86
<u>PART 6 – EMBEDDED SQL</u>		87
6.1	Introduction	87
6.2	The RunSQL method of the DoCmd object	87
6.3	Adding a row to a table	89
6.4	Updating a row in a table	91
6.5	Creating and dropping tables	93
6.6	Deleting a row from a table	96
6.7	Exercises	98
6.7.1	Add values to Town and County combo boxes at run-time	98
6.7.2	Record class sales	98
<u>PART 7 – MISCELLANEOUS</u>		99
7.1	Introduction	99
7.2	Using tab controls on forms	99
7.3	Importing/Exporting spreadsheet data	101
7.3.1	The TransferSpreadsheet method	101
7.3.2	Importing, using a named range and named columns	102
7.3.3	Importing, using un-named columns	105
7.3.4	Importing, using a defined range	105

Continued...

Contents:		Page
	7.3.5 Exporting data	106
	7.3.6 Some errors you may encounter	108
7.4	Backups, compacting, etc	109
	7.4.1 Making backups	109
	7.4.2 Compacting your database (keeping the size down)	109
7.5	Linking to an external database (separating data from the rest)	109
	7.5.1 DIY method	110
	7.5.2 Using the Database Splitter Wizard	111
	7.5.3 Re-linking after a back-end database has been moved or renamed	111
	7.5.4 Multi-user access to a back-end database	111
7.6	Preparing your database for distribution	112
	7.6.1 Removing the database window and menu bars	113
	7.6.2 Creating MDE (Microkernel Development Environment) files	113
7.7	The WITH statement	114
	7.7.1 Using a single With statement	114
	7.7.2 Using nested With statements	115
	7.7.3 Using With statements with parameter objects	115
7.8	Exercises	116
	7.8.1 Member bookings tab on Membership form	116
	7.8.2 Record class sales	116
	7.8.3 New Chelmer application	116
	7.8.4 Use the With statement	116
<u>PART 8 – WORKED EXAMPLES OF BOOKING PROCEDURES</u>		117
8.1	Introduction	117
8.2	Example 1 – Member Bookings for Courts	118
	8.2.1 Create table of booking times	118
	8.2.2 Start the Court Bookings form	118
	8.2.3 Create Query to see Court availability	119
	8.2.4 Use the Outer Join query for a form list box	120
	8.2.5 Do more on the CourtBookings form and the Court1 Availability query	121
	8.2.6 Viewing Court availability	122
	8.2.7 Making a member booking for a Court	124
	8.2.8 Preventing double-bookings	126
	8.2.8.1 Using DCount	126
	8.2.8.2 Setting a unique index	127
	8.2.9 Deleting a member booking	128
	8.2.10 Finally...	129
8.3	Example 2 – Class Bookings for Rooms/Halls	131
	8.3.1 Start the Class Bookings form	131
	8.3.2 Selecting a class	132
	8.3.3 Creating a BookingDate table	134
	8.3.4 Seeing Class availability	135
	8.3.5 Making the Bookings and checking for double-bookings	136
	8.3.6 Finally...	137
8.4	Example 3 – Make bookings using a 'diary page' grid	138
	8.4.1 Creating a Crosstab query	138
	8.4.2 Create the diary page form	140
	8.4.3 Specify a booking date via a parameter	140
	8.4.4 Trap the error where the Crosstab columns are missing	142
	8.4.5 Using Conditional formatting for the booked/free slots	143
	8.4.6 Making Bookings	144
	8.4.6.1 Making a start	144
	8.4.6.2 Setting up a Bookings form	145
	8.4.6.3 Making member/Class Bookings	146
	8.4.6.4 Using a list box for the Membership/Class No	147
	8.4.7 Deleting Bookings	147
	8.4.8 Finally...	149
8.5	Exercises	149
	8.5.1 Member Bookings	149
	8.5.2 Class Bookings	149
	8.5.3 Using a 'diary page' grid	149
	8.5.4 Recording attendance	150

Continued...

Contents:	Page
Appendices:	151
A Events Overview	151
B Coding Standards	152
C Common Coding Errors	153
D Code Documentation Form	154
E Some naming conventions for variables, procedures, etc.	156
F Some basics of programming	157
F.1 Declaring and using variables and constants	157
F.1.1 Datatypes	157
F.1.2 Variables and constants	158
F.1.3 Scope	158
F.1.4 Public/Private	159
F.1.5 Arrays	159
F.2 Assignment statements	160
F.2.1 General format	160
F.2.2 Literals	160
F.3 Control Constructs	161
F.3.1 Sequence	161
F.3.2 Selection	161
F.3.3 Iteration	162
F.4 Procedures and parameters	163
G Overview of SQL	164
G.1 CREATE TABLE	164
G.2 ALTER TABLE	164
G.3 CONSTRAINT	165
G.4 INSERT	165
G.5 CREATE VIEW	165
G.6 SELECT – inner join	166
G.7 UNION	166
G.8 SELECT – outer join	166
G.9 COMMIT	167
G.10 GRANT (and REVOKE)	167
G.11 Aliases	167
H Built-in functions	168
H.1 Date and time functions	168
H.2 String functions	168
H.3 Maths functions	169
H.4 Financial functions	169
H.5 Miscellaneous functions	170
H.6 Domain aggregate functions	170
H.7 Type conversion functions	171
I The Forms Collection	172
J Some useful DoCmd methods	173
Index	174

A "Further VBA" Trainer is also available. Copies are available from the Student Advice Centre at Leicester. Students at Associate College Centres should enquire of Centre staff.

Topics covered include:

- *password protection*
- *accessing user login*
- *error trapping and custom error messages*
- *using Data Access Objects (DAO)*
- *creating a 'diary page' grid to make, view and delete bookings using DAO code and arrays*

PREFACE

Why should you use this Trainer?

If you intend using MS Access for a project module (HNC, HND or Degree) then you will almost certainly need to use some VBA to improve on the standard features provided by Access.

The purpose of this Trainer (and the "Further VBA" Trainer also available) is to provide a self-study opportunity for students who wish to extend their knowledge of Microsoft Access features and learn how to use VBA. Code in this document was written and tested using Access 2002 with Access 2000 file format (the default format).

Many textbooks list code elements of VBA and provide little, often disjointed, examples of code. Students, who may not have a detailed and experienced background of commercial programming, often find it difficult to make the connection between little examples and what they want to do in their assignments and projects. Building and programming systems involves problem-solving techniques; programming involves a great deal more than just writing code. The main sticking point for many students is in actually understanding how to use the tools available in order to solve a business problem. VBA is demonstrated here by showing the practical applications of techniques for situations that arise in applications, by building up code to improve a database created purely by the standard 'point-&-click' facilities of Access.

VBA is a fourth generation programming language (4GL) provided with Access. Previous versions of Access used macros to provide some custom facilities, but this is now being phased out in favour of VBA and will not be covered here (see Appendix J). VBA can also be used within other Microsoft Office applications, such as Word and Excel; note that the 'A' stands for 'Applications', not for 'Access'. By working through the examples here, you will become familiar with using VBA, the Debugger and Help system, and should gain the confidence and knowledge to be able to explore VBA more fully yourself for items that are not covered here. Using this Trainer should also help you to understand event-based programming; see also Appendix A.

Most of the code used in this document is pretty simple; just small pieces of code which can make a big difference to the professional 'look and feel' of your system, and with which most students should be able to cope. There is a "Further VBA" Trainer which introduces more advanced coding.

You are expected to work through all the examples shown. Later examples will depend on the earlier examples having been done. Each section ends with a selection of exercises, which may also depend on earlier examples and/or exercises.

Pre-requisite knowledge

This Trainer assumes that...

- ...you have a good foundation knowledge of MS Access 'point-&-click' facilities, such as given by the book *Smart Guide to Access 2000, Further Skills*, by Nat McBride, published by Thompson. This book is used on several database modules at DMU. Throughout this Trainer, the textbook will be referred to as *McBride*.
- ...you know how relational databases are designed and constructed (this is useful but not essential for this Trainer).
- ...you have created the tables for the Chelmer Leisure Database as used in the above-mentioned textbook, and have created (or can create) the Membership and Membership Category forms.
 - o The Chelmer Leisure database used in *McBride* has some oddities of design, but is used here exactly as it is in the textbook, so that you can continue to use the database that you may have created already at DMU.
- ...you have a good basic understanding of programming and coding principles, including:
 - o Datatypes, variables, scope, arrays
 - o Assignment statements, strings, literals, operators, expressions
 - o Control constructs such as IF, WHILE, CASE
 - o Creating and using procedures and functions, using parameters
 - o SQL (see also Appendix G).

See Appendices B, C, E & F for further information and an overview of some programming topics.

Note that the purpose of this Trainer is not to show you how to code everything you will need in Visual Basic, but to get you started with using VBA as a 4GL with Access.

You should also note that, when creating a database, there is **no substitute for a good design**. Time spent analysing user and data requirements and deriving tables is time very well spent. Before creating your tables, check that all the known (and some predictable) requests for tasks and reports can be satisfied (i.e. verify your design). The earlier an error is spotted (and, of course, removed) the less time has to be spent later trying to fix the problem. Once you are satisfied that your tables are correct, you can plan the structure of your forms, how they will be used (this helps when planning buttons, combo boxes and the like), and plan the layout of your reports, perhaps trying prototypes to test out ideas. Remember that a database is not just about the data that is in it but is about how the user will use it and the information and functions that the database can provide to the user. There is not much point putting data into a database if you are not going to do anything with it!

Design is also necessary with code, as you need to work out the logic of what you want to do before you code it. You must then test it thoroughly. At various points throughout this document pointers to help you with code design and testing are given, to try to encourage you to think 'design, code, and test' rather than simply 'code'.

Why use VBA?

By using the basic facilities provided in Access (such as the menus, property boxes, wizards) you can create a simple database pretty quickly. But Access is a general software package and there are bound to be things that you want to be able to that do that Access does not do, or does not do in the way that you want. To create a really professional application (and to do a good project) you will need to code in VBA.

Some examples of occasions for which you will need VBA code are:

- To perform a calculation that is too complex for a built-in expression.
- To avoid duplicating code in several places, by writing a standard procedure or function that can be re-used whenever wanted.
- To provide graceful and appropriate error handling, with more useful and relevant messages for the user.
- To document your database, as code can be printed and maintained. (It is very easy to forget how you created a complex feature if you did it via standard Access facilities).
- To give a more professional and personal look and feel to your application.
- To assist data entry for the user (perform calculations automatically, perform complex validations, anticipate what the user may wish to do, enable/disable form controls, etc).
- To display information messages to the user (e.g. 'Record updated OK').
- To add code behind buttons on forms, create menus, etc.
- To prevent users from changing data when they do not mean to (the Access default is to allow users to change all data).

And much more besides...

Other Sources of Information

Check out *Various Teaching Materials for MS Access and VBA* on my home page at <http://www.cse.dmu.ac.uk/~mcspence/Access>. This site has answers to questions asked by students studying database modules, when doing HNC, HND and Final Year projects and when on placement. There are also several example databases there, most arising out of situations that have arisen during project development or placement work; these are not reproduced in this document (it seems pointless to duplicate things). The site also has links to other sites that you may find useful.

Any Comments?

If you have any comments about this Trainer, have spotted any errors or have any suggestions for improvement, please email mary.spence@dmu.ac.uk

PART 1 – THE BASICS

In this part of the Trainer you will see...

- ...that VBA code is organised into modules. Each form and report has its own module for event code created automatically by Access. Developers can also create modules for other code.
- ...that modules consist of a series of procedures (subs and functions, and can contain global or local constants or variables.
- ...that VBA involves event-based programming, where code is triggered by an event such as the user clicking on a form object.
- ...how to code, compile and run a procedure in an Access module and to see VBA code generated automatically by the Access Command Button Wizard. Such code can be edited to allow custom features.
- ...that objects on forms and reports have specific events attached to them.
- ...that event code can be created/opened by clicking on the 'build' icon in the property box, or via the drop-down lists at the top of the module code window.
- ...that form and report modules can contain private procedures for use only within that code module.
- ...that procedures to be used from more than one place are best coded in a separate Access code module (good practice).
- ...that reusable procedures can be very useful, and will repay any initial work needed to set them up.
- ...some basic features of the Access Debug facility: breakpoints, stepping through code, looking at data values.
- ...that user-defined functions can be used (indeed, reused) directly in queries.
- ...how to display messages and ask questions.
- ...a simple method of asking for a value from the user.
- ...suggestions for documenting your code.

See Appendix A for a list of events that occur for Access objects.

See Appendices B and C for some help and advice when coding.

See Appendix D for a blank code documentation form and examples of use.

See Appendix E for some suggestions for naming conventions for variables, procedures, etc.

See Appendix F for an overview of some basics of programming.

See Appendix G for an overview of SQL.

See Appendix H for details of built-in functions.

1.1 Some Terminology and Explanations

1.1.1 Form, Report and Access Modules

All VBA code is organised (contained) in Modules.

- Form Modules – there is one module per form with code applicable to that form.
- Report Modules – there is one module per report with code applicable to that report.
- Access Modules – these are used for free-standing code, such as general-purpose procedures and functions. You can have as many of these as you like in a database.

Form and report modules are created automatically by Access.

You create Access modules, as you need them.

1.1.2 Procedures – Sub and Function

Procedure “A named sequence of statements executed as a unit. For example, **Function, Property, and Sub** are types of procedures. A procedure name is always defined at module level. All executable code must be contained in a procedure. Procedures can't be nested within other procedures.”
 Extract from Access 2002 help.

A module consists of a series of subs or functions.

- A Sub (often referred to just as a Procedure) is a piece of code that performs a specific task (known as a subprogram or subroutine in some other languages). It is referred to by name when it is needed; this is known as calling or invoking the procedure. It may use arguments (also known as parameters) if needed.
- A Function is a particular type of procedure that simply returns a value. It may optionally take arguments as input values.
 - o There are several standard (built-in) functions in Access, such as Date(); see Appendix H. When you code =Date() for a database field, you are calling the Date function, and storing the value returned in the field you have specified. The empty brackets after the function name show that this function does not use any arguments.
- Subs and functions within modules can be Private or Public – Private means that they can be used only within the module in which they are written; Public means that they can be used by other modules as well (Public procedures/functions are normally written in an Access Module).
- For further information use the VBA Help Answer Wizard with the text “calling sub and function procedures”. Also see Appendix F.4.

1.1.3 Properties and Methods

You will be aware that forms, reports, and the objects on them, have Properties. For example, a command button has an Enabled property that can be set to Yes (True) or No (False) to determine whether the button is to show or be greyed-out.

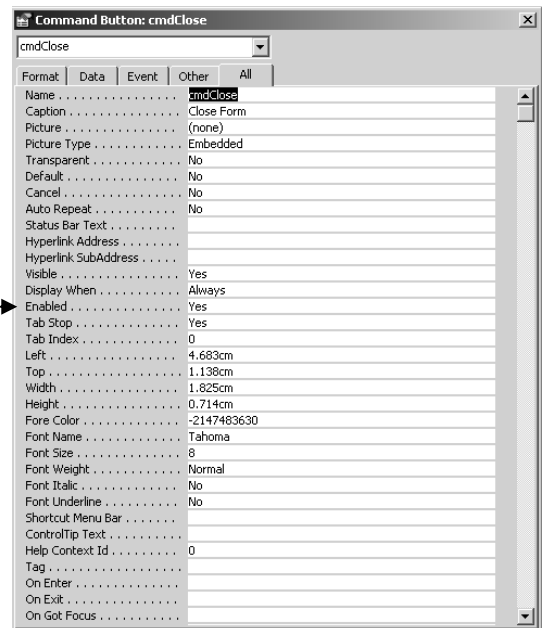


Fig 1.1.1 Properties Window

Fig 1.1.1 shows some of the properties for a command button to close a form. The button has been given the name of cmdClose. (See Appendix E for examples of naming conventions).

Most of these properties can also be set using VBA code:
 cmdClose.Enabled = True (OR False)

The general format for assigning a value to a property is:
 ObjectName.Property = Value

Note the use of the dot (.) operator to specify the Property for the ObjectName. Access Help has the following information:

The . (dot) operator

The . (dot) operator usually indicates that what follows is an item defined by Microsoft Access. For example, use the . (dot) operator to refer to a property of a form, report, or control. You can also use the . (dot) operator to refer to a field value in an SQL statement, a Microsoft Visual Basic for Applications method or a collection.

When you code these statements in VBA, the VB Editor will provide useful prompts after you type the . (dot), as shown in Fig 1.1.2.

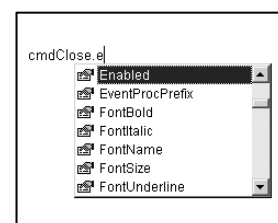


Fig 1.1.2 Prompt for property(note icon)

Forms, reports and the objects on them also have *Methods*. The format for specifying a method also uses the dot operator:

ObjectName.Method {optional Arguments}

The VB Editor will also provide prompts for methods, as shown in Fig 1.1.3. The SetFocus method will move the cursor to the object named (useful when validating a form field if you want the user to enter the value again).

e.g. [Category No].SetFocus will move the cursor to the Category No field.

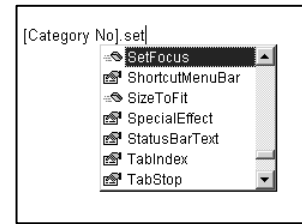


Fig 1.1.3 Prompt for method (note icon)

Methods for forms and reports do not always need to specify the ObjectName. If it is omitted, then the current form or report is assumed. For example, if you wish to *Repaint* a form (in order to show details of records that have been updated in tables, list boxes, and combo boxes etc) you can code

Repaint OR *Me.Repaint*

Me means the current form/report. If you code using *Me* then you will get the prompt box to help you.

1.1.4 Event-Based Programming

With VBA in MS Access you will be performing *event-based programming*. An *event* is simply something that happens in response to something else, such as an action by the user (clicking on a command button, for example). The user is largely in control of the order of events. This is quite different from procedural programming, where the program is in charge of the order of events.

A single action, whether by the user or by VBA code, can trigger other events that neither you nor the user may be aware of. It is important that you, as a programmer, are aware of this. For further information see MS Access Help (oddly, not VBA Help) and type *order of events* into the Help Answer Wizard. If you view all, and wish to print the information out, it takes four pages.

There are *many types of event*. See Appendix A for an overview of the events for forms, reports, command buttons, combo boxes, etc. The full list of programmable events for each object is shown on the Event tab of the object's property box.

1.2 A first look at a code module

There are several ways of accessing code modules to see, add, amend or delete code:

- For code for a form or report:
 - o Open the form or report in design mode and click on the *view code* icon (also available via the *View* menu). This method is useful if you wanted to add a procedure to be called from an event procedure or if you wanted to look at, or print, all the code.
 - o Open the form or report in design mode and double click or right-click on the object name for which you want to add an event, and then select the event type from the property box. This method is useful to get directly to existing code or for quick creation of the required header and footer for a new piece of event code.
- For free standing code in an Access standard module, click the modules tab in the database window, then create a new module or open an existing one, as appropriate.

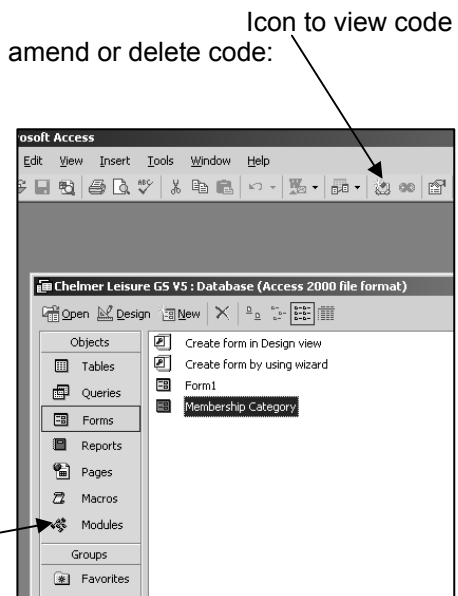


Fig 1.2.1 How to view a code module

Open your Chelmer Leisure database and look at the form code module for the Membership Category form. This form is created in Unit 14 of *McBride*. You will now see the code in the module as shown in Fig 1.2.2.

The form name is highlighted in the little window on the left. You can open other code windows from here by double-clicking on the relevant name.

The code module is empty initially apart from two standard lines at the start, which should always be there as defaults. Once you have set the options they will apply to all new code modules created, but will not apply to existing code; you will have to code them manually in existing code modules.

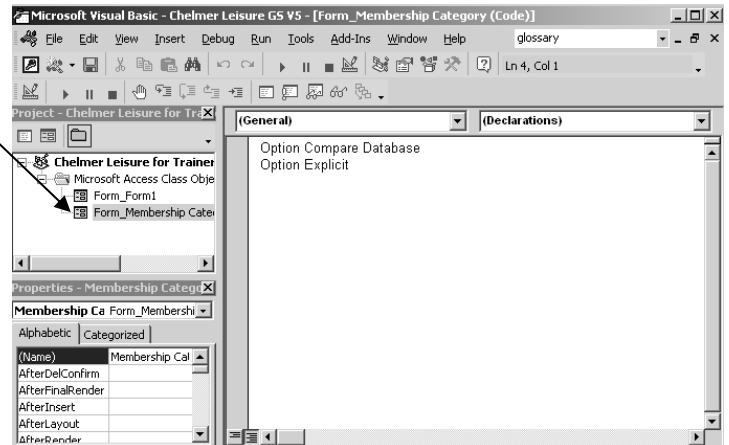


Fig 1.2.2 Code Module Window

You may need to set Option Explicit as a default. In Access 97 it was standard, but in Access 2000/2002 it does not appear to be. Do this via the code window *Tools* menu, choose *Options*, then set *Require Variable Declaration* on the *General* tab as shown in Fig 1.2.3.

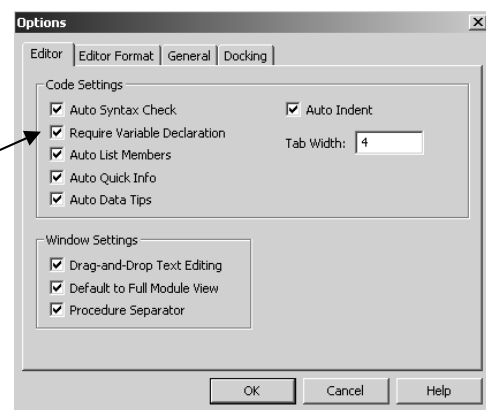


Fig 1.2.3 Setting Option Explicit as a default

Access has a very useful *Help* facility, which you will use now to see what these two lines mean. Move the cursor to the word *Compare* and press keyboard function key F1. You should now see a Help screen explaining this word. Similarly check *Explicit* and *Option*. Now that you know of this facility, do be sure to make good use of it to check anything you do not understand. This document will assume that you will do so.

In particular, note the meaning of *Option Explicit*. I would strongly advise that you **always** set this option, otherwise a simple spelling error would cause the declaration of a new variable, and errors like that are very hard to find.

Click on the arrow to the right of the field headed *General* in Fig 1.2.2. You will see a drop-down list of all the objects on the form. This list can be used when creating event code for the selected form object (and for checking the names of these objects). Click now on the field *Category_Type*. You will see the following code in the module window:

```
Private Sub Category_Type_BeforeUpdate(Cancel As Integer)

End Sub
```

These lines are the first and last lines for a sub procedure for a *BeforeUpdate* event on the field *Category Type*. The procedure has a single argument with the local name of *Cancel* and of data type *Integer*. Access writes these lines for you automatically, ready for you to enter the code that you want between them. If the code already existed, Access would assume that you wanted to alter the code, and would position the cursor ready in the existing code. If you wanted to write code for another event for this field, simply click on the arrow by the field to the right (*Declarations*) and choose the event that you want. See also Appendix A. Use the F1 key to check the meaning of terms in this code. Note especially the meaning of *Private*; Access has assumed that the procedure will be used only from this module. Close the code module window and the form without saving anything.

Code can also be accessed via the property box for an object. With the Membership Category form open in design mode, double click on the Category No field. The property box for the field is now displayed. Click on the Event tab (and see the full list of allowable events for this field) then on the Click event. Now click on the Build icon to the right (the icon with three dots) and choose 'Code Builder' from the list shown. See Fig 1.2.4.

You are now in the code module positioned ready to enter code in a sub procedure for the Click event on the field Category No.

```
Private Sub Category_No_Click()

End Sub
```

We are not going to enter any code just yet, so close the code window and the form without saving anything.

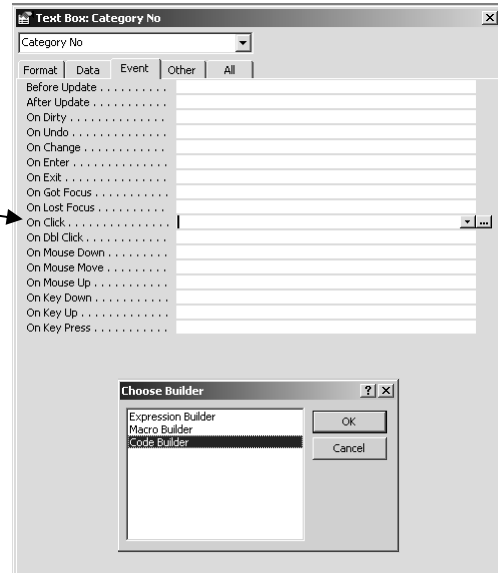


Fig 1.2.4 Creating code for an event

Finally, a look at the code window for an Access module. At the main database window, click the modules tab and then click on New. You will have a window that looks just like the window shown in Fig 1.2.2 above, but there are some differences:

- The code module name is probably something like Module1. You will be prompted for a name of your own when you save the module.
- Clicking on the General box does not bring up a list of objects (as this is an Access module and is not attached to anything in particular).

You can create as many Access modules here as you want. It is useful to group all like procedures together, for example, a module for messages, a module for calculations, etc. Close the window without saving.

1.3 The Help Systems

In Access 2000/2002 there are two different Help systems. You should all be familiar with the system available from the database window. VBA Help is available via a code window.

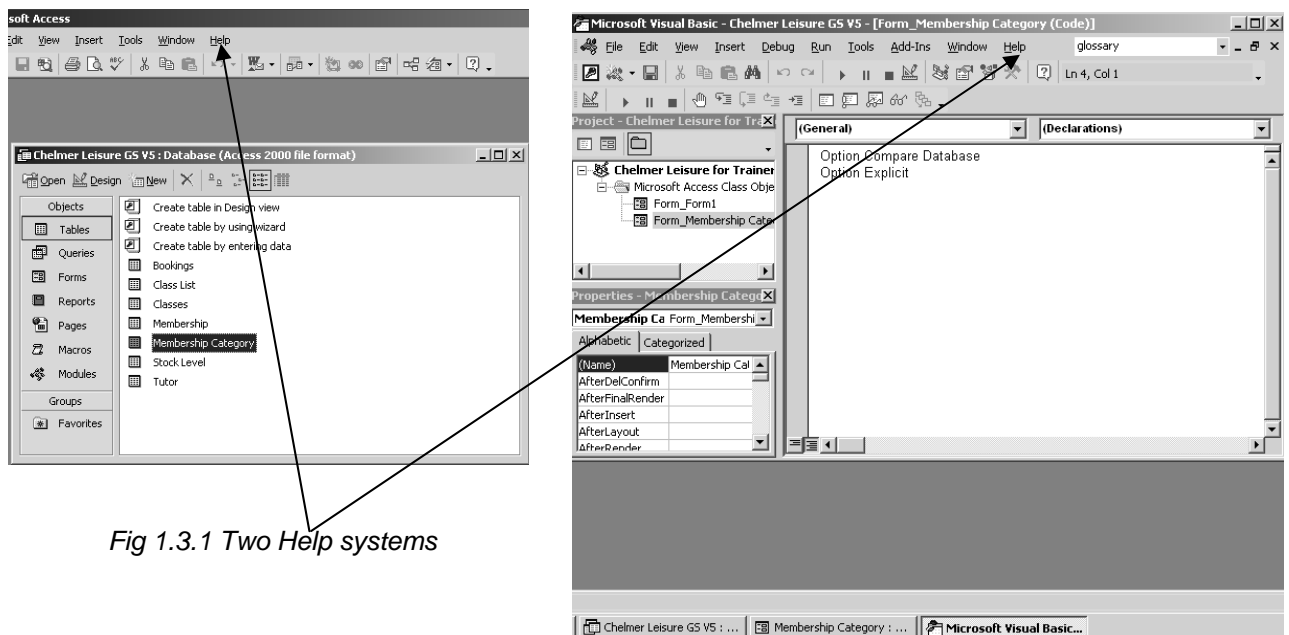


Fig 1.3.1 Two Help systems

The formats of the Help windows, and how they work, are the same for the two systems; it is the Help content that varies. The title bar at the top of each window will tell you which system you are in (Microsoft Access Help or Microsoft Visual Basic Help). If you cannot find what you are looking for, check that you are using the right Help system.

Access has an extensive Help system, part of which includes:

- A full glossary of all terms (so no glossary is included with this document).
 - o Access Help links to a web page. This covers 14 pages if you print it off.
 - o VBA Help refers to ADO (Active Data Objects). This topic is not covered in this Trainer.
- A full list of all VBA statements, Functions, Constants, etc.
 - o Look at *Visual Basic Language Reference* on the Contents tab of VBA Help in Access 2000/2002.
 - o Most of the Help references have examples of use.
- Instructions on how to debug VBA code. See also section 1.4 below.

You are strongly advised to make full use of these Help systems. In fact, this Trainer assumes that you will do so. Help screens can be printed via *Options*.

When you open a code window you are positioned in the built-in VBA Editor, and in a separate window; see the bar at the bottom of the right-hand screen in Fig 1.3.1. You can move between the two windows as you wish.

1.4 Creating and Debugging a simple Function

Now it's time for you to write some code. The Membership Category table holds a list of categories and annual fees. It is not unusual for lists of this sort to be amended from time to time; we are all used to the notion of fees increasing. With a small list such as this it is a fairly simple matter to amend the fees via the form or directly into the table. However, many systems will have much larger numbers of records to cope with, so it could be useful to have a procedure to update such tables. So we will now create a function to update the fees and then reuse it. This will be somewhat of a sledgehammer to crack a very small nut, but the general principles learned here will be of use in your later applications.

1.4.1 Create a simple function

Suppose that the Chelmer Leisure Centre wishes to increase the fees as follows:

Category 1 and 2	- increase by 5%
Category 3	- increase by 7.5%
Category 4	- increase by 10%
Category 5 and 6	- no change

Open the code module for an Access Module (see section 1.2 above if you cannot remember how to do this). Type in the code that is shown in the box in Fig 1.4.1 and save the module with the name Calculations.

The function header

```
Public Function myUpdateFee(prmFee As Currency, prmCategory As Byte) As Currency
```

shows the function name `myUpdateFee`, two arguments (each with the prefix `prm` to identify them as parameter values to the reader of the code; see Appendix E), their data types, and the fact that the function returns a Currency value. The function is `Public` so that it can be accessed from outside this module.

Tip: If you prefix all your own procedure names with the letters 'my' then you are unlikely to use a name that is also the name of one of Access's own procedures. If you do use the same name as Access, then you will get the error "Compile Error: Expected variable or procedure, not module".

Comments can start with `Rem` or an apostrophe (`'`). Both methods are shown here. They show with a green font in the code window (but print out as black, which is a shame).

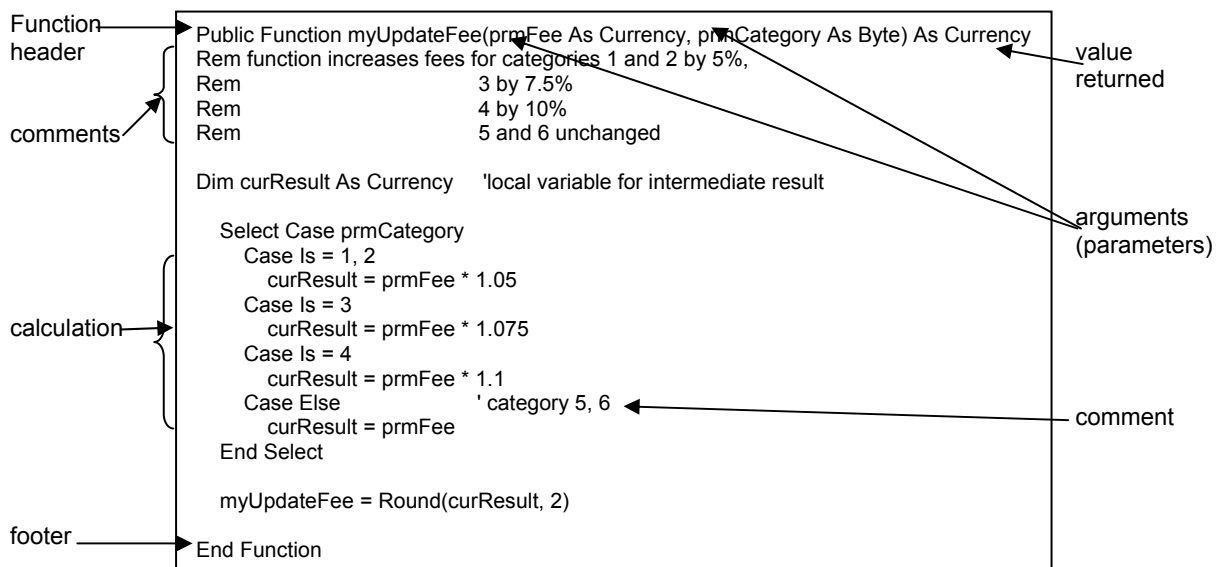


Fig 1.4.1 Code for myUpdateFee Function

A CASE statement is simply another way of coding IF ... ELSE ... (See Appendix F.3.2)

The statement above is equivalent to:

```
IF prmCategory = 1 OR prmCategory = 2 THEN
    myUpdateFee = prmFee*1.05
ELSEIF prmCategory = 3 THEN
    Etc....
```

If you need to perform multiple checks on the same field, then a CASE statement is often neater than a series of IFs.

The statement `curResult = prmFee * 1.05` is an assignment statement (see Appendix F.2). The result of multiplying the value in `prmFee` by 1.05 is assigned to the variable `curResult`. At the end of the function the statement `myUpdateFee = Round(curResult, 2)` assigns a value to the name of the function, so that this is the value that will be returned by the function. `curResult` is a local private variable; it exists only within this procedure.

It is worth noting that Currency datatypes hold values to 4 decimal places, not to 2 as you might expect; but this is not as illogical as it may seem at first, as it means that calculations are more accurate. Thus, calculations on these fields may not result in an exact number of pence. Access 2000/2002 has a built-in function `Round` which is used here to round the final result to two decimal places. (This function was not available in Access 97). Use the Help system (position cursor to the word `Round` and press F1) to have a look at how this function works. While you were typing the code, you may have noticed the helpful prompt that the VBA Editor gave, as shown in Fig 1.4.2. This prompt applies when using built-in functions and when using functions that you have written yourself.

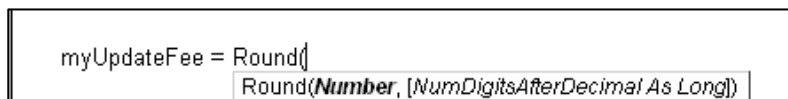


Fig 1.4.2 Prompt box when using a function

But there is a very important omission from Access VBA Help; the `Round` function works on a round-to-even basis, not the way that you may think it works.

Example 1. `Round(3.1275, 3) = 3.128,`
`Round(3.5, 0) = 4`

This is what you would normally expect; 5 is rounded upwards.

Example 2. `Round(3.1265, 3) = 3.126,`
`Round(2.5, 0) = 2`

This may not be what you would expect, as 5 is now rounded downwards.

In each case, the rounding is such that the last digit in the result is an even number.

For fuller details see <http://www.cse.dmu.ac.uk/~mcspence/Access.htm>, go to the Frequently Asked Questions page and look at VBA FAQ 13.

1.4.2 Compiling your code

Your code will be compiled to a certain extent as you type it, and errors will be reported as they occur. The relevant code is highlighted in red font. See Fig 1.4.3.

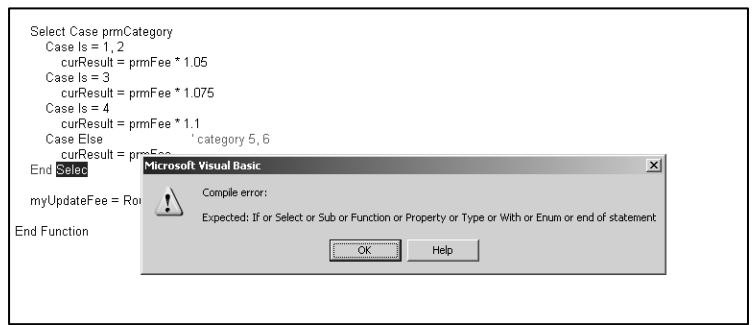


Fig 1.4.3 Compilation error when typing code

Not all compilation errors are picked up this way, but you can compile all the code via the *Debug* menu. See Fig 1.4.4.

It is not essential that you do this, but if you don't you will find that you get the compilation errors occurring when you attempt to run the code, as the code is also compiled at run-time.

Try making deliberate errors in the code to see what compilation errors you get (this is a good learning exercise!).

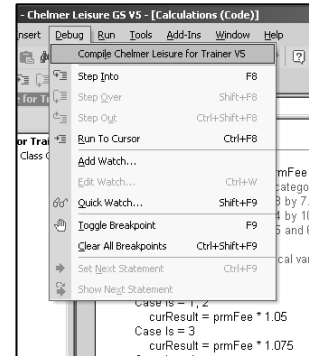


Fig 1.4.4 Compile the code

1.4.3 Debugging your code

Before you use your new myUpdateFee function in its intended context, you need to test it. Access has a very useful (and easy to use) Debug facility. Students are often reluctant to use Debuggers (perhaps because they are yet something else to learn?) but Debuggers are an invaluable tool for every programmer (especially if you hate programming!).

1.4.3.1 The Debug Toolbar

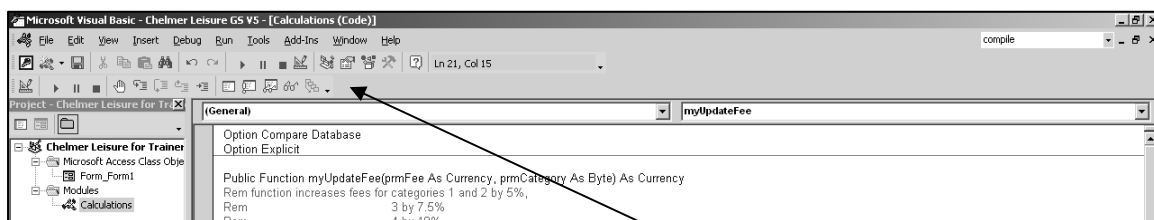


Fig 1.4.5 The Debug Toolbar

You may not have this toolbar in your code window, but you can add it via *View*→*Toolbars*. It can be customised just like any other toolbar via *View*→*Toolbars*→*Customise*.

To see a description of what each icon represents, go to VBA Help and type *Debug Toolbar* into the answer wizard. Do this now. You may find it useful to print the page out for reference.

1.4.3.2 Using the Immediate Window



Click on the icon for the Immediate Window and note the sub window that opens at the bottom of the code window. See Fig 1.4.6.

Type in the following code into this window and press enter:

```
?myUpdateFee(20,1)
```

This line means “run the function myUpdateFee with a fee value of £20 and category of 1”. Note that you must have a ? at the start of the line; you will get a compilation error message if you miss it off.

The function should return the value 21, which is a 5% increase on 20, as required. This value will be shown on the next line in the debug window. See Fig 1.4.6.

Experiment and test the rest of the function, for all the categories. What if there is a category 7 in the table? What will be calculated then? Try it and see. Is the answer correct? Can you suggest an improvement to this function? See test 7 of the test plan below and see also Exercise 1.9.2. The table in Fig 1.4.6 shows a possible test plan.

Testing Notes...

- ...you must try to test every path through the code. For this function, that means testing every branch of the Case statement in order to test each of the current categories and fees.
- ...you also need to think of ‘what if’ conditions, such as ‘what if the category value passed to the function was not in the expected range of 1-6?’ and test these conditions as well.
- ...if you make a change to your code then the function is different from before, however apparently minor the change was. This means that you must repeat all tests again at the end, just to check that earlier tests (done before the change) still work.

Test No	Data		Reason for test	Expected result
	Fee	Category		
1	25	1	Check calculation for each expected category and fee.	26.25 (25 * 1.05)
2	30	2		31.5 (30 * 1.05)
3	10	3		10.75 (10 * 1.075)
4	15	4		16.5 (15 * 1.1)
5	18	5		18 (no change)
6	20	6		20 (no change)
7	any	7	Category 7 does not exist in the Category table. What will the function do?	What do you think will be the result?

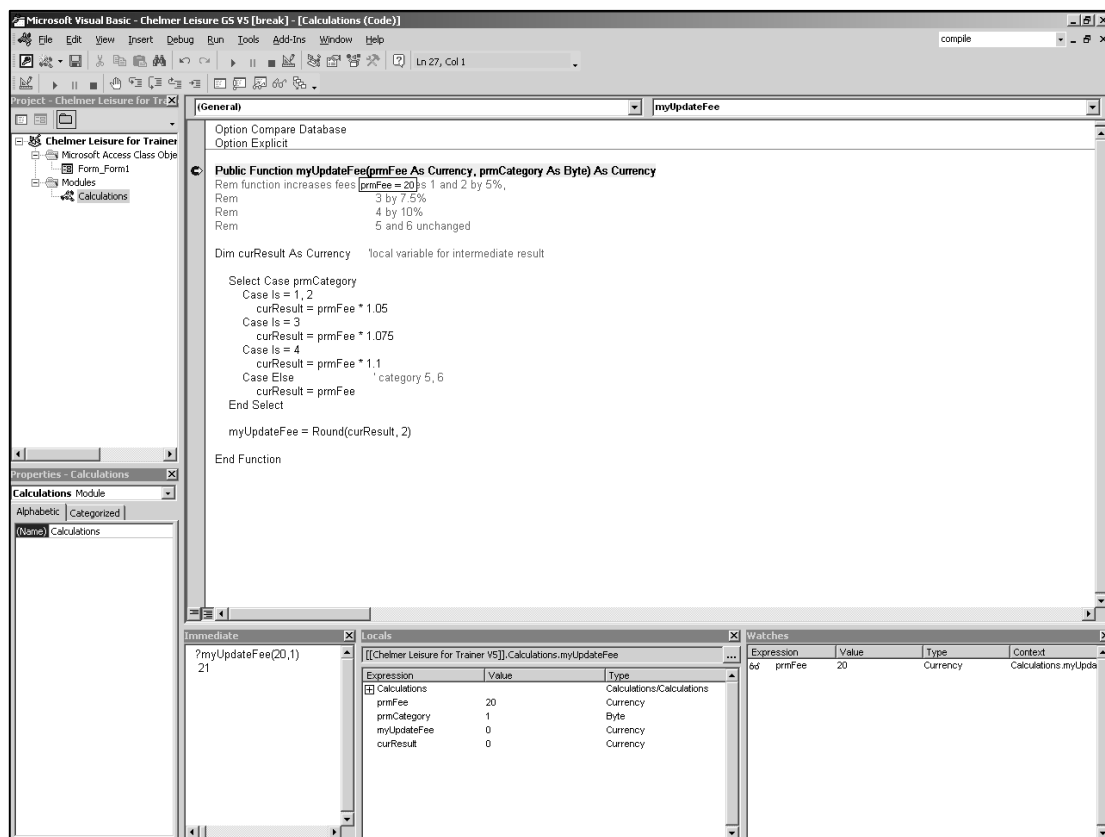


Fig 1.4.6 Testing code in the Debugger, showing the Immediate Window (left), Locals Window (centre) and Watch Window (right) and a simple test plan.

1.4.3.3 Breakpoints, stepping into code, watching values

Correcting coding errors can be very frustrating, especially if all you have to go on is knowledge of what went in and what came out. With the Debugger, you can step through code line by line, look at the values in data items, and spot just where a fault is (and then work out what is wrong and correct it). You will now try stepping through the code of your `myUpdateFee` function and looking at data values. Look at the code window of Fig 1.4.6.

- Go back to the code window, position the cursor on the function header and set a breakpoint. The code line will be highlighted to denote that a breakpoint has been set. (Simply click again to unset the breakpoint). Two ways to set a breakpoint are:
 - Click on the Toggle Breakpoint button (on toolbar or via *Debug* menu).
 - Click in the margin of the code line (where the coloured dot is).
- Go to the Immediate Window, retype `?myUpdateFee(20,1)` and press enter. (Or you could just position the cursor at the end of the existing line and press the enter key). You are now positioned inside the code window whilst the code is running (the breakpoint line highlight has changed colour). The execution of the code has stopped at the breakpoint, waiting for instructions from you. (That is what a breakpoint is used for – a point to break execution).
- Move the cursor over the name for the field `prmFee`. A box will appear (look at Fig 1.4.6) showing you the value in the field. You can use this facility at any time to check on values in data items. Check the values in `prmCategory`, `curResult` and `myUpdateFee`.
 - There is a Watch option that will keep values constantly in the Debug window for you to check. Move the cursor to the required item, click on the Watch icon and choose to Add the item chosen.
 - There is a Locals Window that will show all local values for a procedure/function. Click on the Locals icon.
- Click on the Step Into icon (on toolbar or on *Debug* menu). The cursor moves to the first executable statement. Successive clicks on this icon will step you through the code and you can see exactly what is happening and can see/watch the values in the variables. Close the window when you want to stop, and say 'yes' to confirm. Remove the breakpoint (click on the Toggle Breakpoint icon again).
 - To see the effect of the Round function try testing `?myUpdateFee(25,3)`
- Try experimenting with some of the other options provided in the Debugger. Note that you can set as many breakpoints as you like. Step Out will execute until the next breakpoint and then stop.

Finally, note that a function or sub procedure declared as `Private` can only be invoked from within the module within which it is declared. This also means that the Debugger cannot find it. In this case you get the message shown in Fig 1.4.7. You also get this error if you misspell a procedure name.

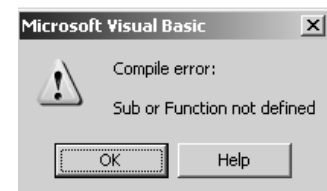


Fig 1.4.7 Message when function cannot be found

1.5 Using a Function in Queries

Before using the function to update the values in the table, it might be useful to see what the new values are going to be. So, create a query (call it Check Update Fees) to do this, as shown in Fig 1.5.1.

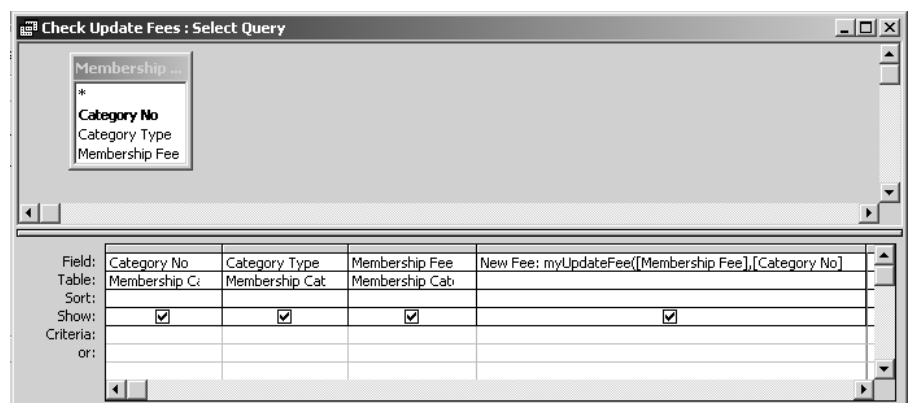


Fig 1.5.1 Check Update Fees query, using `myUpdateFee` function

This query has the first three columns taken directly from the Membership Category table. The fourth column is a calculated column called New Fee, where the function myUpdateFee is called, passing the values from the [Membership Fee] and [Category No] columns for the row as arguments to the function. The result of the function call is then put in the column when the query is run. See Fig 1.5.2.

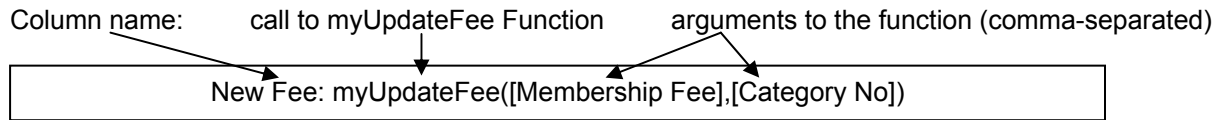


Fig 1.5.2 function call in 4th column of query

The general format of a function call is: *FunctionName (arguments, comma-separated)*

The function will thus calculate the appropriate new fee and return this value for the fourth column. Running the query will give the dynaset as shown in Fig 1.5.3.

Category No	Category Type	Membership Fee	New Fee
1	Senior	£25.00	£26.25
2	Senior Club	£30.00	£31.50
3	Junior	£10.00	£10.75
4	Junior Club	£15.00	£16.50
5	Concessionary	£18.00	£18.00
6	Youth Club	£20.00	£20.00
*		£0.00	

Fig 1.5.3 result of running the Check Update Fees query

If you wanted to step through the code and watch what happens for each row, simply set a breakpoint as shown in section 1.4.3.3, then run the query. You will be positioned back in the code window and can step through, look at data values, etc.

This query will not change the values in the Membership Category table, as it is just a select query.

In exactly the same way as above, the function myUpdateFee can be used in a query to update the Membership Category table. Look at Fig 1.5.4. Here the 'Update To:' row is simply a call to myUpdateFee as before: myUpdateFee([Membership Fee],[Category No])

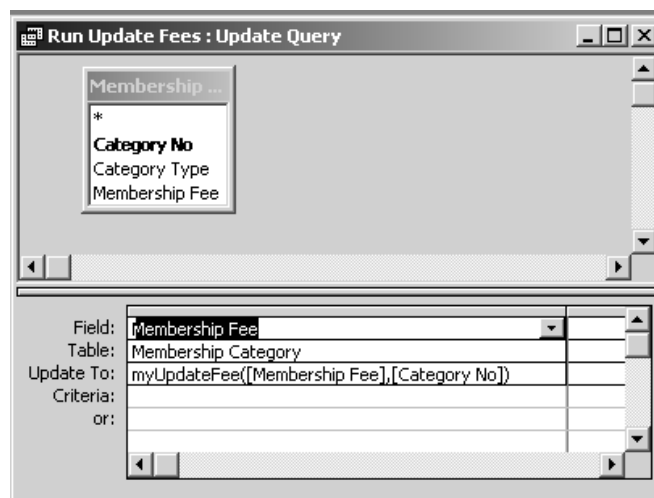


Fig 1.5.4 Run Update Fees update query using myUpdateFee to change the table

You have now used (reused) your myUpdateFee function to calculate column values in both queries. If you have a calculation that needs to be used from more than one place, it is good practice to use a function or procedure so that the details of the calculation are only written in one place. This makes development, testing and maintenance so much easier and reduces errors. After you have tested and

run this Update query, you will need to set the Membership Category table fees back to their original values; see Fig 1.5.3.

1.6 Running a Query from a Command Button

In an application, users are normally limited to the facilities that the developer has provided. They do not normally have access directly to the database, via the Access database window, and they certainly do not normally have access to any code. So, if Chelmer Leisure Centre staff wanted to use the two queries developed in the preceding section, we would need to do something to allow them to do so. You will now add a command button to allow the user to request the Check Update Fees query.

Open the Membership Category form (see Unit 14 of *McBride*) in design mode. Extend the form footer so that there is room for some buttons. Access has a Command Button wizard, so have a look at that now:

- You will need to use the Toolbox, so click the icon (also available via the *Tools* menu) if it is not displayed.
- Select the Control Wizards tool in the Toolbox, then click the Command Button tool. Move to the form footer and click where you want the command button to be put.
- A Command Button Wizard box now pops up, asking you what you want next. Select *Miscellaneous* from the 'Categories' list and note that the 'Actions' list has now changed. This, conveniently, has *Run Query* so choose that, then click Next
- You will now see a list of all the queries in your Chelmer Leisure Database. Choose the *Check Update Fees* query and click on Next.
- Choose Text rather than Picture, as this makes more sense in this context. Change the button text to something more meaningful (such as *Check Fee Changes*) and click on Next.
- Give the button a more meaningful name than the default of *Command99* (e.g. *cmdCheckFees*), and then click on Finish.

If you make an error while creating the button, simply use the cancel or back buttons to correct things. If you have finished but want to delete it, just 'cut' it out of the form. Note that when you delete (cut) a button from the form, any generated code is not deleted. This can be very useful; see section 2.4.2.

Category No	Category Type	Membership Fee
1	Senior	£25.00
2	Senior Club	£30.00
3	Junior	£10.00
4	Junior Club	£15.00
5	Concessionary	£18.00
6	Youth Club	£20.00
*		£0.00

Record: 1 of 6

Fig 1.6.1 New Command Button on Membership Category Form

That's it. If you change to form view you will see your button on the form and if you click on it, you will see the result of the query. See Fig 1.6.1.

VBA code has been used to run the query, but you haven't written any yourself; the code has been generated by the command button wizard. Open the Membership Category form in design view and look at the code module. This should look like Fig 1.6.2.

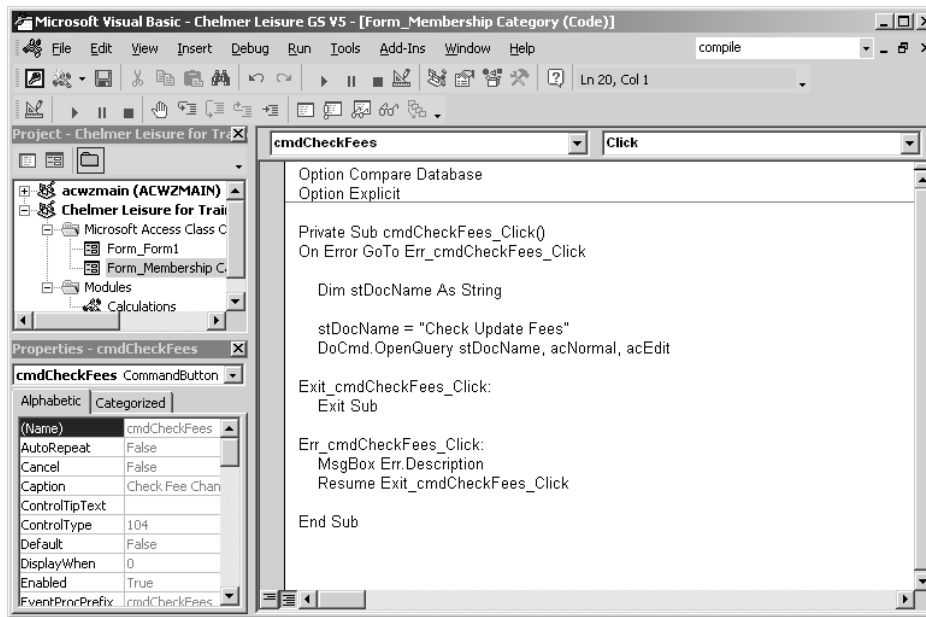


Fig 1.6.2 Check Fee Changes command button click event code

Access has created code automatically for you in the form code module. The code is for a Click event (when the user clicks with the mouse) on the cmdCheckFees object (the command button you have just created). Set a breakpoint in the code then run the query again. When the breakpoint is reached, execution will stop and you will be positioned in the code so that you can step through it, look at data values, etc.

The code in Fig 1.6.2 is explained below. Use the F1 key to find out more.

Private Sub cmdCheckFees_Click()

- *Procedure Header.* Access has assumed that the procedure will not be used from any other code module, so has declared the procedure to be *Private*.
- A procedure name has been generated made up of the command button name and the type of event: *ObjectName_Event*
- The open and close round brackets () at the end of the procedure name mean that this procedure has no parameters.

On Error GoTo Err_cmdCheckFees_Click

- *Standard line generated by all/most wizard code.* Calls the error-handling routine further down in the procedure.
- One situation in which this would be used is if you renamed your query and forgot to change the name of the query in this code.

Dim stDocName As String

- *Definition of a string variable, which is used in the next statement.* It is a local, private variable.
- Use of a variable is not essential, but is good practice as values in variables can be checked in the Debugger.

stDocName = "Check Update Fees"

- *Assignment statement to put the name of the query (as selected in the wizard dialog) into the string variable.*
- *If you change your query name you must also remember to change this!*

DoCmd.OpenQuery stDocName, acNormal, acEdit

- *This statement runs the query.*
- *There are several DoCmd (Do Command) methods in Access; see Appendix J.*

Exit_cmdCheckFees_Click:

- *This is a label not an instruction. Note the colon (:) at the end of the line.*

Exit Sub

- *This will terminate execution of the code by exiting the procedure.*
- *If this line was not here the code would 'drop through' to the code below it in this procedure.*

Err_cmdCheckFees_Click:

- *This is a label not an instruction. Note the colon (:) at the end of the line.*
- *Code here is only executed if a run-time error has occurred.*

MsgBox Err.Description

- *MsgBox is a built-in function to create a message dialog box. You will see how to use this in section 1.7.*
- *The Err object contains the run-time error number and is used to access the appropriate description to be displayed in the message.*

Resume Exit_cmdCheckFees_Click

- *Resumes execution after an error-handling routine is finished, and resumes at the label specified.*

End Sub

- *Procedure footer.*

The query dynaset displayed to the user when the query is run is far from elegant. It's fine for developers when testing, or for users who have a database for their own private purposes, but may not be suitable for a professional-looking application. A better way could be to create a form or report based on the query, and then open that form or report to show the query results. The DoCmd method OpenQuery is used (unsurprisingly) to open a query. The methods OpenForm and OpenReport are also available; see Appendix J. Use the *Help* system to see how these work, and then change the command button code to provide a more elegant display of the information. Or, if you look again at the facilities provided by the Command Button Wizard, you will see that you can choose here to open a form or report, so you could alternatively delete the command button (and its associated code) and create a new one to perform the revised function, creating the form or report first, of course.

It could be useful to have an 'Update Fees' button on the form as well, but this would mean that the user can change the table, and every time the button is clicked, the (new) table values will be updated (again). If this facility is to be allowed, some security measures need to be in place as well. For example, an 'Are you sure?' question in case the button is chosen by mistake. (Even better security would be a method of restricting tasks such as this to certain authorised users only. A common way of doing this is by having a log-in procedure with passwords and authorisation codes for all users). The next section will show how to use questions and messages to provide some simple security for this task. After that, you should be able to do exercise 1.9.3.

1.7 Creating & Using Message & Question Procedures

1.7.1 Sending a Message to the Screen

Open a new Access module, and enter and compile the procedure shown in Fig 1.7.1 below. Save the module as 'Messages and Questions'.

```
Public Sub myDisplayInfoMessage()
' Display an information message to the user
MsgBox "Category Table updated OK", vbInformation
End Sub
```

Fig 1.7.1 Simple Information Message procedure

Use F1 to find out more about the MsgBox function. Note the prompt boxes (and their contents) that pop up to help you when you are typing the code. You will see that there are several types of default buttons and combinations, also several different types of message.

In the code in Fig 1.7.1 MsgBox, although it is a function, is being used as a sub procedure, so no round brackets are required.

Run the procedure in the Debugger by using the Immediate Window as shown in Fig 1.7.2. The message box shown in this figure should then pop up. The information icon (the letter i) is generated by use of the keyword vbInformation.

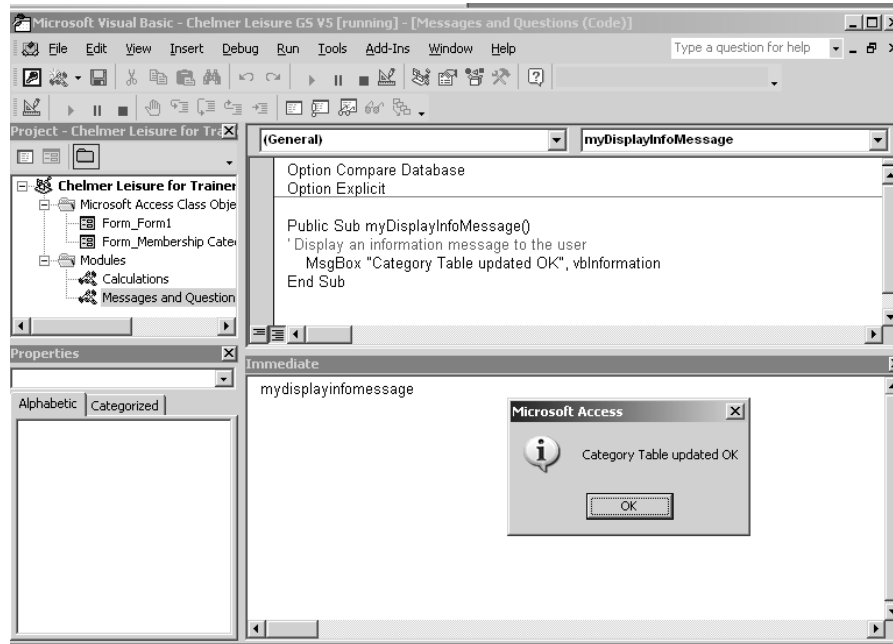


Fig 1.7.2 simple message box test

The message box would look much more professional if the application name was displayed in place of 'Microsoft Access' and would help to distinguish between messages generated by this application and by Access. The procedure would be of much more use if it could be used to display any given message. Amend the code in your 'Messages and Questions' module to look like the code shown in Fig 1.7.3 below.

This code has a Public constant data item for the application name, declared as a global constant by being at the head of the module. This can be referenced by other code as well, so the name will be consistent wherever used (and, if it changes, it only needs to be changed in one place). It could therefore be used for forms and reports as well (see section 2.2.1).

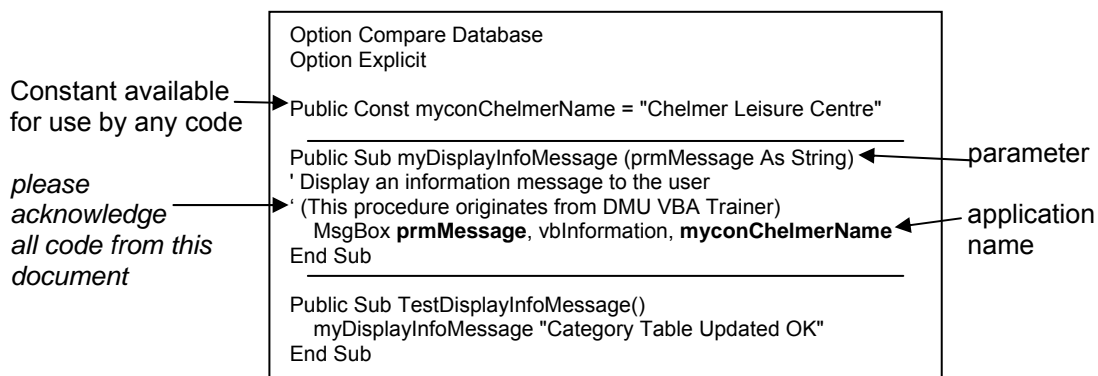


Fig 1.7.3 Better Information Message Procedure

myDisplayInfoMessage now has a parameter prmMessage, with a String datatype, which must be passed to it by the calling code. The parameter is referred to by name within the procedure, but maps to the value passed to it at run time. By default, parameters are passed by reference (keyword ByRef); this means that the procedure is passed the address of the parameter so can reference it directly (and change it if needed). If you want to prevent accidental changes to data passed to a procedure then call it by value

(keyword `ByVal`) in the procedure declaration (`myDisplayInfoMessage (ByVal prmMessage As String)`), but VBA Help states that this method is not so efficient as `ByRef`.

The `MsgBox` line now uses the parameter value in place of the literal value used originally. Note that there is also an extra item on this line, being the constant for the application name.

If you try running `myDisplayInfoMessage` via the Debug Immediate Window, you will need also to specify the message as a parameter. Alternatively, use a test procedure, such as the `TestDisplayInfoMessage` procedure. This procedure simply calls `myDisplayInfoMessage` and passes the required message as a parameter. Simple! Try running the test procedure via the Debug Immediate Window and note that it not only displays the required message but that it also shows the application name in the dialog box header. You now have a useful general-purpose procedure to display messages in this application.

You could extend this procedure to display the module code, form name or process name as well as the application name, by using another parameter as shown in Fig 1.7.4. This little change introduces several useful concepts:

- **Optional parameter.** The user can miss this off the procedure call if wished.
 - o You can specify a default value to be used if wished:
Optional `prmSource As Variant = "default"`
this example simply uses the (rather unimaginative) string 'default' as the default value if the parameter is omitted.
 - o The parameter must be of a Variant datatype. Check help for further information on this datatype. Note that textboxes on forms and reports are automatically Variant datatypes.
- **IsMissing function.** This is one of Access's many built-in functions, and returns a Boolean (True/False) value to indicate whether or not the parameter is missing.
 - o If a default has been set then `IsMissing` returns `False`.
- **If / Else / End If.** An example of how to code using `If`.
- **Concatenation using &.** The `&` (ampersand) character is used to concatenate (join) elements in a string. Here it has been used to join the Chelmer Name and Source strings for the `MsgBox` title and to put a colon (`:`) character between them.

You should now be able to do exercise 1.9.1.

```
Option Compare Database
Option Explicit

Public Const myconChelmerName = "Chelmer Leisure Centre"

Public Sub myDisplayInfoMessage(prmMessage As String, Optional prmSource As Variant)
' Display an information message to the user
' (This procedure originates from DMU VBA Trainer)
  If IsMissing(prmSource) Then
    MsgBox prmMessage, vbInformation, myconChelmerName
  Else
    MsgBox prmMessage, vbInformation, myconChelmerName & " : " & prmSource
  End If
End Sub

Public Sub TestDisplayInfoMessage1()
  myDisplayInfoMessage "Category Table Updated OK", "Update Fees"
End Sub

Public Sub TestDisplayInfoMessage2()
  myDisplayInfoMessage "Category Table Updated OK"
End Sub
```

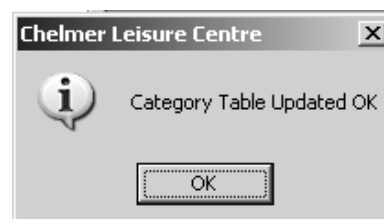


Fig 1.7.4 message boxes with Chelmer Leisure name and optional source of message

1.7.2 Asking a Question

Applications frequently need to ask the user a question and then take different actions depending upon the answer. Open your Messages and Questions code module, and add the code shown in Fig 1.7.5.

The function `myYesNoQuestion` in Fig 1.7.5 asks a given question of the user and presents the user with two buttons by which to reply; the Yes button and the No button. Use VBA Help to check the meanings of the various new keywords in this module that you have not seen before, such as `vbQuestion`, `vbYesNo`. Note that you can use the + operator to specify more than one of these options. (The full list of options can be seen with the Help for the `MsgBox` function). The reply from the user is assigned to the function name, so that this is the value returned by the function.

The test function simply looks to see if the value returned is Yes or No, and uses the existing sub procedure `myDisplayInfoMessage` to display which value was returned. Note that, as the function is now checking the value returned, round brackets must be used around the parameter values. Compare with the code in Figs 1.7.1 and 1.7.3 for sub procedures.

Now test this via the Debugger and check that the function displays the question appropriately and returns the correct reply. See Fig 1.7.6.

If you wished, you could extend the code to add the source of the message, following the code already demonstrated in Fig 1.7.4.

```

'-----
Public Function myYesNoQuestion(prmQuestion As String) As Byte

' function takes a given question, displays it in a question box to the user
' with Yes and No buttons, then returns the yes or no value of the user's reply
' (This procedure originates from DMU VBA Trainer)

    myYesNoQuestion = MsgBox(prmQuestion, vbQuestion + vbYesNo, myconChelmerName)

End Function
'-----
Public Sub TestYesNoQuestion()
    If myYesNoQuestion("are you sure?") = vbYes Then
        myDisplayInfoMessage "yes"
    Else
        myDisplayInfoMessage "no"
    End If
End Sub

```

Fig 1.7.5 Function to ask a simple question



Fig 1.7.6 Running the test for the `myYesNoQuestion` procedure

1.7.3 Getting a value from the screen

As well as the `MsgBox` function used in section 1.7.1, Access has an `InputBox` function to allow you to display a simple dialog box on the screen and ask the user to enter a value.

Add the procedures in Fig 1.7.7 to your Messages and Questions code module and test them using the Debugger to see what happens. The '0' (the last argument in the procedure call) is a default value in case the user decides not to enter one; using a default is optional. Other examples of use of `InputBox` are in section 7.3.2 and Appendix H.5 and H.7.


```

'-----
Public Function myGetValue()
    myGetValue = InputBox("Please enter a number", myconChelmerName, 0)
End Function

'-----
Public Sub TestGetValue()
    myDisplayInfoMessage (myGetValue)
End Sub

```

Fig 1.7.7 simple procedure to demonstrate use of Access InputBox command

1.8 Documenting your Code

As well as documenting your system design, table layouts, test plan and log, form and report layouts, and the like, you should also document your code. There are several methods that you can use to print your code, but none of them will show the useful lines that separate each procedure on the code window, which is a shame. An option is for you to code a comment at the start of each of your procedures, which is simply a dividing line, such as

```
'----- (see fig 1.7.7)
```

Module code can be printed in several ways, three of which are:

- Open the code module and click on the print icon on the main Access toolbar. This method does not provide headers or footers with essential details such as the module name, date printed, page numbers, etc. The method may be adequate for your own development purposes, but not for any final documentation.
- Print via the Access Documenter facilities. This is a reasonable format, with all useful information in the page header plus code line numbers.
- Use Microsoft copy and paste procedures to put into a Word file, and add your own headers and footers. This method is useful if you wanted to incorporate the code in a larger document. But be careful, as some lines can get truncated. And you have to do it all again if you make changes to your code.

Unfortunately, the different font colour for comments (shown in the VB code window) does not transfer to the printer nor to Word files, but you can edit the Word file yourself to show this. Or you could paste lots of code screen prints into a Word file; this will be fiddly, but will show font colours and dividing lines.

So far, you have only created code in three modules, so it should be fairly easy to remember what exists, where it is coded and what it does. But, as your application gets more complex, the number of code modules and procedures will get much larger and more difficult to keep track of. And if that is difficult for you as the developer, just think how much more difficult it will be for future maintenance programmers. Below are two suggestions for documenting code modules and the procedures and functions within them.

1.8.1 Using a standard form

Look at the form shown in Appendix D. This form simply shows a summary of all code in a module. It would be useful if all code for a form or report object was listed together. For further information, a programmer would look at the code and the comments in that code. Appendix D also shows samples of completed forms for the three code modules you have created so far.

1.8.2 Using a database

Create a table for each code module, with columns corresponding to those shown on the form in Appendix D. These tables could be part of the Chelmer Leisure database, or (perhaps better) kept in a separate documentation database. Queries and reports can then be written to list the tables, sorting in object or procedure name order. This method, though requiring a little more work initially, has the advantage that information can be altered and reprinted as changes to the Chelmer Leisure database are made. It also has the advantage that a UNION SELECT query (and possibly a report) can be used

to list all procedures together, sorted by module, which could prove a useful overall reference. SQL for this UNION query for tables for the three code modules (for form 'Membership Category', and the two Access standard modules 'Messages and Questions' and 'Calculations') so far would be:

```
SELECT Object, Type, Event, Name, Public, Description, "Membership Category Form" AS Module
FROM [Membership Category Form]
UNION
SELECT Object, Type, Event, Name, Public, Description, "Calculations" AS Module
FROM Calculations
UNION
SELECT Object, Type, Event, Name, Public, Description, "Messages and Questions" AS Module
FROM [Messages and Questions]
ORDER BY Name;
```

documentation database table names

See Figure 1.8.1 for the result of this query; the tables have been created to show some of the procedures from this part of the Trainer. See also Appendix G.7.

Object	Type	Event	Name	Public	Description	Module
Test fee changes button	Command button	Click	cmdCheckFees_Click		Runs query Check Update Fees to see result of applying the changes	Membership Category Form
Make fees changes button	Command button	Click	cmdUpdateFees_Click		Runs query Run Update Fees to update the fee values in the table.	Membership Category Form
N/A	N/A	N/A	myDisplayInfoMessage	Yes	Displays a given message, showing app name and info icon.	Messages and Questions
N/A	N/A	N/A	myUpdateFee	Yes	Calculates the new fees	Calculations
N/A	N/A	N/A	myYesNoQuestion	Yes	Asks a given question, with reply OK or Cancel. Shows app name and info icon.	Messages and Questions
N/A	N/A	N/A	TestDisplayMessage		Tests myDisplayInfoMessage	Messages and Questions
N/A	N/A	N/A	TestYesNoQuestion		Tests myYesNoQuestion	Messages and Questions

Fig 1.8.1 Result of UNION query to list procedures

1.9 Exercises

1.9.1 myDisplayWarningMessage sub procedure

Use the example set in section 1.7.1 to create a new sub procedure called `myDisplayWarningMessage` to display a warning message. (Hint: `vbExclamation` will display an exclamation mark in the message dialog box; look at VBA Help for `MsgBox`).

1.9.2 Improving the myUpdateFee Function

In Section 1.4.3.2 it was implied that the code for `myUpdateFee` could be improved. You now have the tools to do this.

Suppose a new category (7) was added to the Membership Category table. The sub procedure `myUpdateFee` has assumed that the only categories are 1-6, and `Else` has been coded to cover categories 5 and 6. Your task is now to improve the coding to specify all known codes explicitly and to use `Else` to cover any unexpected codes. A suggestion is that you code the following within `Else`:

```
myDisplayWarningMessage "Unexpected Code - " & prmCategory & " - Fee is unchanged"
```

The `&` sign will concatenate (join) the message text with the value in the `prmCategory` parameter to form the full message. You should have created the procedure `myDisplayWarningMessage` in exercise 1.9.1.

Now, if the Membership Category table contains an unexpected code, a warning message will display to alert the user to a problem. It is always good practice to try to anticipate future changes and to trap exceptions. It is also good practice to use Else in a Case statement as a 'catch-all' and to test explicitly for all other conditions.

Add a new record to the Membership Category table and test your code by running the Check Update Fees query. Don't forget to delete the new record afterwards.

1.9.3 Running an Update Query

You should now have enough expertise to go back to the problem mentioned at the end of Section 1.6 of adding a command button to the Membership Category form to run the Run Update Fees query. You now have all the tools that you need to do this by yourself. Do the following:

- Add a command button to run the query (refer to section 1.6 if you can't remember how to do this). Your form will now have two command buttons. See Fig 1.9.2.
- Edit the code that Access has generated for this as follows (see Fig 1.9.1 for the logic):
 - o ask a suitable 'Are you sure?' question first (see Section 1.7.2)
 - o run the Do Update Fees query, or not, depending upon the user reply
 - o display a message to the user to say whether or not the table was updated
 - o repaint the form to show the new values (see section 1.1.3)
 - o disable the run query button so that the user cannot hit it twice (see section 1.1.3). You will need to move the cursor away from the button first.
 - o Run (and debug if necessary) your code for the button on the form, testing both answers to the 'Are you sure' question.

(Normal text shows Access-generated items, ***bold italics show your own added code***)

IF answer to "Are you sure" question = Yes then

set query name

run the query

display message "Fees Updated OK"

redisplay (Repaint) form to show updated values

move cursor (use SetFocus) away from run update fees update button

grey out (use Enabled property) the run update fees button

Else

display message "Fees Update Cancelled"

End If

Fig 1.9.1 Logic for Run Update Fees command button code

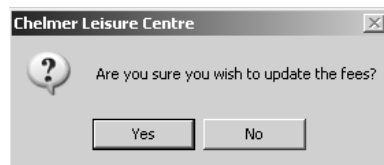


Fig 1.9.2 The form after updating the fees, showing the Update Fees button greyed out (disabled) and the new values on the form

Category No	Category Type	Membership Fee
1	Senior	£26.25
2	Senior Club	£31.50
3	Junior	£10.75
4	Junior Club	£16.50
5	Concessionary	£18.00
6	Youth Club	£20.00
*		£0.00

Buttons: Check Fee Changes, Update Fees

Record: 1 of 6

PART 2

USING EVENT CODE ON FORMS – DATA MAINTENANCE

REVIEW OF PART 2:

In this part of the Trainer you will see...

- ...that a form header can be set up from a constant in an Access module.
- ...that the form `AllowEdits` property can be used to allow the user to change data, or not. It is set to true by default. To alter to View mode, set the `AllowEdits` form property to `False`.
- ...that the `Form_Load` event is used to specify what you want to happen when the form is first loaded.
- ...that the `Form_Current` event is used to specify what you want to happen for each record shown.
- ...that form, report and object default properties set by Access can be amended via the property box and/or via VBA code.
- ...some methods of showing which command button/process the user has chosen.
- ...that VBA has built-in colour constants, or you can use the `RGB` Function to create your own custom colours.
- ...that generated (or your own) code for an object will remain in the code module even though the object has been deleted, and you can use this to link to other objects or as a procedure to be called from elsewhere.
- ...that event code can be used to perform automatic calculations. If the results of the calculations are for information only, they can be displayed in an unbound control.
- ...that the `Form_BeforeUpdate` event is called before data is saved and can be used to trap unsaved changes to data.
- ...that the `Form_AfterUpdate` event is used when data has been updated and can be used for completion processes such as displaying a confirmation message to the user.
- ...that the `On Error` clause within an event can be used to trap specific errors and take appropriate action. For more details see the "Further VBA" Trainer.
- ...that wizard code for deleting records has its own 'Are you sure?' procedure, but this can be disabled via the `Tools` menu.
- ...how to split a message over two lines in a dialog box.
- ...and how to split a code statement over two lines.

See Appendix A for a summary of events that occur for Access objects.

See Appendices B, C and F for some help and advice when coding.

See Appendix E for some suggestions for naming conventions for variables, procedures, etc

2.1 Introduction

In this part of the document, you will see how to create code to customise and improve on the basic data maintenance facilities supplied by MS Access. You will be using the Membership form that you created in Units 14, 15 and 22 of *McBride*. Your basic form will look something like that shown in Fig 2.1.1. There are some differences here to the form shown in Unit 14 of the book:

- The sub heading is now 'Membership Details' as this form will be used here for more than just recording the details of new members.
 - o The default control for `Smoker` is a text box but the user has to type Yes or No (and will see -1 or 0 when the cursor is in the box!). This is not good HCI, so it has been changed to a check box as discussed in *McBride* Unit 22 Task 4.
- The default control for `Sex` is also a textbox, and makes even less sense than for the `Smoker` field. So I have changed it to use an option group (see *McBride* Unit 22 task 3)
 - o Setting -1 [Yes] for Male and 0 [No] for Female when prompted by the wizard.
 - o The group is called 'Sex' and is still of a Yes/No datatype and is bound to the `Sex` field in the Membership table.
- `Membership No` is an `AutoNumber` field, so properties (in the property box) have been set to make it locked (so that the user cannot type into it), and to make it look unlike a data entry field. It has been moved to the top of the form.

This Part of the Trainer places all command buttons directly on the Membership form. Part 4 shows an alternative method using main and sub menus.

Fig 2.1.1 The Membership Details Form

A database normally exists because of a need to process certain information. In order to provide the required processing function, the data has to be entered and then maintained. In order to do this, you will normally need to provide facilities to view, edit, insert and delete data, plus other specific functions needed for the database. The rest of this section will show how VBA can be used to improve on the basic Access facilities, to assist the data entry and maintenance processes. Fig 2.1.2 shows a simple test plan for Part 2, covering the main tests (not an exhaustive plan). Do the tests as you go along, and repeat all tests again at the end after all changes have been made.

Test No	Reason for test	Expected result
1	Default settings when form opens (sections 2.2.1, 2.2.3, 2.4).	Form is in View mode, View button is active, Form header is correct, cannot enter/change data.
	After section 2.6.1.	Form will be in new record mode, with Edit active.
2	Default settings when move to next/previous record (sections 2.2.3, 2.4).	Form is in View mode, View is active, cursor in Title field, cannot enter/change data.
3	Working of Edit and View buttons (sections 2.3, 2.4).	Click on Edit button, Edit button is active, can edit data.
	Also click on Edit button then repeat test 2.	Click on View button, View button is active, cannot edit data.
4	Choose Edit mode, change a field on a record, and click Save button. Choose not to save changes. (section 2.5)	Get own 'Save changes' message. Reply of No does not alter record (plus own conformation message if coded).
5	Choose Edit mode, change a field on a record, and click Save button. Choose to save changes (section 2.5).	Get own 'Save changes' message. Reply of Yes changes record, with confirmation message.
6	Choose View mode and click on Save button (section 2.5).	Get own 'Cannot save unless in Edit mode' message.
7	Redo tests 4 and 5, but move to new record instead of clicking Save button.	As for tests 4 and 5. The unsaved changes are trapped.
8	Add a new record – test both saving and not saving. (section 2.6.1)	New record automatically in Edit mode. Can choose to save or not, and will get appropriate own messages.
9	Delete a record – test both deleting and not deleting (section 2.6.2).	Can choose to delete or not, and will get appropriate own messages.
10	Experiment with various combinations of button clicks, saving or not etc. (all sections)	Active button shows appropriately, get correct own messages and correct actions.

Fig 2.1.2 Simple test plan for Part 2.

2.2 Viewing Data

Open the Membership form in form view. Note that the first record in the table is displayed. (See Section 2.6.1 for how to display a blank record). Change data in a field. Close the form and reopen it. The new data is now in the field, but you were not informed that anything had been changed. It is very easy to change data inadvertently when you are browsing through the records.

2.2.1 Setting a form default

The Form Load event is used for coding all the things that you wish to happen when the form is first loaded. VBA Help: “By running an event procedure when a form’s Load event occurs, you can specify default settings for controls, or display calculated data that depends on the data in the form’s records.” There is also a Form Open event; look at VBA Help for more information.

Change to design view, and look at the properties for the form. Click the Data tab. Note that the AllowEdits property is set to Yes by default. It is a simple matter to set it to No via the property box, but it is more flexible to set it via VBA code, as sometimes you will want it to be Yes and at other times to be No. So, we will start by setting it every time the form is opened.

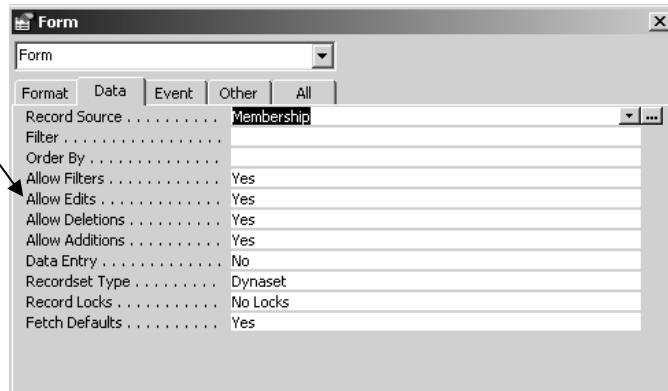


Fig 2.2.1 Form property box, Data tab

Another useful task that could be done in the Form Load event is to set up the main heading on the form. At present, you have probably done this by typing the text directly into a label in form design view, but it is very simple to use the text already in your public constant myconChelmerName (see Fig 1.7.3). Give the name lblHeading to the label and note that the caption property contains the text that you typed in. In the example shown in Fig 2.2.2 the text has been changed to ‘heading’.

To refer to the property you code
 lblHeading.Caption
 (See section 1.1.3)

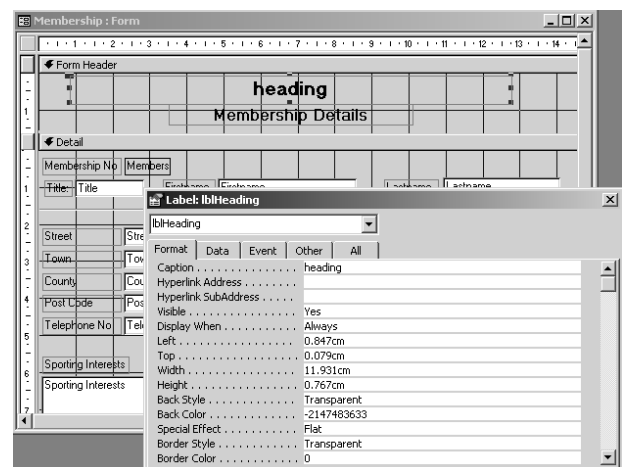


Fig 2.2.2 Label property box showing caption property

Now open the Membership form code module and select Form from the top left-hand box. This will generate the header and footer for the Form_Load event code. (Alternatively, choose the Form_Load event from the form’s property box event tab, click on the build icon (...) and choose Code Builder [see Fig 1.2.4]). Change the code so that it looks like that in Fig 2.2.3.

```
Private Sub Form_Load()

    AllowEdits = False           'set the form to view mode
    lblHeading.Caption = myconChelmerName 'set main heading

End Sub
```

Fig 2.2.3 code for Form Load event

You could also code Me.AllowEdits = False Me is taken to be the current form. See section 1.1.3.

Now open your form in view mode and try to alter some data; anything that you type into a field will simply have no effect.

Look at the heading; this should now contain the text in `myconChelmerName`. You can change the text in the constant and the change will be reflected in the heading and in your message and question procedures that use the constant for the title. This way you can get a consistent title throughout your application and you only need to change it in one place.

You could also put the text in the form caption (the blue line at the top of the form) by coding

```
Caption = myconChelmerName OR me.Caption = myconChelmerName
```

2.2.2 Using a View Command Button

The user will need to inform the database what he or she wishes to do. One way of doing this is by using command buttons. Do the following to create a *View* button:

- Open your form in design mode and extend it to the right hand side or use the footer. (Buttons can be added anywhere on a form).
- The Command Button Wizard does not have a wizard for 'View' so turn the Toolbox Wizards off (click on the toolbox wizard icon) and create a command button on your form. This will have a default name such as `Command27`, so open the property box, click the **All** tab and change the name and the caption to something more meaningful (e.g. `cmdView` and *View*). Note that the name is the object name that you will refer to in your code and the caption is the wording that appears on the button on the form. They do not have to be the same.
- Now you need to create the coding behind the button. Open the code module and select `cmdView` from the left-hand list of objects; Access has assumed that you wish to create code for a `Click` event for `cmdView` and creates the code header and footer for you. Copy and paste the line to set the `AllowEdits` property as shown in Fig 2.2.3.
- Open the form in form view. As before, you cannot enter data. Click on the *View* button, move the cursor to a field and you still cannot change data.
 - o But try commenting-out the line in the `Form_Load` event then trying this procedure out again, and see the effect of the code in the *View* button. This time you will be able to change data even after pressing the button! It is important to note that the `AllowEdits` property applies only to unsaved changes; so if you make a change to the data and then click on the *View* button you can continue to make changes until the record has been saved.

2.2.3 Setting a default for each new record

There is one more situation in which you may wish to reset defaults, this being whenever the user changes to a new record. For this you need to use the `Form_Current` event; VBA Help: *"The Current event occurs when the focus moves to a record, making it the current record, or when the form is refreshed or queried."* The event is also called automatically whenever a form is loaded; so setting the `AllowEdits` property in the `Form_Load` event is not strictly necessary. Use the `Form_Current` event to code anything that you wish to happen when you move to a new record.

Do the following:

- Open the form's property box and click on code builder for the `Form_Current` event. This will create a header and footer for a procedure that will be executed every time the user chooses a new current record.
- Copy and paste the line to set the `AllowEdits` property to this new procedure.
- You may have noticed that when the form is opened, or when you move to a new record, the cursor is positioned in the Membership No field. You could prevent this by changing the field tab order, but you can also do it by coding


```
Title.SetFocus ' position cursor to first field
```

 Do this now in the `Form_Current` event.

2.2.4 Using a common procedure

You have now set defaults to view mode in three situations, and have been guided to use 'copy and paste' to create the code. Duplicating code is bad practice. Further buttons will be added to the form for more maintenance functions, and there may be more defaults to be reset. Do the following:

- Create a new private procedure `mySetToViewMode` in the form code module
- Code (or copy and paste) the line to set to view mode into this new procedure
- Change the `Form_Load`, `cmdView_Click` and `Form_Current` events by deleting the line to set to view mode and replacing it by a call to `mySetToViewMode`. See example in Fig 2.2.4.

```
Private Sub cmdView_Click()
    mySetToViewMode      'activate view mode
End Sub

Private Sub mySetToViewMode()
    AllowEdits = False   'set the form to view mode
End Sub
```

Fig 2.2.4 Use of new procedure to set to view mode

2.3 Editing data

Now add a new command button to your membership form:

- There is no wizard to create an *Edit* button, so create one without the wizard (as for your *View* button).
- Create a Click event for the *Edit* button. You should be able to work out that the code to allow edits would be: `AllowEdits = True` 'allow user to edit data' so enter and save this code.
- Open the form in form view. You cannot enter data as the `Form_Current` event (section 2.2.3) has set the default to view only. Click on the *Edit* button, move the cursor to the desired field and you can now change data.
- Try clicking on the *Edit* button, then moving to a new record. Try editing the new record. You will not be able to, as the `Form_Current` event procedure has set the default back to view only for the new record (and has also positioned the cursor to the `Title` field).
- In a similar manner, you could create buttons to perform frequently required actions such as 'change address' and 'renew membership', moving the cursor to the appropriate field.

2.4 Showing which is the active button

It would be useful to provide the user with some feedback that the click has been actioned and which button is currently active. Unfortunately, there is not an automatic raised/sunken facility with command buttons, so the rest of this section will show you how to indicate which button is active. There are three possible methods shown here, which also demonstrate various VBA code elements and techniques.

2.4.1 By changing text colour on command buttons

Open the property box for the *Edit* button, choose the *format* tab and you will see that there is a `ForeColor` property. This shows the default colour for the text on the button. This property can be changed using VBA code.

There are currently two buttons on the form, *View* and *Edit*. We need to code for the following:

- *View* is the default mode for the form and for each new record
 - o Set initial `ForeColor` on *View* button to black
 - o Set initial `ForeColor` on *Edit* button to white
- *Edit* is invoked only when the user clicks on the *Edit* button.
 - o Do the reverse of the colours above.

First open one of your Access modules and add the following two lines to create constants:

```
Public Const myconButtonActive = vbBlack
Public Const myconButtonInactive = vbWhite
```

These constants are defined here as Public so that they can also be used by other form code.

The *colour constants* vbBlack and vbWhite are built-in to VBA. Look them up in VBA Help; you will see that there is a list of colour constants that you can use. If you want to define a value for a different colour you can use the *RGB Function* (look it up in VBA Help). E.g. to define a colour purple do:

```
Public myPurple As Long      code at the start of an Access code module; global public variable
myPurple = RGB(120, 0, 255) code in Form_Load, possibly for a menu when system starts up
```

Then whenever you want to set something to the colour purple you can use your public variable. Note that you cannot set a constant to the result of a function; you must define a variable and then set the value in the variable.

Next change your code module for the Membership form so that it looks like that in Figure 2.4.1.

```
Option Compare Database
Option Explicit

'-----
Private Sub cmdEdit_Click()
    AllowEdits = True           'allow user to edit data
    myResetButtonsToOff        'clear all button text
    cmdEdit.ForeColor = myconButtonActive 'show edit button as activated
End Sub

'-----
Private Sub cmdView_Click()
    mySetToViewMode           'set the form to view mode
End Sub

'-----
Private Sub Form_Current()
'default settings when user moves to a new record
    mySetToViewMode           'set the form to view mode
    Title.SetFocus           'position cursor to first field
End Sub

'-----
Private Sub Form_Load()
    mySetToViewMode           'set the form to view mode
    lblHeading.Caption = myconChelmerName 'set main heading
End Sub

'-----
Private Sub mySetToViewMode()
    myResetButtonsToOff        'reset button text colours
    AllowEdits = False         'set the form to view mode
    cmdView.ForeColor = myconButtonActive 'show view button as activated
End Sub

'-----
Private Sub myResetButtonsToOff()
' set forecolor to inactive for all buttons
' IMPORTANT - add appropriate line(s) here for each new button added
    cmdEdit.ForeColor = myconButtonInactive
    cmdView.ForeColor = myconButtonInactive
End Sub
```

Fig 2.4.1 Membership form code so far, showing colour change for Edit & View buttons

There are several things to notice about this code:

- There is a new private procedure myResetButtonsToOff. Anticipating that other buttons may be added to the form, this new procedure sets ForeColor text on all (known) buttons to white. A little extra work and planning at the start to choose reusable procedures can save a lot of time later.
 - o This procedure uses the constant myconButtonInactive to set the ForeColor for each button to white.
 - o If you wished to use a different colour you simply change the value that is defined with your constant. This is much better than using the colour constant vbWhite in several places; it makes maintenance much easier, and makes coding more consistent.
- This new procedure is called from the cmdView_Click and cmdEdit_Click procedures, to set the button text ForeColor to the default (white) on all known buttons.

Open your Membership form and see that the *View* button has black text and the *Edit* button has white text. Click on the *Edit* button and watch what happens. Move to a new record and see that the *View* button is now active. See Fig 2.4.2. Put a breakpoint in the *Form_Load* event and use the Debugger to step through code.

The colours black and white have been used here as this document is printed in black and white. You should be able to work out how to use different colours if you wanted.

Fig 2.4.2 the membership form with the first two command buttons showing text colours, view mode

2.4.2 By using labels to simulate raised/sunken buttons

The standard Command Buttons provided by Access have some drawbacks:

- The only back colour allowed is 'grey', as there is no *BackColor* property.
 - The only special effect allowed is 'raised', as there is no *BackStyle* property.
- These defaults cannot be changed (the relevant properties are not available).

You can simulate command buttons by using labels. Fig 2.4.3 shows the Membership form with these 'label-buttons'. (You cannot use pictures on labels, but you can change the *SpecialEffect*, *BackStyle* and *BackColor* properties. You can simulate pictures by placing a label over a picture and putting the label *BackStyle* to *Transparent*, so that the picture shows through).

Fig 2.4.3 Membership Form with labels as buttons, Edit mode

To take advantage of the code generated for command buttons, plus your own code so far, create a copy of your Membership form (the code will be copied with it) and do the following to create your own label-buttons:

step 1. Delete the *View* command button. The code for the Click event will not be deleted.

step 2. Create a label to replace the command button.

- On the *Format* tab, change the Caption to 'View'.
- On the *Other* tab, change the label name to cmdView.
- On the *Event* tab, create code for a click event. As your label name is the same as your original command button name, the event procedure here will link to that originally created for your (now deleted) command button of the same name.
- On the *Format* tab, change
 - Special Effect to Raised
 - Text Align to Center
 The form here has also had Font Weight set to Bold.
- Use *Format*→*Align/Size* to match the *View* label-button alignments with that of your *Edit* button.
- Look at the form in form view – the label looks just like a command button.

step 3. Follow the above steps for the *Edit* button. Look at the form in view mode and click on the 'buttons' and see that the text colour changes as before.

step 4. At the start of the Access code module add the following constant declarations:

```
Public Const myconSunken = 2      (See Help→SpecialEffect)
Public Const myconRaised = 1
```

step 5. Use *Edit*→*Replace* to change:

ForeColor	to	SpecialEffect,
myConButtonActive	to	myconSunken
myconButtonInactive	to	myconRaised

Now open your form in view mode again and see how the 'buttons' work. Note also that you have made this change with minimal changes to the code, partly by making use of constant values. However, for new label buttons you will need to use the wizards to create code for a command button, then delete the button, then create the label button and link it to the wizard code, so this method is not quite as straightforward as a simple use of command buttons.

2.4.3 By using labels to look like hyperlinks

You don't have to use buttons on forms for the user to click on for certain actions. Labels also have Click events so you can use labels instead.

Make another copy of your basic Membership form and do the following:

step 1. Add the following lines to your Access module:

```
Public Const myconLinkActive = vbBlue
Public Const myconLinkInactive = vbBlack
```

step 2. Delete the *View* and *Edit* buttons, but do not delete the code.

step 3. Create labels called lblView and lblEdit with View and Edit as text. Set the FontUnderline property to Yes.

step 4. Use *Edit*→*Replace* in the Membership form code to change myconButton to myconLink. This should change the code to use the new constants set in step 1.

step 5. Change the headers for cmdView_Click and cmdEdit_Click to lblView_Click and lblEdit_Click.

step 6. Using the property box (event tab) for each label, link to the Click event code.

step 7. Change the lines of code that set the ForeColor property to change it for the label not the command button.

- Example, change cmdView.ForeColor = myconLinkActive to lblView.ForeColor = myconLinkActive

Your form should now look like that in Fig 2.4.4.

Just as for the method in section 2.4.2 above, you will need to create wizard code for buttons, delete the buttons, and then link the code to your labels. Alternatively, you can create buttons and put them over the labels (create the label first then the button and move the button to the same position on the form as the label), setting the button Transparent property to Yes. See Fig 2.4.5.

Membership with labels looking like hyperlinks

Chelmer Leisure and Recreation Centre
Membership Details

Membership No:

Title: Firstname: Lastname:

Street: Occupation:

Town: Date of Birth:

County: Category No:

Post Code: Telephone No:

Sporting Interests:

Smoker: Sex: Male Female

Date of Joining: Date of Renewal:

Record: of 20

Fig 2.4.4 Membership form using labels to look like hyperlinks.

2.4.4 Some useful command button properties (these also apply to labels, textboxes etc)

The property box tabs use 'Yes' and 'No' for the settings. In VBA code, you must use True and False, respectively.

Property	Tab	Setting	Result	Example of use
Enabled	Data	Yes	Button click is actioned	Normal button use
		No	Button is 'greyed out' and click is not actioned. Cannot give the focus to the button in VBA code	To show the button on the form, but prevent the user from using it (perhaps until a data value has been validated). See section 3.3.3.1.
Visible	Format	Yes	Button is shown on form	User can see the button
		No	Button is not shown on form and click on area where it should be is not actioned. Cannot give the focus to the button in VBA code	To hide the button to prevent user from both seeing & using it. See section 4.3.
Transparent	Format	Yes	Button is not shown on form but click on area where it should be is actioned if Enabled = Yes	Buttons on a map or diagram – user clicks on features of map or diagram, but user does not see the actual button. See section 3.2.2.1.
		No	Button is shown on form	User can see the button
Control Tip text	Other	Enter own text	Text shows up when cursor is positioned over the object.	Help tips. See section 4.2.6. Useful also when combined with Transparent property on a picture or diagram.

Fig 2.4.5 Some useful command button properties

The remaining sections in this document use the command button version of the Membership form from 2.4.1. If you prefer to use one of the other versions, simply adapt the code as appropriate.

2.5 Saving records

So far, when you have edited data, the changes are saved automatically. It would be far better to give the user control over when, or even whether, to save data.

2.5.1 Using a Save Command Button

Add a new command button (cmdSave) with text (Save) in white. You can use the wizard to do this and the wizard will create code to save the record under a click event for the button. Look at the code.

Add a line to your myResetButtonsToOff procedure to include the new cmdSave button.

Try the button out. If you have not changed to Edit mode you will get an Access error message as shown in Fig 2.5.1, caused by the fact that you can't save when in View mode (but this fact is not very obvious from the message).



Fig 2.5.1 Message when trying to Save when in View mode

Amend the wizard code for the Save button so that it looks like the code in Fig 2.5.2. The code in bold will be your added code.

```
Private Sub cmdSave_Click()
On Error GoTo Err_cmdSave_Click

If AllowEdits Then           'if AllowEdits is True

    myResetButtonsToOff       ' clear all button text
    cmdSave.ForeColor = myconButtonActive ' show save button as activated

    DoCmd.DoMenuItem acFormBar, acRecordsMenu, acSaveRecord, , acMenuVer70
    'form BeforeUpdate event will be called from here automatically
    'if save is OK, then the form AfterUpdate event is called

Else
    myDisplayWarningMessage "Cannot save unless in Edit mode"

End If
mySetToViewMode           'set back to view after save – here for now – see section 2.5.3

Exit_cmdSave_Click:
Exit Sub

Err_cmdSave_Click:
MsgBox Err.Description
Resume Exit_cmdSave_Click

End Sub
```

Fig 2.5.2 Amended code for Save button

Note the following:

- This code checks the AllowEdits property to give a more meaningful custom message to the user if View mode is set on. This is probably better than just doing nothing as it gives feedback to the user.
- The code uses the myResetButtonsToOff procedure (reuse of code) which should now refer to the Save button as well.
- Comments after the DoCmd statement to save the record refer to the BeforeUpdate and AfterUpdate events for the form; these are explained below.
- The code ends by setting back to view mode (calling the private procedure within the module).

Your form will now look something like that shown in Fig 2.5.3 below. Clicking on the Save button will save the record. How to display the 'saved OK' message is discussed in section 2.5.3.

Fig 2.5.3 Membership form using the Save button from Edit mode

2.5.2 Using the Form_BeforeUpdate event

Now try editing a record and then moving to a new record or closing the form without clicking on the Save button. Your changes have been saved automatically, but you have not been informed that this has happened. The same thing happens if you close the form after changing a record. You need to be able to trap the fact that some data has been changed and alert the user. The event to do this is the Form_BeforeUpdate event. VBA Help *“The BeforeUpdate event occurs before changed data in a control or record is updated”*. (Look at the VBA Help system for full details).

```
Private Sub Form_BeforeUpdate(Cancel As Integer)
'catch MSAccess check when a record is changed
'this procedure is invoked if a change has been made to a record and the
' user is attempting to move to a new record or close a form
'it is also invoked by save_click, addnewrecord_click, etc.
' which gives a useful chance for the user to cancel if the button was
' pressed in error
If myYesNoQuestion("Save Changes?") = vbNo Then
    Undo           'undo changes
    Cancel = True  'cancel save
Else
    ' Access will automatically save the record
End If
End Sub
```

The Cancel parameter is for you to tell Access whether or not you want to save the record. It is initially set to False (by Access) which means that the save is to go ahead. You can set it to True if you want to cancel the save.

Fig 2.5.4 code for Form_BeforeUpdate event

Create an event procedure for the Form_BeforeUpdate event, as shown in Fig 2.5.4. This piece of code simply asks the user whether or not he/she wishes to save changes. If the user replies 'No', then the changes to the form are undone and the save is cancelled (see information above about the event parameter Cancel. See also section 3.3.). Otherwise the changes are saved.

However, if the user replies 'No', the error procedure within the cmdSave_Click event may be invoked. Access will display a message informing you that the DoMenuItem (i.e. the save) has been cancelled. To disable this message, go to the cmdSave_Click procedure and change the error coding to read as follows:

```
Err_cmdSave_Click: ← note : (colon) at end of line indicating that this is a label, not code
If Err = 2501 Then
    'ignore DoMenuItem cancelled message caused by cancelling changes in BeforeUpdate event
Else
    MsgBox Err.Description
End If
Resume Exit_cmdSave_Click
```

(On Error is discussed in more detail in the “Further VBA” Trainer. You can put a breakpoint on the line and see the value in the Err object and watch what happens).

You could also display a message to the user confirming that the save has been cancelled (you should know how to do this by now, using your myDisplayInfoMessage procedure).

Now try adding a new record but leaving at least one mandatory field (Required = Yes in table definition) empty (simply tab over it). If you click on the *Save* button, or try to move to another record, Access will prompt you to enter a value in the field, and then you can try to save again. However, if you do this again but click on the *Close* button, even if you say 'Yes' to saving changes, the form is then closed, there is no error message and the record is not saved. It may therefore be best to disable the *Close* button while *Edit* or *New* modes apply and enable it only after each *Save* or *Cancel*. I can't find any way of trapping this condition in code; please let me know if you can. See Exercise 2.7.4.

2.5.3 Using the Form_AfterUpdate Event

It would also be good HCI to confirm to the user that the record has been saved successfully. The best place to do this is in the *Form_AfterUpdate* event. VBA Help: "*The AfterUpdate event occurs after changed data in a control or record is updated.*" (Look at the VBA *Help* system for full details). Add an *AfterUpdate* event for the form with code shown in Fig 2.5.5. Note that the call to *mySetToViewMode* is now here, as this could be a more appropriate place to put it, as we know that the record has been saved OK.

```
Private Sub Form_AfterUpdate()
    'This is executed when a record is updated in the underlying table/query
    myDisplayInfoMessage ("Data for " & [Firstname] & " " & [Lastname] & " saved OK")
    mySetToViewMode 'moved here from cmdSave – see Fig 2.5.2.
End Sub
```

Fig 2.5.5 code for *Form_AfterUpdate* event

Now test your changes by doing the following:

- Open your Membership form. It will be in the default view mode. Change a record, click on the *Save* button. Say NO to changes and check result.
- Change the record again, click on the *Save* button and this time say YES. Check result.
- Change record and move to another record without saving. See what happens. Test YES and NO replies.
- Change record and close form. Test both YES and NO replies.
 - o When you reply NO you will get a 'You can't change this record at this time' message from Access. If you have your own wizard *Close* button on the form and disable the little X in the top right-hand corner, the form will close without giving this message.
- Change record, save it (reply YES), move to a new record.
- Do not change record, move to a new one.

You should now have seen how the *Form_BeforeUpdate* and *Form_AfterUpdate* events work, and that they are both called automatically when a record is saved. See section 1.1.4 about the order of events.

The method shown above cancelled the changes automatically when the user chose not to save. This could be a nuisance if the user hit 'No' by mistake, or if he/she wishes to make more changes before saving. You might prefer to have a *Cancel* button on the form and, when the user chooses 'No' to tell him/her to click on this button to cancel the save. Note that until the save has been cancelled, the user will not be able to do anything else. A *Cancel* button can be created using the command button wizard.

2.6 Adding and Deleting records

As well as the options of *View*, *Edit* and *Save*, two more general maintenance functions are *Add* and *Delete*. In the items below, customise the code (in your *myResetButtonsToOff* procedure and the *cmdNew_Click* and *cmdDelete_Click* events) so that the text colours on the buttons change appropriately when the button is clicked. Note also that edit mode must be set on for a record to be added or deleted.

2.6.1 Add a New Record

It is possible to add records by clicking on the 'new record' button on the Access toolbar on the bottom of the form, but, to be consistent and for better control of operations, it is best to add a command button of your own, to add to the existing *View*, *Edit* and *Save* buttons.

Add a *New* command button (use wizard for Add New Record),

Look at the code generated for this event. The following line presents a blank form to the user for a new record, but the record navigation buttons can still be used to access previous records:

```
DoCmd.GoToRecord , , acNewRec
```

Add the four lines shown in Fig 2.6.1 to the code for the *New* button, after the DoCmd.GoToRecord statement. Click on the *New* button and enter and save a new record. Note that the Form_BeforeUpdate procedure is called automatically to provide the user with a chance to save or cancel, and the Form_AfterUpdate procedure is also called if a record is saved (see Section 2.5), so no new procedures or messages are needed here.

```
cmdNew.ForeColor = myconButtonActive    'show New button as activated
cmdView.ForeColor = myconButtonInactive  'show View as deactivated (was set by Form_Current event)
AllowEdits = True                       'as user will be entering new data
Title.SetFocus                           ' position cursor to first field
```

Fig 2.6.1 code to add to New button event code

If you wanted the form to load with a blank record as (rather than the first record) then you could simply copy the DoCmd.GoToRecord statement into the Form_Load event. However, setting the AllowEdits property to False has no effect with a new record, so showing the form as apparently in *View* mode is rather misleading in this application. The only way round this that I can see is to set a 'flag' (a variable that the code uses to tell it what to do in different situations – this is a common programming technique) to tell the code when the form has just loaded. Look at the code in Fig 2.6.2.

- The lines in **bold font** are the changes to existing code.
- Dim bFormLoad As Boolean declares a global private variable called bFormLoad of type Boolean. *This is the flag.*
- The *Form_Load event* calls the event code for the *New* button in order to set the button activations. Calling event code in this way is perfectly allowable and can be a very useful programming technique. After doing this it sets the bFormLoad flag to true so that the next automatic call to the Form_Current event will know that the form has just loaded.
- The *Form_Current event* is called automatically by Access from the Form_Load event, so this checks the flag to see whether or not it has been called from that event, or from something else (such as a move to a next or previous record). If it has been called from the Form_Load event, then the flag bFormLoad is set to False (as it has now served its purpose) otherwise the form is set to *View* mode.

```
Dim bFormLoad As Boolean 'flag set to True when form is loaded, for new rec at start

'-----
Private Sub Form_Current()
'default settings when user moves to a new record and at Form_Load
If bFormLoad Then      'is call from Form_Load event?
  bFormLoad = False    'Yes - set flag to false - have new record with Edit set to True
Else
  mySetToViewMode      'No - set the form to view mode
End If
  Title.SetFocus       ' position cursor to first field
End Sub

'-----
Private Sub Form_Load()

cmdNew_Click         'open form in new record mode (blank record)
bFormLoad = True    'to tell Form_Current event that call is from Form_Load event

  lblHeading.Caption = myconChelmerName 'set main heading
End Sub
```

Fig 2.6.2 code to load form in New Record mode

2.6.2 Delete an Existing Record

Create a *Delete* command button using the wizard. When you test this you may notice that Access has its own 'are you sure?' procedure. To make this more elegant, do:

- Ask your own 'are you sure?' question.
- Disable the Access message by using *Tools* → *Options* → *Edit/Find* and clearing the tick for *Record Changes*.
- You will also need to add code to *myResetButtonsToOff* as before.
- Suggested changes to code for the *Delete* button are shown in bold in Fig 2.6.3. There are two new features demonstrated in the code for the message.
 - `vbCrLf` is a built-in constant for a new line. Look at the message in Fig 2.6.4 and note that the text is now spit over two lines. (CrLf stands for 'carriage-return, line-feed')
 - The `if` statement is rather long so is split over several lines. The end of one line terminates in a space and then an underline character, and the start of the next line begins with `&` and a space character.

```

cmdDelete.ForeColor = myconButtonActive      'show Delete button as activated
cmdView.ForeColor = myconButtonInactive      'show View as deactivated (was set by Form_Current event)
AllowEdits = True

If myYesNoQuestion("Are you sure you want to delete this record? " _
  & vbCrLf & [Firstname] & " " & [Lastname]) _
  = vbYes Then
  DoCmd.DoMenuItem acFormBar, acEditMenu, 8, , acMenuVer70 'wizard code
  DoCmd.DoMenuItem acFormBar, acEditMenu, 6, , acMenuVer70 'wizard code
myDisplayInfoMessage "Record deleted OK"
Else
myDisplayInfoMessage "Deletion cancelled"
End If

```

Fig 2.6.3 Changes to Delete button code to ask an 'Are you sure?' message

Note that this procedure will physically delete the record, and any cascaded deletions, from the database. This may not always be what is wanted, as the record details could not then be used in any historical data analysis. In some applications it may be more appropriate to flag the record as deleted or 'not current', possibly by setting a status field in the record to a certain value. Queries that list current records would then have to ensure that these records are excluded from any list. Alternatively, you could move the record details to separate archive table(s) possibly by an Append query or embedded SQL.

Fig 2.6.4 Screen and message for Delete

2.7 Exercises

2.7.1 Buttons to Renew Membership and Change Address

As suggested at the end of section 2.3, add buttons to allow the user to renew membership (perhaps setting a default renewal date of the system date) and change address.

The buttons will be non-wizard buttons, and the cursor should be positioned automatically in the Date of Renewal or Street field as appropriate.

You will need to do coding to ensure the correct button colours and to set the `AllowEdits` property to the correct value.

2.7.2 Use Cancel Button on Membership Form

As suggested at the end of section 2.5, instead of automatically cancelling (undoing) the changes when the user chooses not to save, provide a cancel (undo changes) button that the user can use instead.

You will need to set the `Cancel` property to `Yes` in the button property box (check VBA Help to see why).

Try making the button invisible or greyed-out initially (in the property box in form design view), make it visible by code if the user replies 'No' to the `Save` message, and set it back to invisible by code (move the cursor to another button or field first) after it has been used.

This is a good exercise to make you think about the order of events.

2.7.3 Stock Levels Form

Create a form for the Stock Levels table, and add some add data maintenance functions for that as demonstrated in this section for the Membership form.

This form will be used again in the next section.

2.7.4 Enabling/Disabling Close button

As discussed at the end of section 2.5.2, enable the `Close` button only when there is no unsaved data on the form.

A suggested method is:

- `Form_Dirty` event: disable the button.
 - This event is activated whenever there is a change to any of the data on the form.
- `Form_BeforeUpdate` event: if the user says No to changes, then enable the button.
 - The user has decided to abandon some changes, and may want to close the form.
- `Form_AfterUpdate` event: enable the button.
 - The record has been saved, so the form can be closed.

PART 3 – USING EVENT CODE ON FORMS – MISCELLANEOUS FEATURES

REVIEW OF PART 3:

In this part of the Trainer you will see...

- ...that a BeforeUpdate event can be used to perform field or form validations before a value or record is updated.
- ...that an AfterUpdate event is used for any calculations or actions that you want to do with the new value once data has been updated.
- ...how to code for field and form validations.
- ...use of the Forms Collection to reference objects on an open form.
- ...how to write a function to calculate a person's age from their date of birth.
- ...how to use various built-in functions, including some to help with validations and some for data conversions.
- ...how to use Domain Aggregate Functions to find and count records in a table or query, and to use this to show total records on a form.
- ...that command button code can be called just like any other procedure code. This can be useful for coding automatic saves, for example.
- ...that field GotFocus and LostFocus events are used for actions required when the cursor leaves or enters a field.
- ...how to filter records on a form and count up the records filtered.
- ...that a filter condition, and a criteria condition for a Domain Aggregate Function, both need to be coded just like the WHERE clause for an SQL statement.
- ...some uses of list and combo boxes, and how to change the contents at run-time.

See Appendix H for details about Access built-in Functions.

See Appendix I for details about the Forms Collection.

See Appendix J for some useful DoCmd methods.

3.1 Introduction

The basic data maintenance functions as described in Part 2 only go so far. Part 3 covers some further features such as doing calculations, highlighting fields on a form, searching for records, validations, filtering records, counting records, etc. These features not only improve on the basic data maintenance functions but they make interrogation of the data easier (after all, the whole point of putting data into a database is to do something with it!).

3.2 Automatic Calculations

3.2.1 Stock Form

3.2.2.1 Value of Each Stock Item

It is easy to show some calculations on forms by using the standard Access facilities. For example, using a Stock form (based on the Stock Level table, possibly from Exercise 2.7.3) you can show the total stock value for each item on the form.

Item Code	Item	Stock	Re-order level	Unit Price
1	Sports towel	30	25	£6.99
2	T-Shirt (small)	72	20	£10.50
3	T-Shirt (medium)	17	20	£12.00
4	T-Shirt (large)	42	20	£14.00
5	Skipping rope	13	10	£1.99
6	Knee support	7	5	£3.75
7	Ankle support	3	5	£3.50
8	"Buns of Steel" video	66	30	£12.00
0		0	0	£0.00

Fig 3.2.1 Stock Level table with column for Unit Price

First add a column for Unit Price to your Stock Level table, as in Fig 3.2.1. Then add the Unit Price field to your Stock form.

Add an unbound textbox to the Stock form. This will not be bound to the table. Set the following values in the property box (see Fig 3.2.2):

- A suitable name for the field.
 - The name here is txtValue, where txt identifies the field as a textbox (see Appendix E).
- The formula =[Stock]*[Unit Price] in the ControlSource property.
 - The value in txtValue will change automatically if the quantity or the price is changed.
- Format = Currency.
- The textbox is not a data entry field so show this by:
 - Locked = Yes
 - BackStyle = Transparent
 - SpecialEffect = Flat

An alternative (and possibly simpler and more flexible method) would be to create a query based on the Stock table with a calculated column for the Total Value, and then base the Stock form on this query.



Fig 3.2.2 Properties for the Stock Value textbox

Whichever method you use, your form could now look like that shown in Fig 3.2.3 (apart from the re-order level highlighting and the *Receive Stock* button; these are demonstrated later in this section).

Fig 3.2.3 Stock form showing value in stock for the item shown.

A third method of doing this would be to code `txtValue = [Stock]*[Unit Price]` in the `Form_Current` event. But if you do this you will have to remember to update it yourself (by VBA code) whenever the `[Stock]` or `[Unit Price]` values are changed.

You should note that if you have a value already set in the `ControlSource` property you cannot override this in code. You will get the error message shown in Fig 3.2.4 if you attempt to do this.

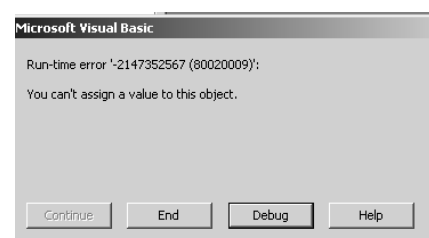


Fig 3.2.4 Error Message when ControlSource property is already set.

3.2.2.2 Highlighting low stock

The form shows the re-order level. It is likely that a regular re-order report will be produced to list all items that need re-ordering, but it could also be useful to highlight these items on the form. A simple method could be to show the re-order level field with bold font and a red background. Add the procedure shown in Fig 3.2.5 to your Stock form code module, and add a call to the module in the Form_Current event. Now, when you open your form and move from record to record, the low stock items are highlighted and the rest are shown normally.

Note that you need to code for both situations in order to reset values. If you did not code the Else clause (and actions) then the Bold/Red highlighting would apply to all records after the first low stock item has been shown, whether they are low stock or not.

```
Private Sub myCheckReorderLevel()
'highlight low stock items

    If [Stock] < [re-order level] Then
        [re-order level].FontBold = True
        [re-order level].BackColor = vbRed
    Else
        [re-order level].FontBold = False
        [re-order level].BackColor = vbWhite
    End If
End Sub
```

Fig 3.2.5 Procedure to highlight low stock

A new feature for Access 2000/2002 is that of Conditional Formatting which works in a very similar manner to that for Excel. You would be able to do the highlighting of the Re-order Level field using Conditional Formatting instead of by code, but code can be more flexible.

If you wished, you could also have information on the form giving further details about the stock situation, such as when the stock was ordered, how much was ordered and when the delivery is expected. This would require further field(s) on the Stock Level table, which you would put on the form and set visible/invisible as appropriate, in the new procedure myCheckReorderLevel. You would not be able to do this using conditional formatting.

3.2.2.3 Adding Stock

When stock is added (stock that has been ordered has now arrived, for example) you need a procedure to do the adding. It is possible to use the form as it is and let the user amend the number in stock, but they may count up incorrectly and get it wrong. It may also be required to record the date when stock was received, but this would need an extra field in the Stock Level table.

You could do this by adding a 'Receive Stock' button to the form, with the code shown in Fig 3.2.6.

```
'-----
Private Sub cmdReceiveStock_Click()
'record new stock

    myResetButtonsToOff
    cmdReceiveStock.ForeColor = myconButtonActive
    AllowEdits = True
    txtReceived.SetFocus 'move focus to field for stock amount

End Sub
'-----
Private Sub txtReceived_AfterUpdate()
'update stock level with value received

[Stock] = [Stock] + txtReceived 'update number in stock
myCheckReorderLevel           'check highlighting etc
cmdSave_Click                  'save automatically, if wished (saves user having to hit the button)

End Sub
```

Only if you have coded button colour procedures

Fig 3.2.6 Code for recording stock received

Points to note:

- The field AfterUpdate event is the normal place for putting actions that you wish to take place once the data has been entered (compare this with the Form_AfterUpdate event).
- txtReceived is an unbound textbox on the form, but it could also be a field from the table if it is required to record the number of items received.
 - o As it is a textbox it will need to be cleared for each record so code txtReceived = Null in the Form_Current event. Textboxes have Variant datatypes so can be set to Null.
 - o Validations are discussed in Section 3.3. See also Exercise 3.7.1.
- Here the event is used to:
 - o Update the number of stock for the item. Note how to code the assignment statement.
 - o Check whether or not the item is still low on stock.
 - o If any information needed to be added to any fields in the table recording date of receipt etc then this could also be done here.
 - o Call the Save button event code. This saves the user the necessity of doing so and could speed up the entering of details of new stock.
 - o If the ControlSource of the textbox txtValue has been set to the formula to calculate the total value, then this value will be updated once all the code in the AfterUpdate event has been completed. If you have coded this formula in the Form_Current event, then you will need to update the value there.

3.2.3 Showing the Member's Age on the Membership Form

3.2.3.1 A myCalculateAge Function

It might be useful to show the member's age on the form. This would act as a check that the correct membership category has been chosen, and may also help with visual identification of members. This formula could be coded directly into a suitable form event, but it is always good practice to code separate tasks in separate procedures, so that the tasks can be reused. Create a new function in your code module Calculations, as shown in Figure 3.2.7.

```
Public Function myCalculateAge(prmBirthDate As Date) As Integer
'given a date of birth, the function calculates an integer age, using today's date
'(This procedure originates from DMU VBA Trainer)
Dim dtDate As Date           'for system date
Dim intAge As Integer        'variable for result of calculation
dtDate = Date                'system date
dtDate = #11 Aug 2004#      'testing only - remove after testing
intAge = Year(dtDate) - Year(prmBirthDate) 'subtract years
If Month(dtDate) < Month(prmBirthDate) Then 'if not reached month yet
    intAge = intAge - 1 'subtract 1
Elseif Month(dtDate) = Month(prmBirthDate) Then 'or if same month
    If Day(dtDate) < Day(prmBirthDate) Then 'but not reached day
        intAge = intAge - 1 'subtract 1
    End If
End If

myCalculateAge = intAge 'set result as value to return
End Function
```

Note this useful method of testing with different 'system' dates. Rather than coding Date() directly in the calculations, put the system date in a variable first. When testing, you can then set the variable to any other date that you wish.

Test No	Birth Date	Reason for test	System date	Expected result
1	1 Mar 1970	Date with month before current month	11 th Aug 2004.	34
2	1 Nov 1970	Date after current month, same year as test 1.	As above	33
3	10 Aug 1970	Boundary test – birthday was yesterday	As above	34
4	11 Aug 1970	Boundary test – birthday is today	As above	34
5	12 Aug 1970	Boundary test – birthday is tomorrow	As above	33
6	31 Dec 1969	Last day of year	As above	34
7	1 Jan 1970	First day of year, 1 day after test 6. Age same as test 6	As above	34
8	2 Feb 1994 1 Mar 2004	Boundary tests with leap year system date.	29 th Feb 2004	10 9
9	29 Feb 1992	Boundary test with leap year birthday.	28 Feb 2003 1 Mar 2003	10 11

Figure 3.2.7 Code and test plan for function myCalculateAge.

Testing Notes...

- ...When you have conditional expressions using '>' or ranges, you must check values 'at the boundaries'. In the example above, the boundaries are when a member's date of birth is one day either side of, and equal to, the system date (Don't forget to check to check for equality; many students check '>' and '<' and forget to check '='!). Boundaries are points at which code can often fail to work as expected.
- ...When testing with dates it is also useful to check the last and first days of a year, and leap years. These are also points at which code can fail to work as expected.
- ..The plan and expected results show the actual data to be used for testing and the actual result expected. Work this out in advance of testing!

Use the Debug Immediate Window (section 1.4.3.2) to test this out: ?myCalculateAge (#1 Mar 1970#)
Note how to enter dates when debugging and coding. Access is American software so there can be confusion between USA and UK formats. It is safest to type as #dd mmm yyyy# format. In code, Access will change the format, but the date value will be as you want it to be.

Compare this new function with the DateDiff function (use the VBA Help system). The statement
DateDiff("yyyy", #1 Nov 70#, date())
will merely subtract the years of the given dates, and in this case will give 34 as the result (assuming today's date is in August 2004).

This new Public function can now be used anywhere that you want to show the member's age.

The function can also be used in a query (just as you used your myUpdateFees function in section 1.5); see Fig 3.2.8. This will give a more accurate value than that used in McBride Unit 13 Task 1.

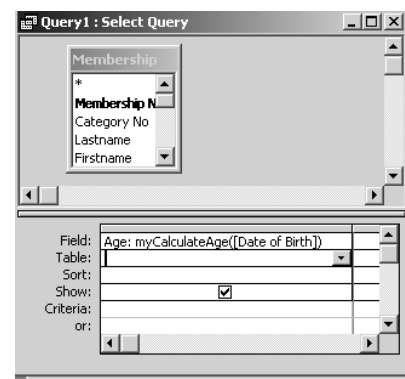


Fig 3.2.8 Using the myCalculateAge function in a query

3.2.3.2 Showing the Age on the Form

Create a textbox on the Membership form by the Date of Birth, and give it the name txtAge with suitable text in the label and suitable formatting (Locked = Yes, BackStyle = Transparent, etc)

Two situations when it would be useful to show the member's age are:

- When a date of birth is changed:
 - o Create code for the AfterUpdate event for the date of birth field and enter the code
txtAge = myCalculateAge([date of birth]) ' show member age
- When each new record is shown:
 - o Open the code for the Form_Current event and enter the line of code shown above. When you open the form with a blank record or move to a new record you will get the error "Run time error '94': Invalid use of Null". The failure occurs because the Date of Birth field for the new record is null.
 - One way round this is to alter myCalculateAge to return a Null, or perhaps an error code, if the date of birth is Null.
 - Another is to trap the error yourself by code. See Part 2 of the 'Further VBA Trainer.
 - A third way is to code the following in the Form_Current event:

```
txtAge = Null ' clear any previous age
If Not IsNull( [date of birth]) Then ' check field is not null first – uses built-in IsNull function (App H.5)
    txtAge = myCalculateAge([date of birth]) ' show member age
End If
```

Your form should now look like that shown in Figure 3.2.9, with the age showing in the age field. (The Category Type is from section 3.3).

Important: look at VBA Help for the IsNull function.

Note that coding `if [date of birth] = Null` OR `if [date of birth] <> Null` will both give the wrong result. You must use the IsNull function to check for Null in a textbox. The textbox must not have a format or input mask that restricts data entry if you want to check what the user has entered.

The screenshot shows a 'Membership' form titled 'Chelmer Leisure and Recreation Centre Membership Details'. The form contains several text boxes for data entry: Membership No (1), Title (Mr), Firstname (Andrew J), Lastname (Walker), Street (16 Dovecot Close), Town (Chelmer), County (Cheshire), Post Code (CH2 6TR), Telephone No (01777 569236), Occupation (Builder), Date of Birth (12/03/1952), Age (52), Category No (2 Senior Club), Sex (Male), Smoker (checked), Date of Joining (03/02/1992), and Date of Renewal (03/02/1997). There are also buttons for View, New, Edit, Delete, Save, Renew Membership, and Change Address. A status bar at the bottom indicates 'Record: 1 of 20'.

Test No	Reason for test	Date of birth	Expected result
1	Change a member's date of birth and check that the Age field changes and is correct. (This tests correct working of field AfterUpdate event).	Member 1, Andrew Walker, DOB = 12/03/1952, Change DOB to 12/03/1953.	Age = 52 initially. Age = 51 after change made.
2	Move from record to record. See the Age for each member. Check that it is correct.	Member 1 DOB = 12/03/1952 Member 2 DOB = 29/11/1960 Etc.	Age = 52 Age = 43
3	Click on the <i>New</i> command button, the Age field should be blank initially, and will show the member's age after the date of birth is entered.	Add new record with DOB of 15/7/1944.	Age is blank initially for the new record. Age = 60 after DOB has been entered.
4	If you have set the form to open with a blank (new) record when it loads then the code should also show the Age field as blank.	N/A	Age is blank initially when form opened in new record mode.

Figure 3.2.9 Membership Form with Age field (run date = 1st August 2004)
And possible test plan

The field AfterUpdate event is the normal place for performing calculations on field data, as the data has now been accepted into the field (see also section 3.3.1). The results of the calculations can be put into other fields as well as into unbound controls. You could, for example, use the age to suggest a suitable membership category for new records and when the date of birth is changed.

If you have coded the *Cancel* procedure for exercise 2.7.2 you may also need to code the calculation for the age in there as well.

An alternative method to having a calculated field on the form could be to base the form on a query which has a calculated column for the age, but:

- either... the myCalculateAge function would then need to cater for the situation where the Date of Birth field is blank (as for a new record)
- or ... you could code `Age: IIf(IsNull([Date of Birth]), Null, myCalculateAge([Date of Birth]))` for the column in the query. IIF is also used on page 134 of McBride. Look it up in VBA Help; it can be very useful.

3.2.4 Showing the category type on the Membership form

The simplest method of doing this is probably to create a new query based on the Membership table and the Category table, with all fields from the Membership table plus the category type. Then change the form RecordSource property to use the new query, and add the category type field to your form.

Another way (and this could be useful if the table was a look-up table not linked to the main table) is to use the DLookup function (see Appendix H) to get a value to be put in an unbound text box (here called txtCategoryType) on the form.

Put the code in Fig 3.2.10 in your Membership Form_Current event.

Coding the IF check for the Category No is not strictly necessary here, as the function will return a Null value if the required Type is not found.

```
txtCategoryType = Null    'clear any previous text
If [category no] <> 0 Then 'check field has a valid value – zero is the table/field default for new records
    'get Category Type from Membership Category table
    txtCategoryType = DLookup("[Category Type]", "Membership Category", "[Category No] = forms!Membership![Category No]")
End If
```

Fig 3.2.10 Using the DLookup function for the Category Type

The DLookup function is one of Access's built-in Domain Aggregate Functions (see Appendices G.6 and H.6) and is the VBA code equivalent of a SELECT SQL statement which returns just a single value. The equivalent SQL would be:

```
SELECT [Category Type]
FROM [Membership Category]
WHERE [Category No]=[forms]![Membership]![Category No];
```

If you have problems getting one of these Domain Aggregate Functions to work (or problems understanding how they work) it can help to create the SQL (via the Query Design Window if you find that easier) as a check on your VBA code. See Fig 3.2.11.

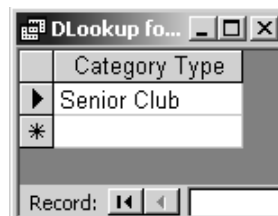
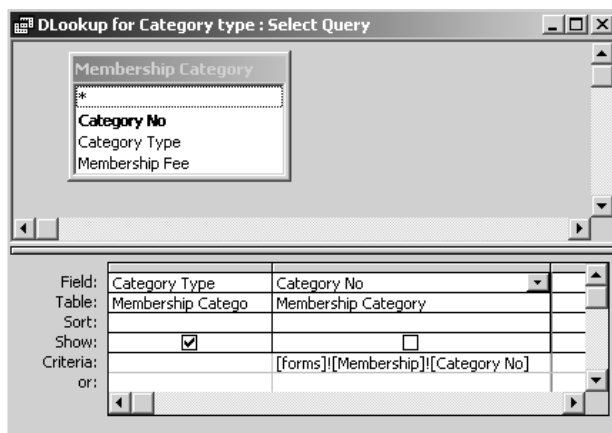


Fig 3.2.11 Query created from the SQL shown above, to show how the DLookup function works. The Membership form is as in Fig 3.2.9.

The DLookup function has the basic format: **DLookup(expression, domain, [criteria])**

- **Expression** = the name of the table field that you are looking for. The SELECT part of the SQL statement.
- **Domain** = the name of the table or query. The FROM part of the SQL statement. Note that you can also create a query to select the rows, then use that query directly in the Domain Aggregate function.
- **Criteria** = the criteria that you wish to apply. The WHERE part of the SQL with exactly the same format. This parameter is optional.

For further information, look the function up in VBA Help.

The criteria expression uses the Forms Collection. If you are not already familiar with this, look at Appendix I.

Finally, the call to the DLookup function also needs to be coded in the Category_No_AfterUpdate event (and the cmdCancel_Click event if you have done exercise 2.7.2), so that any change to that value will also pick up the new Type.

3.3 Validations

3.3.1 Field Validations

It is possible to do some validations by specifying certain properties in a field definition for a table. For example, for the Date of Birth field in the Membership table you could specify field properties as shown in Fig 3.3.1.

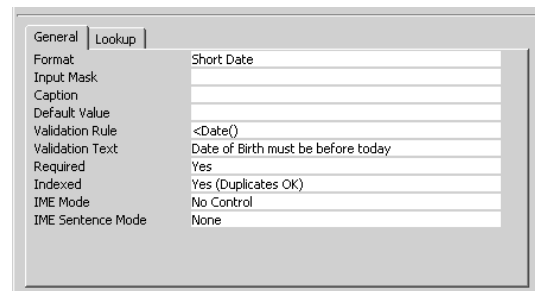


Fig 3.3.1 table field properties for Date of Birth

Required = Yes means that the user must enter a value in the field. If the user starts to enter something, then deletes it, or deletes an existing value, then Access will provide the error message shown in Fig 3.3.2. The same message is shown if the user never enters anything in the field and an attempt is made to save the record.

This message is reasonably understandable but you may wish to replace it with a message that is a bit friendlier and less technical.

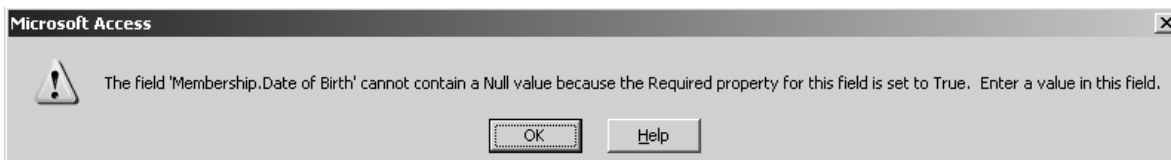


Fig 3.3.2 Access error message when required field is left empty

The validation rule and text mean that the user must enter a value that makes sense in the context. The validation condition here is checking that a member's date of birth cannot be in the future. If the value entered by the user violates the validation condition, then Access uses the text that is provided for the error message. If no text is provided then default text is used. See Fig 3.3.3.

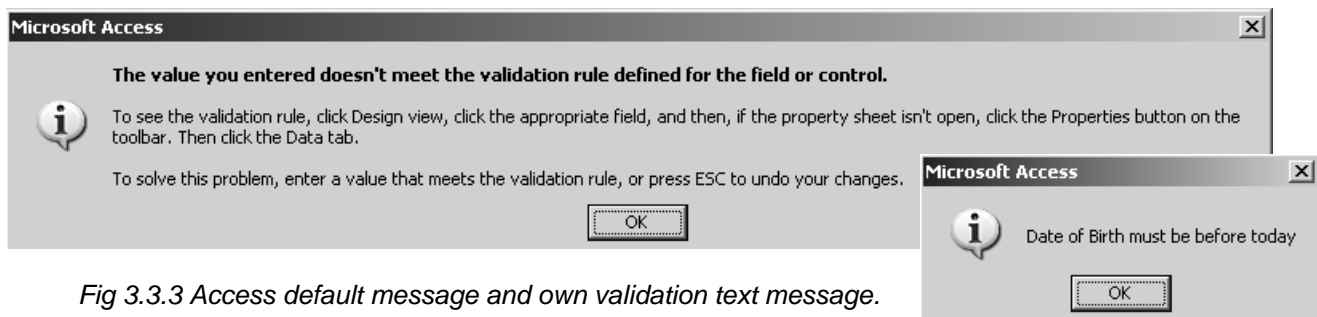


Fig 3.3.3 Access default message and own validation text message.

If you wanted to check that a member must be, say, at least 5 years old, then you would code
`<DateAdd("yyyy", -5, Date())`
 for the validation rule. DateAdd is one of Access's built-in functions (see Appendix H.1 and/or VBA Help).

The Date of Birth field is defined as a Date/Time datatype, so Access will check that the value is a valid date. The error message that Access provides is reasonable (see Fig 3.3.4), but you may wish to replace it with a message of your own.

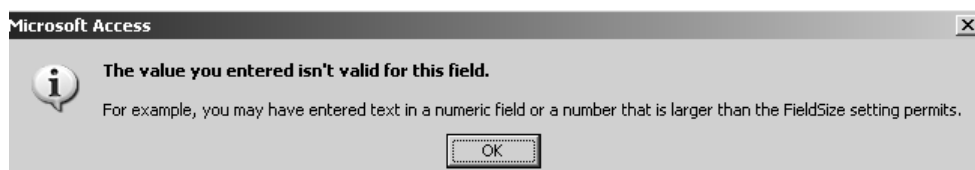


Fig 3.3.4 Access message when data entered doesn't match the datatype

Instead of putting these checks in the table definition, you could code them in a field `BeforeUpdate` event. This event works in much the same way as a `Form_BeforeUpdate` event, in that the code is executed before the value in the field is updated.

- Change your Date of Birth table field definition properties to show:
 - o Required = No (to avoid the message shown in Fig 3.3.2).
 - o Datatype = Text (to avoid the message shown in Fig 3.3.4).
 - o Validation rule and text both blank (all validations will now be coded).
- Delete the current Date of Birth field from the Membership form and recreate it from the field list.
 - o If you are using Access 2003 you may not need to recreate the form field, as one of the new Access 2003 features is that any change to a table field should be automatically reflected in all bound fields.
 - o If you have recreated the field then...
 - ... set the tab order to show the Date of Birth field as coming after the Occupation field (see *View→Tab Order*).; the new version of the field will have been given a tab order after all the controls already on the form.
 - ...link the Date of Birth `AfterUpdate` event (from section 3.2.3.2 back to the code (use the Event tab on the property box).
- Create a `BeforeUpdate` event for the Date of Birth field and enter the code shown in Fig 3.3.5.

```
Private Sub Date_of_Birth_BeforeUpdate(Cancel As Integer)
'validate the data entered in the field

Dim dtDateOfBirth As Date

Cancel = True           'assume value will be invalid
If IsNull([date of birth]) Then 'is field blank?
    myDisplayWarningMessage "Please enter a value for Date of Birth"
Elseif Not IsDate([date of birth]) Then 'is date valid?
    myDisplayWarningMessage "Please enter a valid Date of Birth in dd/mm/yyyy format"
Else
    dtDateOfBirth = [date of birth] 'convert from text to date format
    If dtDateOfBirth >= Date Then 'is date sensible?
        myDisplayWarningMessage "Date of Birth must be before today"
    Else
        Cancel = False           'value is OK
    End If
End If

End Sub
```

The Cancel parameter is for you to tell Access whether or not you want to save the new value in the field (i.e. to update the field). It is initially set to False (by Access) which means that the update is to go ahead. You can set it to True if you want to cancel the update and this will require the user to enter another value before being allowed to move on.

Test No	Data	Reason for test	Expected result
1	Enter something in the date field , then remove it, and tab out of the field.	Leaving field empty (see also test 7).	Message "Please enter a value for Date of Birth", and user cannot move on until date is correct.
2	Something that is not a date ('fghghg' for example)	Not a valid date.	Message "Please enter a valid date of birth in dd/mm/yyyy format", and user cannot move on until date is correct.
3	A date that does not exist, such as 29/02/2003, 32/12/2004, 9/13/2002	Correct date format, but not a valid date.	As test 2.
4	A valid date after system date: <ul style="list-style-type: none"> • 1 day after (boundary) • =system date (boundary) • a longer time after (non-boundary) 	Valid date, but not sensible.	Message "Date of Birth must be before today" and user cannot move on until date is correct.
5	A valid date before system date: <ul style="list-style-type: none"> • 1 day before (boundary) • a longer time before (non-boundary) 	Date OK.	Date accepted. No error message and can now move on.
6	Create a new record but don't attempt to enter a date of birth.	Leaving field empty (see also test 1)	<i>See last-but-one paragraph in this section. What happens here will depend on what you have done to trap this occurrence.</i>

Fig 3.3.5 Validating the date of birth, and a test plan.

The `BeforeUpdate` event has a parameter called `Cancel` which is defined as an Integer datatype. It is actually a Boolean variable, and uses the values `True (-1)` and `False (0)`. (It seems strange not to define it as such...).

- The default value (set by Access) on entry is `False (0)`.
- If you set the value to `True (-1)` then the cursor will be positioned back in the field automatically and the user will not be able to move on until a correct value has been entered in the field.
- If you set the value to `False` (or leave the original value unchanged, which is the same thing) then this means that you are telling Access that the value is OK and the field can now be updated. Any code in the field `AfterUpdate` event (see section 3.2) will now be activated.

The code for the Date of Birth `BeforeUpdate` event will now do the following validations:

- First it will check that something has been entered in the field.
 - o You have already seen the function `IsNull` in section 3.2.3.2. see also Appendix H.5.
- If something has been entered then it is checked to see if it is a valid date.
 - o `IsDate` is another of Access's built-in functions; see Appendix H.5.
- If a valid date has been entered then it is checked to see if it is a sensible date. Here the check is to ensure that the date of birth is before today's (the system) date.
 - o But the text box form field on the form does not have a specified datatype, so first the text entered must be converted to a date; an easy way to do this is to move it to a date variable. Then it can be compared with a date value, in this case the system date returned by the built-in function `Date()`. See Appendix H1.
 - Alternatively, you can use the conversion function `CDate` (see Appendix H7 and section 3.3.2).

All the messages that will be displayed, as they are using the `myDisplayWarningMessage` procedure, will have the same standard format with the Chelmer Leisure name in the header.

One condition that would not be trapped by the above coding would be where the user has been entering data in a new record and has simply tabbed over the Date of Birth field and left it blank (see test 6 in Fig 3.3.5). You could then save the record with this field as blank. To trap this situation you could put suitable code in the `Form_BeforeUpdate` event (see section 3.3.2). If you are validating data entered into a bound field then it may be simpler to rely on the `Required = Yes` property in the field table definition. An alternative method is discussed in section 2.3.3 of the 'Further VBA' Trainer, to use error handling to trap this error yourself and take appropriate action. If you are validating data entered into an unbound field (for example, for a parameter value – see section 3.3.3) then the `Required` property does not apply (though you can set a `ValidationRule Of Is Not Null` if you wanted to).

The above coding was designed to illustrate the three standard tests that you may need for data entry; is anything entered; is it the correct datatype; is it a sensible value (in that order).

3.3.2 Form validations

In an exactly similar way to the above, you can use the `Form_BeforeUpdate` event to undertake any validations on the form that apply to more than one field (for example, checking that the Date of Birth is less than the Date of Joining) or checking that required fields have had data entered..

The code in Fig 3.3.6 sets the value in `Cancel` to `True` if the validation condition is violated; this will prevent the user moving on until the data has been corrected. The cursor is moved to one of the two fields in question, ready for the user to correct either this date, or the other date (or even both of them!).

The new code to be added to your `Form_BeforeUpdate` event (see Fig 2.5.4) is shown in bold. Note that this code makes further use of the `Cancel` parameter.

Note the use of the built-in conversion function `CDate` (see Appendices F.1.1 and H.7). In Fig 3.3.5 you saw how to convert a datatype value by moving it to a variable of the required datatype; use of a conversion function such as `CDate` is an alternative method.

Access does some data conversions for you, but you are normally unaware that these conversions are going on. The values in the two date text box fields must here be converted to date datatypes to compare them correctly, as dates need to be compared as date formats not as strings.

```

If myYesNoQuestion("Save Changes?") = vbNo Then
    ... leave existing coding in here ...

Else

    If IsNull([date of birth]) Then
        'check to see if Date of Birth is null
        myDisplayWarningMessage "please enter a date of birth"
        [date of birth].SetFocus
        Cancel = True

    Elseif CDate([date of birth]) > CDate([date of joining]) Then
        'validate date of birth w.r.t. date of joining
        'must compare in date formats not as in textbox
        myDisplayWarningMessage "date of birth must be before date of joining"
        [date of birth].SetFocus
        Cancel = True
    Else

        ' Access will automatically save the record
        ... leave existing coding in here ...

    End If

End If

```

Fig 3.3.6 form validation to check for Null date of birth and to compare Dates of Birth and Joining

3.3.3 Parameter Validations

As you should already be aware, an unbound textbox on a form can be used for a parameter value. For example, look at the additional queries in Task 5 of Unit 13 of *McBride*. These tasks ask for specific values for the query criteria, but in practice, the user may want the flexibility to change the criteria each time the query is run. Thus, a query may list all members in a given age range, or who joined in a given year, or who smoke/don't smoke, or who have a given sporting interest. The 'given' value is one that is supplied at run-time, so cannot be specified in the query at design-time.

3.3.3.1 Single parameter

Consider Unit 13 task 5 additional query number 3: *'which members joined the Centre between 1/1/99 and 1/1/00?'*

Fig 3.3.7 shows a simple query that lists some basic details of members who joined in a given year. It has a calculated column which uses the built-in function Year (see Appendix H.1) to select just the year part of each date of Joining, and uses a parameter in a field called txtYear on a form called Parameters. The command button has the name cmdMembersInYear.

Note that the button to run the query is disabled initially; the Enabled property has been set to No in the field property box.

It is possible to set the field display format to a number format, and Access will check that the value entered is numeric (see Fig 3.3.4). But doing this will not check that the value is an integer (setting the number of decimal places to zero will only affect the display format), nor that the value is sensible. You could also use an input mask, but violating this causes a very user-unfriendly standard Access error message.

It is also possible to set a validation condition in the ValidationRule property to check if the field has been left empty (Is Not Null), with a suitable message, but this message will have an Access error message format (which is fine if that is what you want).

The coding in Fig 3.3.9 shows how to use the BeforeUpdate and AfterUpdate events for txtYear to validate the value entered by the user and to enable the command button if the value is OK. It assumes that the date must be in the range 1990 to the year of the system date. If an incorrect value has been entered then the appropriate error message should be displayed, as shown in Fig 3.3.8.

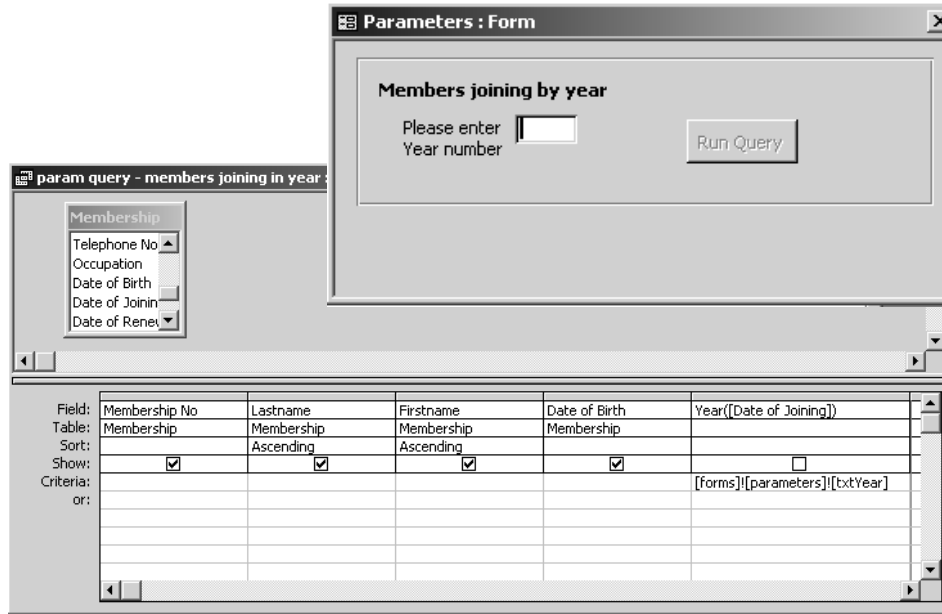


Fig 3.3.7 Parameter query taking an integer value from a form

Test No	Data	Reason for test	Expected result		
			Message	Button	Other
1	Non-numeric values bbb, 11/12/2004	Non-numeric value	Message "Please enter a numeric value".	Disabled.	User cannot move on.
2	Remove the value entered.	Field empty.	Message "Please do not leave empty".	Disabled	User cannot move on.
3	1800, 1956 1989, 2005 2010, 2100	Range-checking: Outside (non-boundary) Outside (boundary) Outside (non-boundary)	Message "please enter a whole number between 1990 and yyyy" (where yyyy = year of system date.)	Disabled	User cannot move on.
4	1992.5	Only integer values allowed.	Message as for test 3.	Disabled	User cannot move on.
5	1990, 2004 1992	Valid value: On range boundaries Inside range	Value accepted.	Enabled	User can run query.

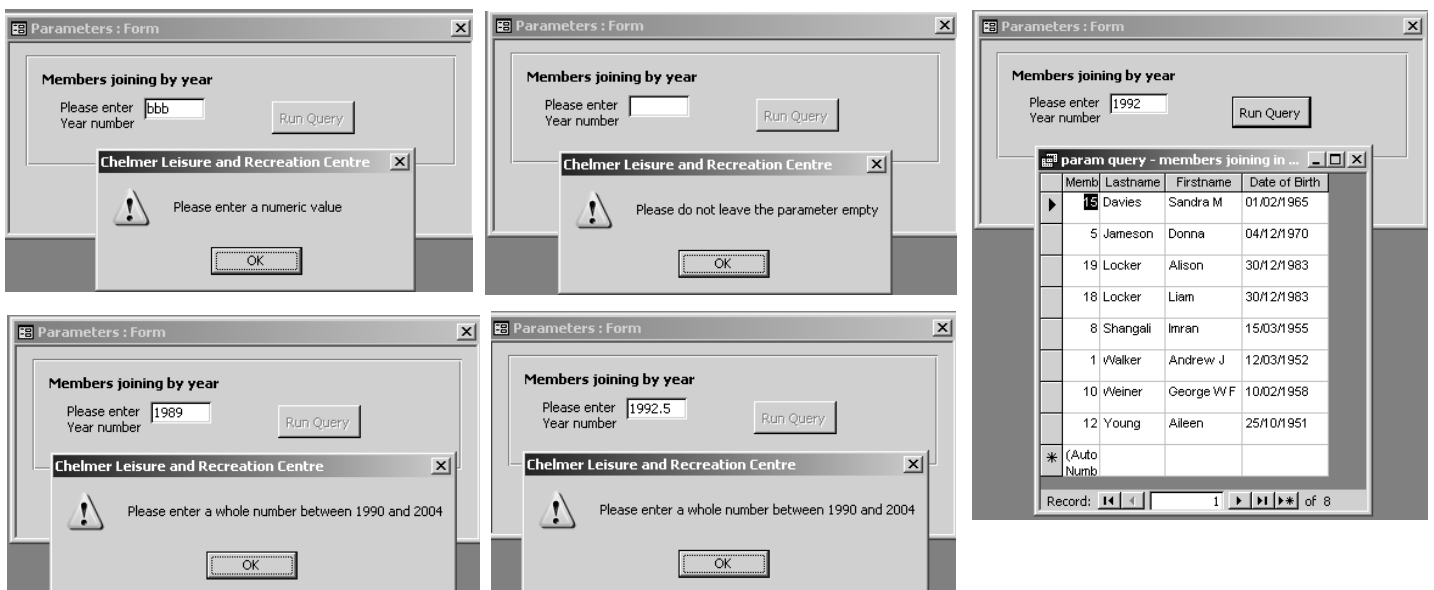


Fig 3.3.8 various error values and error messages, one OK value and result (run date = Aug 2004)

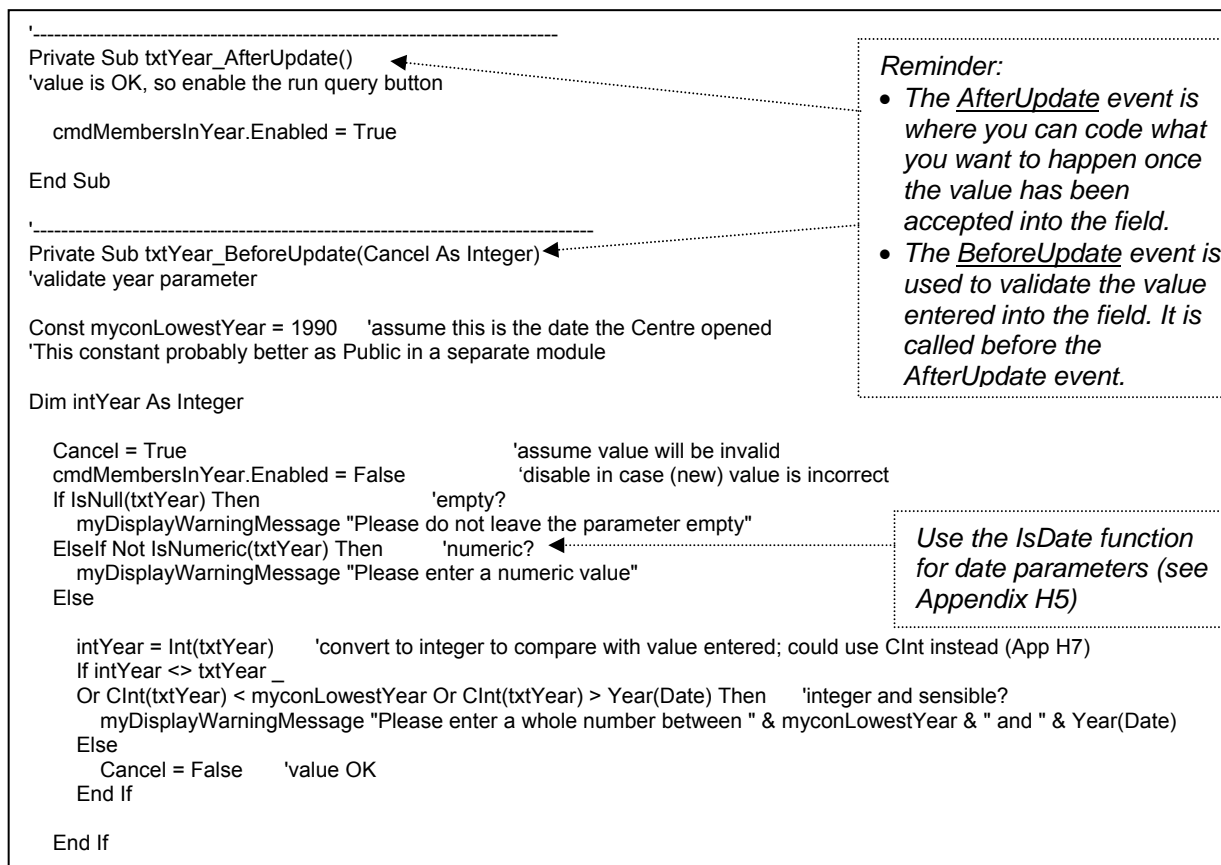


Fig 3.3.9 Using the field *BeforeUpdate* and *AfterUpdate* events to validate a form parameter

Points to note:

- The `IsNull` built-in function is used to check if the field is empty (this will happen if the user has entered a value then removed it). See Appendix H.5.
- The `IsNumeric` built-in function is used to check that what has been entered is a number. See Appendix H.5.
- But if it is a number it could be a fractional number.
 - o The `Int` function (see Appendix H.3) is used to convert the value to an Integer; it removes the fractional part of the number. The integer value is put into an integer variable `intYear`. If the value typed in by the user does not match the integer value in this variable, then the value typed in was not an integer. For example, if the user typed in 1992.5, then `intYear` will contain 1992. The line


```
If intYear <> txtYear
```

 will mean if 1992 = 1992.5....
- The line has been continued (using space then underline) to the range check conditions on the next line, to check that the value is a sensible value.
 - o The range check assumes that the Centre opened in 1990, and uses the year of the system date to get the upper end of the range. The conversion function `CInt` has been used to ensure that the datatypes on each side of the comparison expressions are the same. See Appendix H.7.
 - o The error message displays the values required for the range. These values may also be useful displayed on the form to guide the user.
- All error messages are displayed using the `myDisplayWarningMessage` procedure, so have a common format with the Chelmer Leisure title in the caption.
- As you have seen before, the `BeforeUpdate` event is used to validate the data, and the `AfterUpdate` event has been used to take action once the data is OK.
 - o In this simple example, the single line in the `AfterUpdate` event could also be coded with the `Cancel = False` line in the `BeforeUpdate` event.

3.3.3.2 Two parameters for a value range

If the Leisure Centre wanted to know who had joined between any two given years, say, between 1992 and 1995 (the start of 1992 and the end of 1995, that is), the form and query could be as shown in Fig 3.3.10.

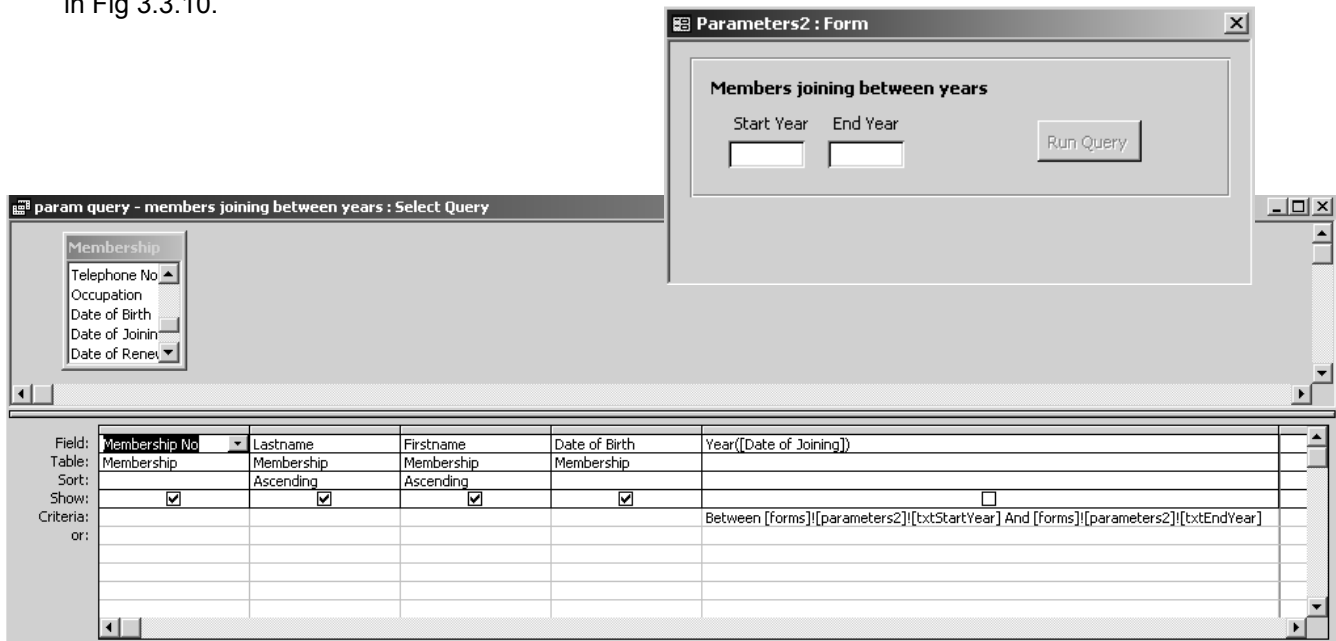


Fig 3.3.10 Parameter query taking two integer values from a form for a range check

This is very similar to the query and form in the previous section, but now there are two parameters and the query checks dates with years between the two values. Each year value will need to be validated as before, but there are a couple of added complications:

- You need to check that the start year is \leq the end year.
- You need to know that each parameter is correct and the relationship between them is correct before you can enable the command button to run the query.

But where do you put these checks? A logical place, at first sight, would appear to be in the `AfterUpdate` code for the second date. But what if the user enters the second date first? (Unlikely, yes – but users often do not use a system as we expect them to). And if the second date is less than the first date, perhaps it is the first date that is wrong and needs correcting. These checks therefore need to go in the `AfterUpdate` code for both the year fields. See Fig 3.3.11.

The coding in Fig 3.3.11 uses two Boolean variables as flags. You have already seen this programming technique in section 2.6.1. There are also two Private procedures.

- The flags are set to `False` before each field validation and to `True` when the field has been validated as OK. Then each field `AfterUpdate` event checks to see if both flags are true. If this is so, then both values have been validated as OK and the further checks can take place.
 - o The further check compares the two years to check that the start year \leq the end year. If this is not so then an error message is displayed and the cursor is positioned to one of the fields ready for the user to correct at least one of the values.
 - o If the check is OK, then the command button is enabled and the query can be run.
- There are now two parameter fields and each will need to be validated using very similar code. Rather than copying-and-pasting the code from one into the other (a common, but misguided, action that many students use!) a common procedure `myValidateYear` has been created that does the validation. There are three parameters:
 - o `prmYear` as Variant – the value entered into the textbox, passed by the calling code, and to be validated by `myValidateYear`.
 - o `prmCancel` As Integer – the value in the `BeforeUpdate` Cancel parameter, to be set by `myValidateYear`.
 - o `prmOK` As Boolean – value for the relevant flag, to be set by `myValidateYear`.
- A common procedure `myCompareValues` to perform the checks has been created which can be called from each `AfterUpdate` event.


```

'used to check both values are OK before enabling the command button
Dim bStartYearOK As Boolean
Dim bEndYearOK As Boolean

Const myconLowestYear = 1990 'assume this is the date the Centre opened
'This constant probably better as Public in a separate module
'-----
Private Sub txtEndYear_BeforeUpdate(Cancel As Integer)
'validate end year parameter
  myValidateYear txtEndYear, Cancel, bEndYearOK
End Sub

'-----
Private Sub txtEndYear_AfterUpdate()
  myCompareValues
End Sub

'-----
Private Sub txtStartYear_AfterUpdate()
  myCompareValues
End Sub

'-----
Private Sub txtStartYear_BeforeUpdate(Cancel As Integer)
'validate start year parameter
  myValidateYear txtStartYear, Cancel, bStartYearOK
End Sub

'-----
Private Sub myCompareValues()
'values are OK, so compare them to see that start year <= end year

  If bStartYearOK And bEndYearOK Then 'each value has been validated as OK

    If txtStartYear > txtEndYear Then 'check values compare appropriately
      myDisplayWarningMessage "Start Year must not be after End Year"
      txtStartYear.SetFocus
    Else
      cmdMembersBetweenYears.Enabled = True 'OK - can run query
    End If

  End If
End Sub

'-----
Private Sub myValidateYear(prmYear as Variant, prmCancel As Integer, prmOK As Boolean)
'used to validate the values in txtStartYear and txtEndYear

Dim intYear As Integer

  prmCancel = True '(assume value will be invalid)
  cmdMembersBetweenYears.Enabled = False
  If IsNull(prmYear) Then 'empty?
    myDisplayWarningMessage "Please do not leave the parameter empty"
  ElseIf Not IsNumeric(prmYear) Then 'numeric?
    myDisplayWarningMessage "Please enter a numeric value"
  Else

    intYear = Int(prmYear) 'convert to integer to compare with value entered
    If intYear <> prmYear _
    Or CInt(prmYear) < myconLowestYear Or CInt(prmYear) > Year(Date) Then 'sensible?
      myDisplayWarningMessage "Please enter a whole number between " & myconLowestYear & " and " & Year(Date)
    Else
      prmCancel = False 'value OK
    End If

    prmOK = Not (prmCancel) 'to return True if validation OK, otherwise false
  End If
End Sub

```

Validate end year parameter.

Validate start year parameter.

Common procedure to compare the values in the two parameters.

Common procedure to validate each parameter value.

Fig 3.3.11 A method of validating two parameter values against each other

In the case of this particular example you could set a default value for each date; for example set the start date to 1990 and the end date to the year of the system date. You can remove the need for the Boolean flags and can enable the command button from the start. This change may not be possible in all similar situations.

To do this, code as shown in Fig 3.3.12 in the `Form_Load` event, and remove all mention of the Boolean variables `bStartYearOK` and `bEndYearOK`.

```
Private Sub Form_Load()
'put default values in the textboxes
'convert them to text for correct and easier comparisons with the text entered by the user.

    txtStartYear = CStr(myconLowestYear)
    txtEndYear = CStr(Year(Date))

End Sub
```

Fig 3.3.12 Code for `Form_Load` event to set default values for the two parameters

Whichever method of coding you use, you must test...

- ...each field as in the test plan shown in Fig 3.3.8
- ...pairs of values to test comparisons:
 - o start year < end year (OK)
 - o start year = end year (OK)
 - o start year > end year (invalid)

3.4 Searching for Records

3.4.1 Replacing the Navigation Bar Functions

3.4.1.1 Next/Previous records

Up till now you have moved from record to record by using the Access navigation controls at the bottom of the form. There are command button wizards for various navigation controls, so add your own buttons for *Next* and *Previous* record to the form; you should know by now how to do this. Remember to add the new buttons to `myResetButtonsToOff`.

Try getting the next record at the end of a record set, or a previous record at the start of a record set. You will see (if you debug whilst trying this) that the error procedure within the event is invoked. Using the Debugger (the Watch facility, with the cursor positioned on the word `Err` is useful) find out the value that is in `Err`, and replace this message with a more meaningful one of your own, using your `myDisplayWarningMessage` procedure. (Hint: look at the coding in section 2.5.2 for the `Form_BeforeUpdate` event, where you suppressed the `DoMenuItem` cancelled message).

Now that you have your own buttons for *Next* and *Previous*, turn the form navigation controls off (see the Format tab for the form's property box). The navigation controls allowed the user to move around the set of records, but you have now replaced most of these functions with command buttons.

3.4.1.2 Count of records

The navigation controls also showed you the total number of member records, but this is now not shown.

In section 3.2.4 you saw a use of the Domain Aggregate Function `DLookup`. Another very useful one of these functions is `DCount`. See Appendix H.6.

- If you were to list all the membership numbers, the SQL you would use is:
SELECT [Membership No] FROM Membership;
- To count the rows you would use an aggregate function:
SELECT Count([Membership No]) AS [CountOfMembership No] FROM Membership;
and this is what the `DCount` function does.

Add an unbound textbox to the header of your Membership form. Call it txtTotalMembers, and set suitable properties (as it is not a data-entry field). Code the following line in the Form_Load event:

```
txtTotalMembers = DCount("[Membership No]", "Membership") 'optional criteria omitted
```

or

```
txtTotalMembers = DCount("[Membership No]", "Membership", "") 'optional criteria is the empty string
```

These two lines are equivalent; a missing criterion and the empty string amount to the same thing.

If you have problems getting one of these Domain Aggregate Functions to work (or problems understanding how they work) it can help to create the SQL (via the Query Design Window if you find that easier) as a check on your VBA code. See Fig 3.4.1.

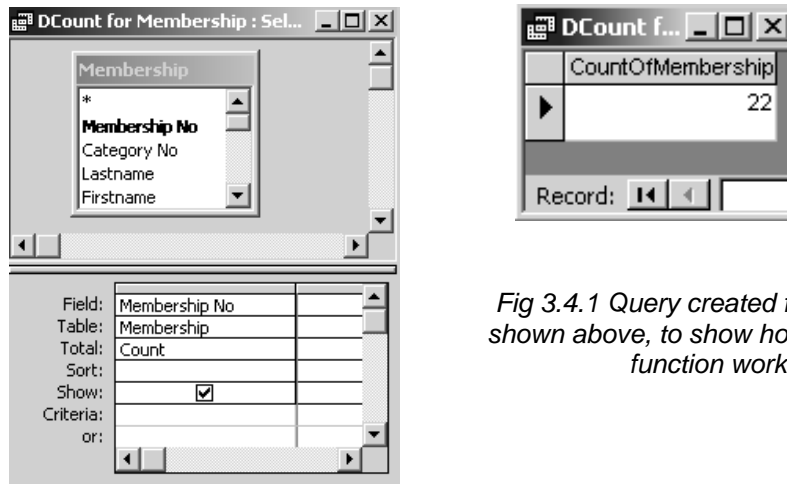


Fig 3.4.1 Query created from the SQL shown above, to show how the DCount function works.

The DCount function has the basic format: **DCount(expression, domain, [criteria])** (that is – the same as the DLookup function used in section 3.2.4).

- **Expression** = the name of the table field that you are looking for. The SELECT part of the SQL statement, but excluding the aggregate function itself as the counting of the rows is done by the function.
- **Domain** = the name of the table or query. The FROM part of the SQL statement. If you wished to use a criterion, you could create a query to select the rows, then apply the Domain Aggregate function directly to the query.
- **Criteria** = the criteria that you wish to apply. The WHERE part of the SQL with exactly the same format. This parameter is optional, and is not used here as we wish to count up all the records in the Membership table.

Your form will now show the total number of members in the Membership table. See Fig 3.4.3. You will also need to code the line shown above in the Form_AfterUpdate event (for when a new record is added) and when a record is deleted.

You could also create a common procedure myShowMemberCount, code a DCount statement in there and call the procedure whenever needed. See Fig 3.4.2.

```
Private Sub myShowMemberCount(prmFilter As String)
    'prmFilter is a WHERE condition, used when records are filtered

    txtTotalMembers = DCount("[Membership No]", "Membership", prmFilter)

End Sub
```

Fig 3.4.2 Common procedure to count records for the form

To call the procedure to count all the records, simply code

```
myShowMemberCount ""
```

the "" represents the empty string, and means that there is no WHERE condition.

The point of the parameter prmFilter will become apparent in section 3.5.

3.4.2 Looking for a Particular Record

With a very small data set, it is possible to trawl through all the records until the required record is found. With a large data set (such as a membership list) it is unacceptable to require the user to do this. So do the following:

- Open the Membership form in design mode and, using the toolbox wizard, create a combo box on the form. (You may need to shuffle your command buttons around to make room). See Fig 3.4.3.
- Choose to have the combo box used to find a record on the form (the third option you are given), and select Lastname, Firstname and Street from the list given (the list of fields in the table). The Membership Number will be chosen automatically as this is the Primary Key.
 - o Note that the form must be based directly on a table or query. Check the form RowSource property; if this is an SQL statement then you won't get the third option with the wizard. If you do have an SQL statement here, use it to create a new query, then alter the RowSource property (use the drop-down box provided) to reference the new query.
 - o For more details see Access FAQ 29 on the Frequently Asked Questions page of <http://www.cse.dmu.ac.uk/~mcspence/Access.htm> ,
- Choose suitable text for the label, e.g. Find by Name.

You will now have a combo box on the form that will display a list of names from the Membership table. If you look at your code you will see that AfterUpdate code has been generated to find the record chosen from the drop-down box. But this doesn't work properly yet (try choosing a name, then choosing a second name) and the name generated (something like Combo99) isn't very meaningful, so:

- Change the name to (for example) cboFindName. Make this change in the property box and in the AfterUpdate code (**two** places in the code!). Check the property box – you may need to reinstate the link to the AfterUpdate event code.
- The reason the code doesn't work properly is that we have set the AllowEdits property to False for each new record, so we cannot update the combo box once we have moved to a new record.
 - o Create code for a GotFocus event for the combo box. Add code in here to set the AllowEdits property to True. In exactly the same manner, create code for a LostFocus event on this box, to set the property back to False (just in case the user doesn't move to another record after all).

You can now use the box to move to a new record, and the code has a meaningful name. Remember to use the *Help* system to check on anything that you do not understand.

Your form with the combo box should now look something like Figure 3.4.3. You can search on other fields in exactly the same manner.

Lastname	Firstname	Street
Harris	David J	55 Coven Road
Jameson	Donna	25 Alder Drive
Jones	Edward R	17 Mayfield Avenue
Locker	Alison	2 Beech Close
Locker	Liam	2 Beech Close
Perry	Jason R	59 Church Street
Robinson	Petra	16 Lowton Lane

Figure 3.4.3 Form with Find by Name combo box (run date = August 2004)

Look at the property box for your *Find By Name* combo box (see Fig 3.4.4). There is also further useful information about combo boxes in both Access Help and VBA Help, and in section 3.6 of this document.

- The BoundColumn property (Data tab) of the combo box is set to 1. If you look at the RowSource property (also on the Data tab) you will see that the SQL on which the combo box is based has the Membership No as the first column, as this is the Primary Key. In order to reference the combo box to put a value in it, or display the row found, you simply need to use the combo box name. You can also use the Column property (not in the property box) to reference elements within a combo box. Try typing the following into the Event code and see what you get:

```
MsgBox cboFindName.Column(0) 'displays Membership No
MsgBox cboFindName.Column(1) 'displays Lastname
MsgBox cboFindName.Column(2) 'displays Firstname
MsgBox cboFindName.Column(3) 'displays Street
```

- The Membership No, although in the SQL, does not show in the drop-down list (see Fig 3.4.3). If you look at the ColumnWidths property you will see that the width for the first column is 0 (zero); this hides the column.
 - You can use this property to change the column widths. You may also need to adjust the ListWidth property.

- If you look at the event code for the combo box you will see the line

```
rs.FindFirst "[Membership No] = " & Str(Nz(Me![cboFindName], 0))
```

This line references the combo box name to get at the bound value assigned to it, viz. the Membership No. If the Membership No was, say 3, then this would equate to the following at run-time:

```
rs.FindFirst "[Membership No] = 3"
```

The code here is using a Recordset method to find the record with the required Membership No. Recordsets are discussed in the Further VBA Trainer.

Look at VBA Help to see what the Str and Nz built-in functions do. See also Appendix H.5.

- If the user subsequently moves to a new record, using the *Next Record* or *Previous Record* button, the text in the combo box is left unchanged. Add the following line to the

Form_Current event:

```
cboFindName = [Membership No] 'set combo box to current record
```

This line references the combo box to put a value in it, so that the combo box matches the current record.

- It could be confusing for the user to have a combo box with a white background the same as for data entry fields, so change the BackColor property to Transparent. (Format tab in property box).
 - Do not set the Locked property to Yes or you won't be able to use the box at all! (Try it).
- It would be more useful if the names were listed in alphabetical order of surname. Look again at the RowSource property (Data tab). Click on 'build' (...) at the right of the row and change the query to list the names in order. This will make finding a record even easier.
 - Entering the first character of the surname is a quick way to jump to the first surname starting with that first letter; the AutoExpand property is what allows this, so set it to No if you don't want this facility to be available.
- You can set the ColumnHeads property to Yes if you want column headings to show.

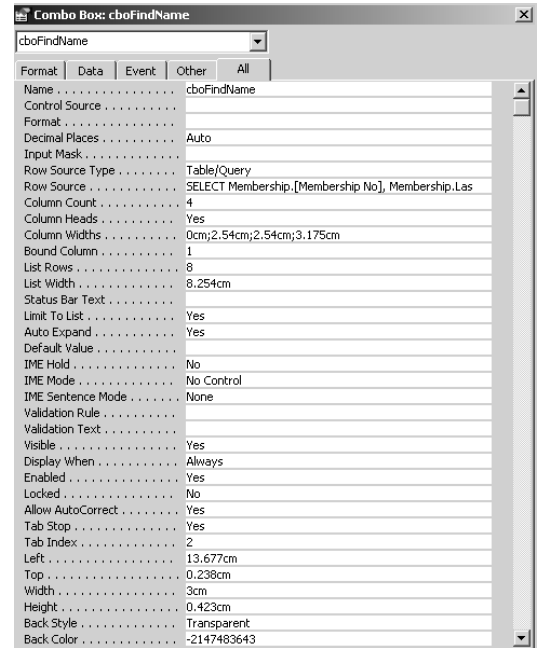


Fig 3.4.4 Combo Box properties

3.5 Applying a filter to a form (with a count of records)

There is a command button wizard for finding records, but this uses a pre-set Access dialog box and is not very user-friendly. This may or may not be suitable for a particular application.

However, it is actually quite simple to code filters for the form we have been using so far, to allow the user to select records according to specific requirements.

3.5.1 Filtering on a text (string) field with a wildcard

Suppose we want to provide a facility for users of the system to select all records where the Lastname begins with selected character(s).

- Create a text box anywhere on the form. Put suitable text in the label. Call the textbox txtFilterName. The user will enter the start character(s) of the required Lastname in this field.
- Create a non-wizard command button, call it cmdFilterByName and give it a suitable caption. Create a Click event for this new button and add the code shown in Fig 3.5.1.
- Create a GotFocus event for txtFilterName and set the AllowEdits form property to True, as the default (see Sections 2.2.1 and 2.2.3) is False.
- Test your code:
 - o Enter J in the textbox and click on the button. The form will now show the records for Jameson and Jones only. The *Next* and *Previous* buttons can be used to scroll through this list. See Fig 3.5.2.
 - o Enter Z in the textbox and click on the button. There are no names beginning with Z, so no records are found.

```

Private Sub cmdFilterByName_Click()
'filter records by surname
Dim strName As String      'to store user text and add wildcard
Dim strFilter As String    'filter condition

    strName = txtFilterName & "*"
    strFilter = "[Lastname] like '" & strName & "'" ← see explanation below – this has mix of single and double quotes
    DoCmd.ApplyFilter , strFilter

    txtTotalMembers = DCount("[Membership No]", "Membership", strFilter) ← Alternative methods of coding the
    myShowMemberCount strFilter ← See explanation below.

End Sub
'-----
Private Sub txtFilterName_GotFocus()
'need this to allow user to be able to enter a value in this field

    AllowEdits = True
'the property will be set back to false when the first record in the list is displayed
'(see Form_Current event).
End Sub

```

Fig 3.5.1 Code to select by string filter with wildcard

Explanation of each line of code for cmdFilterByName_Click in Fig 3.5.1:

- **Dim strName As String**
 - o A variable that will be used to store the user text and add a wildcard.
- **Dim strFilter As String**
 - o A variable that will be used to store the details of the filter. The contents of this can be checked at run-time in the Debugger, so you can see how this works.
- **strName = txtFilterName & "*"'**
 - o This takes the letter(s), if any, entered by the user and adds the wildcard * to the end of them, putting the result in the hidden field on the form.
 - o There is no need to convert the text to upper (or any other) case.
- **strFilter = "[Lastname] like '" & strName & "'"**
 - o This sets up the filter condition. If the user enters the letter J, strFilter will contain: "[Lastname] like 'J*". Note that this is just like the WHERE clause for SQL.
 - o As the user text is a string value you must put quotation marks around the value in strName.

- o The full string is made up of three parts, joined with the concatenation character &:
 - “[Lastname] like ‘ ” – note double quotation marks around the full string, and a single quotation mark at the end of the string which will come just before the value in *strName*.
 - **strName** – this will pick up the contents of the variable.
 - “ ‘ ” – a single quotation mark (enclosed in double quotation marks as it is a string character) to come at the end of the value in *strName*.
- o The code here uses single quotation marks around the value in *strName*, but it is also possible to use double quotation marks or a variable containing the character for quote. This is very clearly explained in VBA Help; use the keyword ‘quotation mark’.
- o It is not essential to use a variable, as the string “[Lastname] like ” & strName & ”” could be used directly on the next statement, but using a variable often makes debugging easier as you can look at the value in the string
- **DoCmd.ApplyFilter , strFilter**
 - o This line applies the filter and positions the form with the first record found.
 - o The form has a *Filter* property and this is set to the value in *strFilter*.
 - o The form also has a *FilterOn* property. This will now be set to *True*.
 - o By using the *Next* and *Previous* buttons the user can scroll through the filtered records.
- **txtTotalMembers = DCount(“[Membership No]”, “Membership”, strFilter)**
- **myShowMemberCount strFilter**
 - o The first line uses the *DCount* function to count up the records, putting the value in the field already used for this purpose in section 3.4.1.2. It doesn’t really matter which field is chosen for the first parameter, as the function simply counts up the rows.
 - o The second line is an alternative to the first line, and uses the procedure created in Fig 3.4.2. The value to be passed as a parameter is the *WHERE* condition already set up in *strFilter*.
 - o Both lines (re)use the value in *strFilter*; another reason for putting the condition in a variable.
 - o If you wanted to display a message as well when there are no records to show, you could code the following at the end of *cmdFilterByName_Click()* in Fig 3.5:


```
If txtTotalMembers = 0 Then 'no record shown
    myDisplayInfoMessage (“There are no names starting with “ & txtFilterName)
End If
```

Fig 3.5.2 showing the result of filtering for names starting with ‘J’.
(Using lower case ‘j’ would give the same result).

There have been no validations coded here to check what the user has entered in *txtFilterName*, as the value is a string value so could take any characters. In assignments and projects you should add checks as listed below (you should know by now how to do most of them):

- Set an input mask to check for alphabetic characters.
- Create your own function to check that the characters are all within the required character codes for A-Z, a-z. There does not appear to be a built-in function to do this. One method could be to use the *UCase*, *Len* and *Mid* functions (see Appendix H.2) to code a loop (see Appendix F.3.3) to check each character for a valid value in the range “A” to “Z”. A possible function header could be:


```
Public Function myIsAlphabetic(prmString As String) As Boolean (See exercise 3.7.8).
```
- Check that the field is not null.
- Have the command button disabled initially and only enable it when a valid, non-null, value has been entered.

All the above has been coded in a Click event for a command button, but could be coded in an AfterUpdate event for the textbox instead, removing the need for a command button.

The example above looked for a match with the start characters of a field, but by putting a wildcard at the front of the text as well you could look for a match anywhere in a field. For example, you could filter all records for members who have chosen 'aerobics' as one of their sporting interests, by looking for the text anywhere in the Sporting Interests field. You must test fields where the required character(s) is (are) at the start, middle and end of the field, also where there is no match.

3.5.2 Filtering on a text (string) field for an exact match

If you are looking for an exact match, then no wildcard would be necessary, so you can reference the form field directly for the filter condition.

An example of where this might be of use is for a field such as Town or County. You could create a wizard combo box on the form showing the values currently in use (base the combo box on a query selecting the relevant field from the Membership table, and add DISTINCT to the SQL to remove duplicate rows). Set the LimitToList property to Yes to prevent the user entering any other value. Then use the value selected by the user for the filter. Possible code is shown in Fig 3.5.3.

This code is much simpler as there is no need to add a wildcard to the value.

The filter condition shows how to reference the combo box (here given the name cboTown) using the Forms Collection reference (see Appendix I). The format is exactly as you would use if putting this as a criteria condition in a query.

```
Private Sub cmdFilterByTown_Click()
'filter records by Town
Dim strFilter As String

    strFilter = "[Town] = forms!membership!cboTown"
    DoCmd.ApplyFilter , strFilter

    txtTotalMembers = DCount("[Membership No]", "Membership", strFilter) 'or myShowMemberCount strFilter
End Sub
```

Fig 3.5.3 Filtering with an exact match

You could also code `strFilter = "[Town] = " & cboTown & ""` with embedded quotation marks, as used in the example in section 3.5.1.

3.5.3 Filtering on a numeric field

It may be useful to show all members in a certain category. This can be done in exactly the same way as in Section 3.5.2, by creating a combo box this time based on the Category Type from the Membership Category table, with the LimitToList property set to True. See Fig 3.5.4.

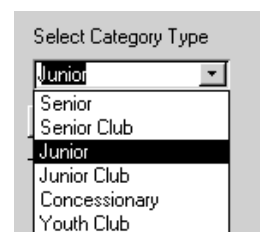


Fig 3.5.4 Combo box for the Category Type

Although the combo box will show the text for the Type, the value associated with the control is the key value of Category No as this will be the bound column (see section 3.4.2). The line to set up the filter condition could be:

```
strFilter = "[Category No] = " & cboCategory
```

As the Category No is a numeric value, no quotation marks are required around the value. If you use the Debugger to look at the contents of strFilter at run-time, you will see that it contains

```
"[Category No] = 3"
```

If you prefer to code using the Forms Collection reference, then you would code:

```
strFilter = "[Category No] = forms!membership!cboCategory"
```


3.5.4 Filtering on a Yes/No field

Suppose the Centre wishes to filter all members who are smokers. This filter will not require a textbox (or similar) control, but will only need a command button. The code to set up the filter condition will be:

```
strFilter = "[Smoker] = Yes"
```

If you want the Centre to be able to select smokers or non-smokers, add a check box (or a radio button group if you prefer). The code to set up the filter condition would then be:

```
strFilter = "[Smoker] = " & chkSmoker
```

where `chkSmoker` is the name of the check box.

No quotation marks are needed for a Yes/No field.

3.5.5 Filtering on a date field

Suppose the Centre wishes to filter to see who has joined since a certain date (perhaps to see who has joined in the past week). In this case the filter condition would be set by:

```
strFilter = "[Date of Joining] >= #" & txtJoinDate & "#" OR strFilter = "[Date of Joining] > #" & txtJoinDate & "#"
```

depending on the exact condition required. (`txtJoinDate` = unbound text box on the form for user date).

The date is a Date/Time value so needs to be enclosed within # marks. At run time, if the user entered the date 1/1/1992, `strFilter` would contain:

```
"[Date of Joining] >= #1/1/1992#" (OR "[Date of Joining] > #1/1/1992#")
```

and this is just like the condition you would code in SQL.

If you wanted to use the Forms Collection reference you would code:

```
strFilter = "[Date of Joining] >= forms!membership!txtJoinDate"
```

The value that the user enters should be validated in a similar fashion to that shown in section 3.3.1.

3.5.6 Removing a filter

A form has a property called `FilterOn`. This takes the value `True` if a filter is applied and `False` if not.

To remove filters, all you need to do is set this property to `False`. See Fig 3.5.5.

```
Private Sub cmdFilterClearAll_Click()
    'remove all filters

    FilterOn = False
    txtTotalMembers = DCount("[Membership No]", "Membership") 'or myShowMemberCount ""
End Sub
```

Fig 3.5.5 Removing a form filter

As the count is to show all the records, the third `DCount` parameter (which is optional) need not be specified on the `DCount` statement.

3.5.7 Combining Filters

So far, all the filters that have been shown here work individually. But what if the Centre wants to know, for example, how many smokers have joined in the past week?

This is very simple to do, as you just need to join each filter to any existing filter, using `AND` (just as in multiple conditions in SQL). If the `FilterOn` property is set to `True` then you combine the new filter condition with the existing condition(s) in the `Filter` property. Rather than putting this code with each filter code, it would be best to write a procedure to take the value in `strFilter` and add any existing filter conditions to it, and then call this procedure wherever it is needed. This is set as an exercise for you to do for yourself.

3.5.8 Filtering on Null values

If you try any of the above filters with a Null value in the field, then the results may not be what you would expect. Two things that could happen if you have not validated to prevent this are:

- For a text (string) field (such as Sporting Interests) the filter will show all records where the field is not Null.
- For a non-string field (such as the category type) the filter may cause a run-time error caused by a missing value in the WHERE condition.

If you wanted to allow the user to filter to find Null values, then you would need to code:

```
If IsNull(txtFilterInterest) Then
    'look for empty field
    strFilter = "[Sporting Interests] is null"
Else .... etc
```

3.6. Using Combo and List Boxes on Forms

This section discusses some useful properties of these form controls, and then demonstrates how to use VBA code to change contents of the controls at run-time and use them to update records.

3.6.1. Some useful Combo and List box properties

These two controls have several properties in common. Some useful properties and settings are discussed below (see Access and VBA Help systems for fuller details). As with most object property settings, these can normally be changed in design view (if you wanted to change the original wizard settings, for example) or by VBA code.

Fig 3.6.1 shows combo box properties for two controls from the Membership form shown in Fig 3.4.3.

- **ControlSource**
 - o If you created the combo/list box via the Toolbox wizard, you would be given the option to save a value (usually a primary key) for use later, and to store it in a field on the form. If you do the latter, the form field name is put in this property.
- **RowSourceType**
 - o This property works together with the **RowSource** property (see next bullet below).
 - o Value List means that the list of items is hard-coded within the control, separated by semi-colons.
 - The list can be altered at run-time by code, but then resets to the original value(s) the next time the form is loaded.
 - o Table/query means that the list of items is sourced via values in a table or query. The contents therefore vary according to the values in the table/query.
 - The underlying table may be updated; see section 6.3.
- **RowSource**
 - o If the **RowSourceType** property is Value List, then this property can be blank, or contain the specific values required to be shown.
 - o If the **RowSourceType** property is Table/Query, then this contains the SQL that selects the appropriate columns from the table/query.
- **ColumnCount**
 - o Specifies the number of columns to be shown in each row of values.
- **ColumnHeads**
 - o By default this is set to No. The following will happen if you set this to Yes:
 - If the **RowSourceType** property is Value List, then the first row of values in the list is taken to be the headings.
 - If the **RowSourceType** property is Table/Query, then the headings from the table/query fields will show in the combo/list box.
- **ColumnWidths**
 - o Lists the width of each column. If a column is not required to show (for example, a primary key value) the width is set to zero (0). In this way you can hide the column from view, but still refer to it in code.

- **BoundColumn**
 - o Specifies the column that is used to reference the row. If you code `myVariableName = cboBoxName` then the value that is put in myVariableName is the value in the bound column for the selected row.
- **Column** *(not available in the design view property box)*
 - o If you want to reference other columns apart from the BoundColumn, then you use this property. Confusingly, the column numbering starts from zero (0) not one (1), which is not consistent with the value set in the BoundColumn property.
 - o See sections 3.4.2, and 3.6.3 for examples of code.
- **ListRows** *(combo box only)*
 - o Specifies the number of rows that show in the drop-down list.
- **ListWidth**
 - o Specifies the full width of the list. If you add/remove columns in a combo/list box, and/or change the individual field widths, you will need to check that the value in here is still appropriate and change it if necessary.
- **LimitToList** *(combo box only)*
 - o No means that the list is purely a suggestion, and that the user can enter other values as well.
 - This is the default value.
 - o Yes means that the user must choose a value from the list. See also section 6.3.

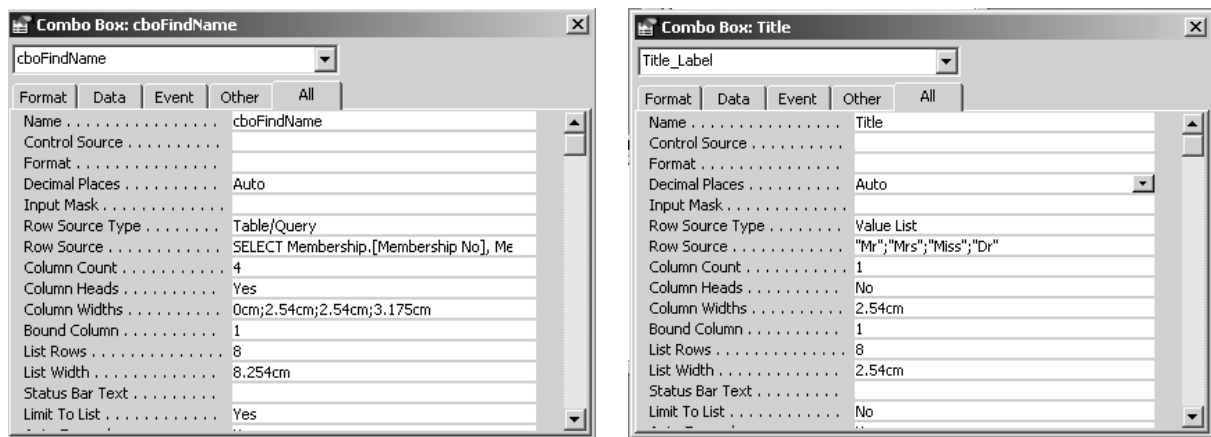


Fig 3.6.1 Combo Box properties for the Membership form 'Find Record' combo box and the Title field

3.6.2. Changing Combo Box contents at run time.

Suppose it is now required to record the Ethnic Origin and Ethnic Category of each member.

Create two new tables as shown here in Fig 3.6.2.

The CatType field on the EthnicOrigin table is a lookup field based on the EthnicCategory table, with the LimitToList property set to Yes.

Add two new fields, CatType and OriginType, to your Membership table. Use the Lookup wizard to base them on the relevant field of the new tables, and set the LimitToList properties to Yes. Note that the properties shown for the combo boxes for these two fields are as those listed in 3.6.1.

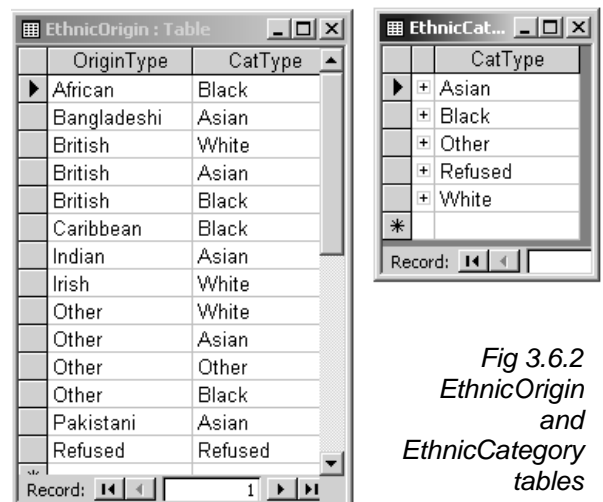


Fig 3.6.2 EthnicOrigin and EthnicCategory tables

Now add the two fields to your Membership form (in the example here a new form, called Membership & EthnicGrouping, being a copy of the Membership form, has been created). Amend the RowSource property of the OriginType field to reference the CatType field, as shown in Fig 3.6.3. If you use the Build (...) button you will see (and can alter) the query in the query design window.

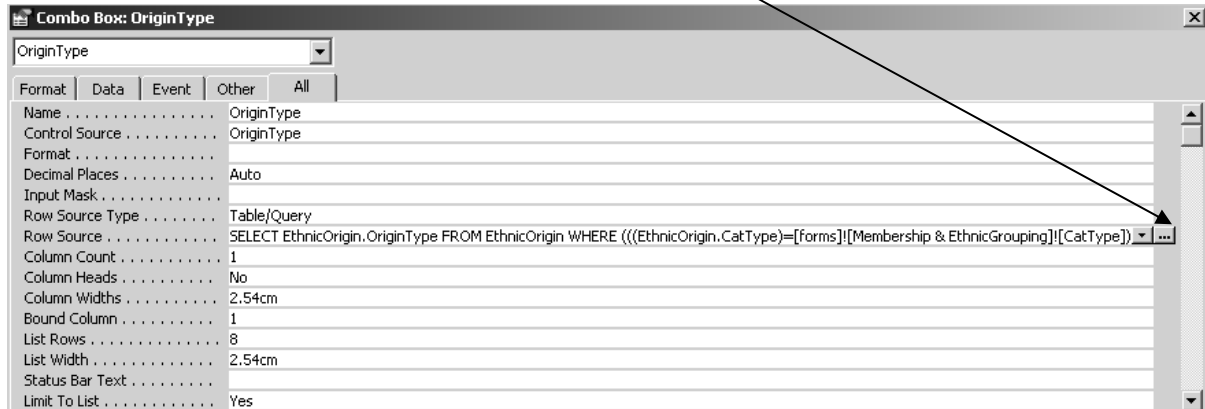


Fig 3.6.3 RowSource property for OriginType form field, with added WHERE clause.

The values in the drop-down list for the OriginType combo box will now be those appropriate to the value in the Ethnic Category. See Fig 3.6.4.

Fig 3.6.4 Membership table with new Ethnic Category and Origin Type fields

However, the list of values shown in the Ethnic Origin drop-down list is only queried by Access when the form is first opened; it will thus show a list appropriate to the value in the Ethnic Category field at that time. There are two events that need to have code in order to ensure that the list shows the correct values at all times:

- **Form_Current event.**
 - o The combo box values need to be updated for each new record, so code the following line in this event:
OriginType.Requery 'reset list for each record
- **CatType_AfterUpdate event.**
 - o If the value in the Ethnic Category field is changed, then the value in the Origin Type must also be changed. The code shown in Fig 3.6.5 will make the Origin Type list drop down automatically, and move the focus to it show that the user can see they must enter a new value.

- o If the Required property for the Ethnic Origin field is set to Yes in the table design, then the line in Fig 3.6.5 that clears the value in the field will fail at run-time. If you want this field to be required, then you will have to check for it when the record is saved, possibly in the Form_BeforeUpdate event.

```
Private Sub CatType_AfterUpdate()
'change Ethnic Origin list to match the value in here

OriginType.Requery      'run the query behind the Origin combo box
OriginType.SetFocus     'move the focus to that box
OriginType.DropDown     'show the origin drop-down list
OriginType = Null       'clear the previous value in the list

End Sub
```

Fig 3.6.5 code to ensure Ethnic Origin is changed if Ethnic Category is changed.

3.6.3 Using a list box to select records and change contents at run-time.

The example here uses list boxes to enable the user to register a class member on a class. The form that is used is as shown in Fig 3.6.6.

Fig 3.6.6 Class Registration form

3.6.3.1. Create a form based on the Class List table.

Using the form wizard, create a form based on the Class List table. This table has just two fields (Class No and Membership No) as it decomposes the m:m relationship between Class and Member. Add the textboxes txtLastname and txtActivity.

In the **Form_Load** event, code the following to open the form for a new registration:

```
DoCmd.GoToRecord , , acNewRec
```

3.6.3.2. Add two list boxes in the form footer.

Add two list boxes as shown in Fig 3.6.6.

- **IstMember**
 - o Base this on the Membership table.
 - o Select enough details to identify the member (Membership No, Lastname, Firstname, Street, Date of Birth) and the Sex. The last field will be used to filter appropriate classes in the Class list box.
 - o Choose to show the Membership No (as this could be a useful check of identity where there are two or members with the same name).
 - o Choose to store the Membership No in the field of that name on the form, bound to the Class list table. When the user clicks on the list box, the Membership No will be put in the form field automatically (see the ControlSource property).
 - o Change the field properties to hide the Sex column (set the ColumnWidth property for this column to zero and adjust the ListWidth property accordingly).
 - o Change the RowSource property to sort the data on ascending Lastname and Firstname. This is a much more useful order than the default order of Membership No.
- **IstClass**
 - o Base this on the Classes table.
 - o Select all fields and choose to hide the class number.
 - o Choose not to store the Class No in the field of that name on the form. This will be done by code later, in section 3.6.3.5, as we will check first to see if the member is already registered on the class.
 - o Set the Visible property to No.

3.6.3.3. Add a textbox called txtLetters near to IstMember.

The list box IstMember shows the full list of members in the Chelmer Leisure Centre. For the 'live' list, there could be several thousand members, which is far too many for the user to search through. The purpose of txtLetters is for the user to be able to narrow down the search by entering the start letters of the surname. Change the SQL/query for IstMember to add the following criterion for the Lastname field:

```
Like [forms]![Class List]![txtLetters] & ""
```

This will select just the members whose Lastname begins with the letters entered in txtLetters.

When the form is first opened, txtLetters is empty, so all members will be selected.

Now add a non-wizard command button called cmdResetMemberList.

Add the code shown in Fig 3.6.7 to the code module for the Class List form. Note the use of the Requery method to run the SQL for the list box, and adjust the rows shown accordingly.

```
Private Sub cmdResetMemberList_Click()
'set list back to show all members
txtLetters = Null
IstMember.Requery

End Sub

'-----
Private Sub txtLetters_AfterUpdate()
'set list to filter for the start characters entered in txtLetters
IstMember.Requery

End Sub
```

Fig 3.6.7 code to filter records in the member list box

3.6.3.4 Create DoubleClick event for IstMember.

Create a textbox called txtSex on the form. In Fig 1.6 this field is shown visible, just above IstClass. In practice, it would probably have its Visible property set to No, but it's useful to see it while testing.

Add a criterion for the Male/Female/Mixed column in the IstClass RowSource property SQL/query of:
[forms]![Class List]![txtSex] Or "Mixed"

This ensures that only classes appropriate to the gender of the selected member will be shown in IstClass.

Create a DoubleClick event for IstMember and add the code shown in Fig 3.6.8. Try double-clicking on the Member list box, see the Membership No, Lastname and Sex appear in the appropriate textboxes, and see classes appropriate to the member's sex in the Class list box.

```
Private Sub IstMember_DblClick(Cancel As Integer)

    If IstMember.Column(5) = True Then
        txtSex = "Male"           'this will be used by lastClass query
    Else
        txtSex = "Female"
    End If

    txtLastname = IstMember.Column(1) 'show member lastname on form

    IstClass.Requery           'requery IstClass and...
    IstClass.Visible = True    '...make it visible

End Sub
```

Fig 3.6.8 Code for IstMember DoubleClick event to set contents of IstClass

3.6.3.5. Check if member is already registered on the class.

There is just one more thing that needs to be done. It should not be possible to register a member on the same class twice. Create a DoubleClick event for IstClass, as shown in Fig 3.6.9.

```
Private Sub IstClass_DblClick(Cancel As Integer)
Dim strCriteria As String

    strCriteria = "[Class No] = " & Forms![Class List]!IstClass _
        & " AND [Membership No] = " & Forms![Class List]!IstMember
    If DCount("[Class No]", "[Class List]", strCriteria) > 0 Then
        MsgBox "Member already registered on this class"
        Undo 'clear form
    Else
        [Class No] = IstClass 'put class no in form field
    End If

End Sub
```

Fig 3.6.9 choosing and validating the class for the member

3.6.4 AddItem and RemoveItem methods

These new methods (not available in Access 97 or 2000) apply only to combo/list boxes where the RowSourceType property is 'Value List'. The RowSource property holds items separated by a semi-colon ";".

Access 97 had a **Clear** method to clear a list box. This is no longer available, but it is very simple to clear the list box by coding:

```
IstOutput.RowSource = ""           "" = the 'empty string'
```

3.6.4.1. Using AddItem

To add to a list box with only column: `IstOutput.AddItem Value1`
 To add where there are two columns: `IstOutput.AddItem Value1 & “;” & Value2` *and so on*

Where:

- The list box is called `IstOutput`
- `Value1`, `Value2` etc are variables that hold the new values for the list box row. Literals can also be used.

3.6.4.2. Using RemoveItem

The code below assumes that the list box is called `IstOutput`.

Delete the last row	Delete the selected row
<pre>Dim intRowNo As Integer intRowNo = IstOutput.ListCount - 1 If IstOutput.ColumnHeads = True Then intRowNo = intRowNo - 1 End If If Not intRowNo < 0 Then IstOutput.RemoveItem intRowNo End If</pre>	<pre>Dim intRowNo As Integer For intRowNo = 0 To IstOutput.ListCount - 1 If IstOutput.Selected(intRowNo) Then IstOutput.RemoveItem intRowNo End If Next</pre>

Fig 3.6.10 deleting a row from a list box

3.7 Exercises

3.7.1 Implement Receive Stock function, with validations

Implement the *Receive Stock* button discussed in section 3.2.2.3. When the user clicks on the command button to confirm the sale, the stock quantity for this item will be updated on the form and in the underlying table as the Number in Stock field is bound to the table.

Your tasks now are to build in checks to ensure that the user...

- ...enters a positive quantity sold for the sale (e.g. code to validate the amount entered and to check that it is greater than zero, with appropriate messages).
- ...only presses the command button once (e.g. set the button `Enabled` property to `False` when the form is opened, to `True` when a valid quantity sold is entered, then back to `False` after the sale is confirmed).
- ...as an optional extra, provide the user with a facility to cancel the receive stock action.

3.7.2 Show the total stock value on the stock form

Another of the Domain Aggregate Functions is `DSum` (see Appendix H.6). This works just like `DCount`, in that it works out a total from the rows found, but `DSum` adds up values rather than simply counting rows.

Use `DSum` to show a Total Stock Value on the Stock form of all stock in the Stock level table. Remember to adjust this total when records are added and deleted. If you put this code in the right place it will also cater for the situations when new stock is received and when the unit price changes. See section 3.4.1.2.

3.7.3 Function to calculate the number of years between any two given dates.

In section 3.2.3 you saw how to create and use a function to calculate an age given a date of birth. Using the ideas demonstrated there, write a new function to take in two dates and return the number of whole (integer) years between them.

Put the function in your Access module Calculations as a Public Function and give it a suitable name, such as MyCalcYears.

It would not be wise to assume that the calling code would always put the dates in the correct order, so start the function by comparing the two dates and putting them into two variables, one for the earlier date and one for the later date. Then compare the values in these two variables.

Use your new function to show the following on the Membership form, thinking carefully about which events to use for the code.:

- Number of years since the member joined.
 - o Create a textbox similar to that for the Member age (or, if you have changed your Membership form to be based on a query, you may prefer to use a calculated column in the query).
- Age at which the member joined.
 - o Similar to above.
- Message on the form to indicate whether or not the membership renewal is overdue.
 - o If the number of years between the renewal date and the system date is ≥ 1 , then the renewal is overdue.
 - o Have a label on the form with the relevant message in it, and set the Visible property to True or False as appropriate.
- Provide a filter of members whose membership renewal is overdue.
 - o Use your new function with the Date of Renewal and the system date as arguments, directly in the filter condition.

3.7.4 Filter on Sporting Interests

Add a filter to the Membership form to filter for specific text within the Sporting Interests field. Count the records. See section 3.5.1.

If the filter text box is Null, then set the filter condition to filter those records where the Sporting Interests field is Null. Count the records. See section 3.5.8.

3.7.5 Combine filters

Implement all the filters in section 3.5 (validating parameters where applicable) and then write and use the function discussed in section 3.5.6 to combine filters.

Change the forecolor property of the filter command buttons (or a similar property of the text or combo box, if you have decided not to use command buttons) to indicate which filters apply. Reset the colours when all filters are removed.

3.7.6 Filter by Sex

Add a filter to select (and count) all Male or all Female Membership records. If you have done exercise 3.7.5, then include this new filter in any code to combine filters and to set/unset colour properties.

3.7.7 Check for missing required fields

Rather than using Required = Yes in the table definition for required fields, code checks in the Form_BeforeUpdate event to put out message(s) when required fields have been left Null. See end of section 3.3.1.

3.7.8 Create myIsAlphabetic Function

Create and test the function suggested in section 3.5.1, then use it to validate string filter parameters.

Suggested logic and a test plan are shown in Fig 3.7.1. Test the function using the Immediate Window of the Debugger. Note that the maximum length of a string is huge, so it will not be practical to test for that (see VBA help).

```

Convert string to upper case (use UCase function - then only need to check for A-Z)
  (put in a separate variable so do not change input parameter)

Get length of string (use Len function)

Set return value of function to True (assume string will be OK)

If empty string (length = zero) then
  do nothing (function will return True)
Else
  For each character in the string (use a For loop with counter from 1 to length of string)
    Get the character (use Mid function with counter to point to the character in the string)
    If the character is <"A" or >"Z" then
      Set return value of function to False
      Make early exit from For loop
    End If
  Next (next character in string)
End If

```

Test No	Data	Reason for test	Expected result
1	Empty string	Can code cope correctly with empty string?	True
2	Single character A Z a z 1	Smallest non-empty string. Alphabetic, at boundaries of alphabet. As above, lower case. Not alphabetic.	True in each case True in each case False
3	The full alphabet ABCD...XYZ abcd...xyz	All 26 alpha characters	True True
4	! " £ (top row) < > , . ? / etc...	Other non-alpha characters on keyboard. Test each one singly.	False in each case.
5	A1 1A B2 1B	Mixed alpha and non-alpha. Valid char at each end of string, and non-valid char at each end of string.	False in each case
6	String with spaces in: This and that	Will a string with spaces be allowed?	<i>What do you think will happen here? Is the space character in the range A-Z? See if you can work out how to cater for spaces as well as alphas.</i>

Fig 3.7.1 Suggested logic and test plan for myIsAlphabetic function

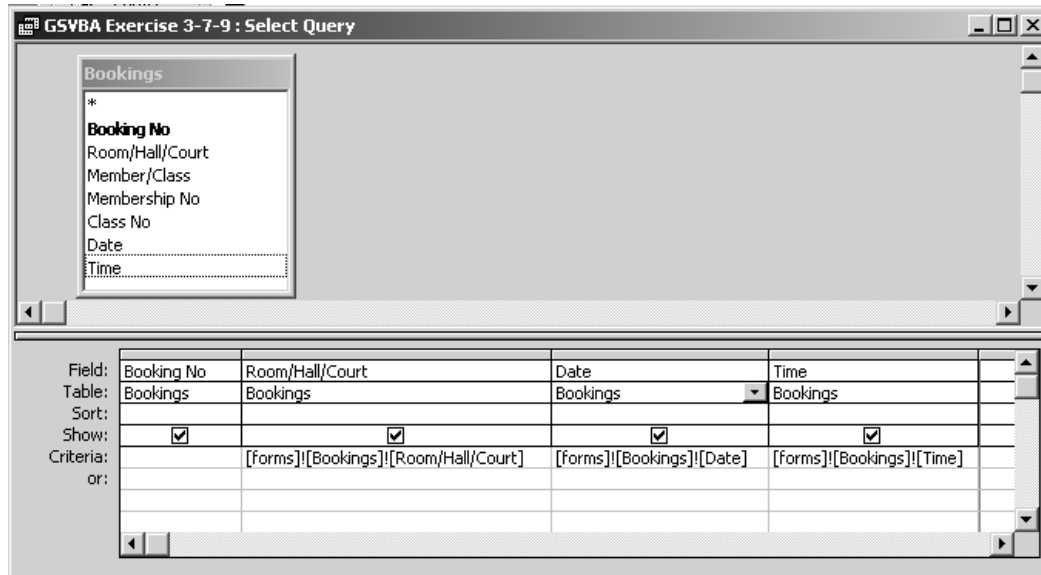
See Appendix H.2 for a list of string functions.

See Appendix F.3.3 for the format of FOR...NEXT loops.

3.7.9 Check for existing bookings (prevent double bookings)

The booking form shown in Unit 16 of McBride will allow double bookings. Using the DCount Or DLookup function, put code behind a Confirm Booking (create using Wizard Save) button on the form to check the Booking table to see if there is already a record for the same Room/Hall/Court on the same date and at the same time. If so, then display a suitable message and cancel the booking. If not, then display a suitable message and make the booking.

The query shown below may help you to understand what to do.



3.7.10 Show booking history information per member

It could be useful to show some booking history information on each member record, such as:

- Date of last booking (use DMax, as this will select the maximum date of booking).
- Total bookings member has made (use DSum).

As you want to show these values for each member record, put the code in the Form_Current event.

PART 4 – USING EVENT CODE ON FORMS – MENUS

REVIEW OF PART 4:

In this part of the Trainer you will see...

- ...how to create and use a non-switchboard Menu.
- ...how to show a dynamic date and time on the menu, using built-in functions.
- ...some useful form format properties.
- ...how to open a database with a specific form and how to suppress the appearance of the database window.
- ...commands to exit an application and Access, just exit an application, close a form.
- ...some information about Control Tips and Accelerator Keys.
- ...how to create and use a sub menu to open a form in different modes.
- ...how to use the form `InsideWidth` property to change the size of a form when loading.
- ...how to filter records via a menu.

See Appendix I for details about the Forms Collection.

See Appendix H for details about Access built-in Functions

4.1 Introduction

For a 'real' application, you would not normally expect the user to access forms, reports, queries, etc via the Access database window, but would provide menus. The user does not want, and does not need, to know the names of forms, tables and the like. The user simply has tasks that he/she wishes to perform and it is the developer's job to design and implement a system to assist with that process. You would normally design your menus as part of the full form and report design, prior to implementation. Here, for demonstration reasons only, things have been done piecemeal.

This Part of the Trainer will discuss how to create a non-Switchboard menu (it is much more flexible to create menus yourself), and some useful features.

The Membership form used so far has the buttons for *View*, *Add*, *Edit*, *Delete* etc on the form itself, and the user stayed with the same form in view for all maintenance operations. This form would normally be called from a Main Menu.

An alternative method used is for the Main Menu to call a Sub Menu which has the various operations on it. This Sub Menu may still use the same form (when all fields are to be on view) or may call forms with just a selection of fields available.

4.2 Creating and Using a Main Menu

4.2.1 Starting a menu – heading, dynamic date & time, day of week

Do the following:

- Open a new form in design view, without specifying a table or query. The form presented will not be very big, so you may want to resize it.
- Create a label called `lblHeading` on the form with a temporary caption such as 'Heading'. Select an appropriate font and size. In the `Form_Load` event write the code to put the value for the Chelmer Leisure Centre name in the label (see section 2.2.1). Use the `UCase` built-in function (see Appendix H.2) if you want the text to show as upper case.
- Save the form as Chelmer Leisure Menu.

It is good HCI to show today's date and time on a form. The day of week may also be useful. Do the following:

- Create two unbound text boxes on the menu form, delete both labels, and name the fields as txtDateTime and txtDayOfWeek.
- Create a Form_Timer event and insert the following line of code

```
txtDateTime = Now()
```

' put date and time on menu – uses Access Now built-in function
A timer event is activated automatically after a set interval, specified in the form TimerInterval property. In the Form_Load event, add the following line of code

```
TimerInterval = 1000
```

'1000 milliseconds = 1 second – to update time on menu
The time on the menu will now be updated every second. Look at the form in view mode to see this. You could also set this value directly in the form property box (event tab).
- To display the day of week code the following line in the Form_Load event (or in the Form_Timer event if your application runs over midnight!):

```
txtDayOfWeek = WeekdayName(Weekday(Date), , vbSunday)
```

' put today's day of week on menu
 - o Weekday is one of Access's built-in functions and returns the day of week number (Sunday = 1 to Saturday = 7) for the given date. Here the code is using the system date, obtained via the Date built-in function
 - o WeekdayName is another built-in function and uses the day number of the system date (here returned by the Weekday function) as one of the arguments. The built-in constant vbSunday is used to specify the start day of week (if you miss this off then the result is one day out).
 - o The WeekdayName function was not available in Access 97; you would have had to code this for yourself, using Weekday and checking each of the seven possible values that it should return.
 - o Look at VBA Help and Appendix H.1 to see details of the functions and how they work. The various built-in weekday constants are listed in Help as well. Use the Debugger Immediate Window to experiment and see what results the functions and constants will return.

The date, time and day of week will now display on the form and will be updated every second. Use the field properties to change the way these fields display (BackStyle, SpecialEffect, etc). Set the Locked property to Yes as these are not fields into which the user will be allowed to enter data. See Fig 4.2.2.

With Access 97, the timer interval interfered with the ApplyFilter Method of DoCmd, causing run-time error 2491; you had to turn the timer event off then turn it on back again afterwards to avoid the error. This problem appears to have been fixed for Access 2000/2002.

4.2.2 Improve the menu appearance.

Open the Menu form property box and change the settings as shown in Fig 4.2.1.

Property	Setting	Meaning
Caption	Main Menu	This replaces the default caption in the blue bar at the top of the form. If you wanted to set it to the value in myconChelmerName you would have to code this in the Form_Load event.
Scroll Bars	Neither	Removes the horizontal and vertical scroll bars.
Record Selectors	No	Removes the ► record indicator from the form .
Navigation buttons	No	Removes the Access buttons from the bottom of the form.
Dividing Lines	No	Prevents the lines that separate the various sections of the form being shown on the form.
Auto Center	Yes	Form will display in the centre of the screen when loaded.
Border Style	Dialog	Changes the edges of the form, so that it looks more like a dialog box. It also prevents the form from being resized.
Min Max Buttons	None	Removes the min/max buttons controls from the top right of the blue bar at the top of the form.
Close Button	Yes (for now) (see section 4.2.5)	Shows the Close button control at the top right of the blue bar at the top of the form. Set this property to No to disable (grey out) the close button).

Figure 4.2.1 Some Form Property Box Settings

4.2.3 Command button to load a form

Now create a command button to open your Membership form. There is a command button wizard for this; choose to open the form showing all records. Use suitable text and field names for the button, e.g. 'Membership Maintenance' and `cmdMembershipMtce`. If you click on this button, the Membership form is loaded. You now have a very simple menu. That is all a menu is, loading forms or reports etc from buttons.

Finally, add the following line to the menu `Form_Load` event `cmdMembershipMtce.SetFocus`. This will move the focus to this new command button, rather than the day or time field on the menu.

Your menu should now look something like that shown in Fig 4.2.2.

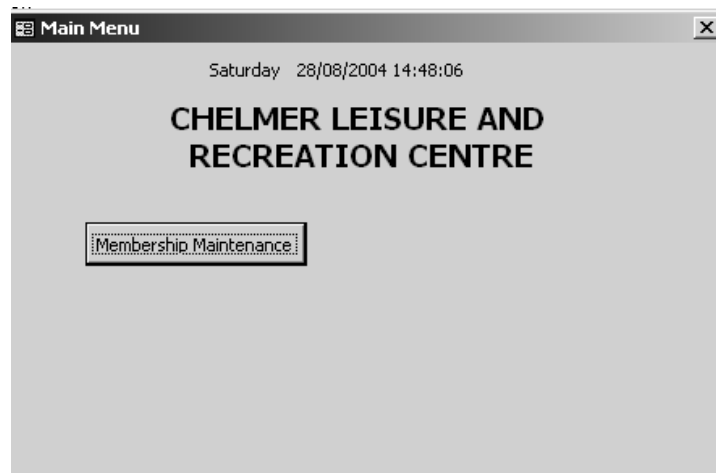


Fig 4.2.2 simple Main Menu showing heading, date, time and command button to open Membership form

4.2.4 Open menu automatically on start-up

From the Database Window, go via *Tools* → *Startup* and change the settings in the Startup dialog box as follows:

- Select the Main Menu from the Display Form box.
- De-select the option to display the database window

Now close and reopen your database. Your screen will now show just the menu, in the centre of the screen. (Simply re-select the database option window whilst developing your system, if wanted).

See section 7.6 for a fuller discussion of Startup options.

4.2.5 Exiting the application

The user can close the menu by clicking on the 'close' box on the top right-hand corner of the menu. However, this will not close the database or Access. For the final version of the system (as distributed to users) it would be better to have a command button for the user to use to exit; then this will be entirely under your control. So, do the following:

- Create a command button to quit the application (there is wizard for this). Give it the text 'Close Menu and Exit'.
- This button will close Access as well as the database, which can be rather a nuisance when testing and amending code, so you may like to comment out the line
`DoCmd.Quit`
 and code
`CloseCurrentDatabase 'closes application but not MS Access`
 or `DoCmd.Close 'closes form only`
- On the Format tab of the form property box, set the Close Button property to No.

4.2.6 Control Tips

Control Tips are a very simple, but very useful, feature to aid the user, especially for command buttons that have pictures rather than text. Do the following:

- Open the property box for the Membership Maintenance command button.
- Click on the Other tab and find the entry for ControlTip Text. Enter a description such as
Add, Edit, Delete and View Membership Details
- Now, when the user positions the cursor over the button, the description will appear as a pop-up box. See Fig 4.2.3.

You can also set the Control Tip Text via VBA:
cmdMembershipMtce.ControlTipText = "text"

Control Tips can also be used with other form objects; look at the property box.

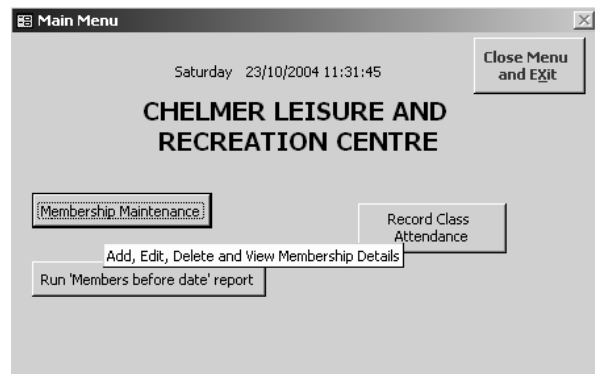


Fig 4.2.3 Control Tip Text for a command button

4.2.7 Accelerator Keys

So far, the mouse has been used to point to fields, click on buttons and the like. But what if the mouse became faulty? How would the user be able to continue, or at the very least, be able to close down the application? Accelerator keys are an alternative to using the mouse. Do the following:

- Open the property box for the Membership Maintenance command button.
- On the Format tab, change the Caption by adding an & before the M of Membership, so that the Caption now reads &Membership Maintenance
- The button will now show an underline character under the M of Membership, i.e. Membership Maintenance in both design and form view.
- If the user presses the M (or m) key, the effect will be the same as clicking on the button, i.e. the Membership form will be loaded.

The accelerator key does not have to be the first key of the caption. Try putting an & before the 'x' of Exit on the close form button, so that the Caption now reads Close Menu and E&Xit. See Fig 4.2.3.

Each accelerator key must be for a different letter, or the results may not be what you expect!

4.3 Data Maintenance via a Sub Menu

This section discusses the changes that need to be made to your current Membership form, and then puts all these ideas together to show how the maintenance functions now work via a sub menu.

4.3.1 Create a sub menu

To create a sub menu, do the following:

- Open a new form in design view, without specifying a table or query
- Set up suitable main and sub headings.
- Remove scroll bars, close button, etc (see Fig 4.2.1)
- Save it and call it Membership Maintenance Menu. See Fig 4.3.1 (but note that this has buttons that you have not created yet).

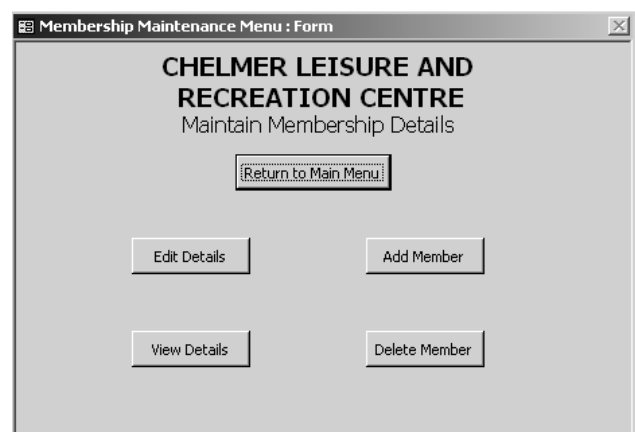


Fig 4.3.1 example sub menu

4.3.2 Exiting from the sub menu

Add a wizard button to return to the Main Menu:

- Using the wizard, create a button to close the Sub Menu
- Open the code module for your Main Menu, and change the line in the `cmdMembershipMtce_Click` event that says


```
stDocName = "Membership"           previously loaded Membership form
```

 to


```
stDocName = "Membership Maintenance Menu"   now loads sub menu
```
- Open your Main Menu in form view, click on the *Membership Maintenance* button, and your Sub Menu will open. Click on the *Return to Main Menu* button and the Sub Menu form will close and you are returned to your Main Menu.

4.3.3 Amend Membership form

The Membership form that you have at present has got *Add* etc buttons on it already. These will not all be necessary if it is to be called via the Sub Menu. The following tasks are fiddly, but are good practice as this is the sort of thing you will need to do if you want to re-use a form (or a report) and adapt it for a new situation (and will also check that you have understood what you have been doing so far). Do the following:

- Create a new version of your Membership form. Call it SubMembership.
- Open your SubMembership form in design view, and delete the command buttons for *View*, *Edit* and *Add*. Leave the *Delete* button where it is but set the `Visible` property to `False`.
- Open the form in form view. You will get a compilation error message as your code will be trying to change the `forecolor` for a non-existent command button. Edit the module code to remove all references to, and event code for, the deleted buttons for *View*, *Edit* and *Add*. Leave the main code for *Delete* (you will use this later) but remove the changes to the `forecolor`.
(**Tip:** use *Edit*→*Find*)
- Once you have done this you will get a run-time error in the `Form_Current` event `DLookup` statement trying to reference the form "Membership" (see section 3.2.4). The form is now called "SubMembership" so you will need to change the reference in all uses of the forms collection.
(**Tip:** use *Edit*→*Replace* – but be careful not to change references to the Membership table!).
- The `forecolor` on the three remaining buttons is white. You may wish to edit your code to leave this as black, e.g. remove procedures `mySetToViewMode` and `myResetButtonsToOff`, as the button colour (or format, if you have used label-buttons etc) is not relevant now.
 - o A simpler course of action could be to remove all code within `myResetButtonsToOff` and leave the rest unchanged.
- Remove all code that amends the form `AllowEdits` property, with the exception of the 'Find' combo box `GotFocus` and `LostFocus` events and the coding for the various filters from sections 3.5 and 3.6.
- Try opening your form. It should now show just the *Save*, *Next*, *Previous* and *Close Form* buttons. See Fig 4.3.2.
 - o If you have done the exercises in section 2.3.6 to provide facilities to change address and renew membership, then these buttons will also show. Leave them as they are.
- The following sections will show how to replace the deleted functions with operations from the Sub Menu.

4.3.4 Editing Membership records

- Using the wizard, create a new command button on your Sub Menu, to open your new SubMembership form showing all the records. Give it a suitable caption and name (e.g. *Edit Details* and `cmdEdit`). See Fig 4.3.1.
- Change the generated code for the `OpenForm` command from


```
DoCmd.OpenForm stDocName, , , stLinkCriteria
```

 to


```
DoCmd.OpenForm stDocName, , , stLinkCriteria, acFormEdit
```
- Use the VBA Help system to look at the details of the `OpenForm` method. The extra parameter added to the command is specifying the `datamode` in which the form is to be opened.

- Open your Sub Menu in form view, click on the *Edit Details* button, and you will see the first record in your file. You can amend this record and save it. You can also use the *Next/Previous* buttons, or your 'find' combo box to locate records. The filters should also still work.

The screenshot shows a Microsoft Access form titled "Membership" for the "Chelmer Leisure and Recreation Centre". At the top, there is a "Find Name" dropdown menu with "Walker" selected. Below this, the form is divided into several sections:

- Membership Details:** Shows "21 members" and a "Clear all filters" button. A "Close Form" button is in the top right.
- Membership No:** A text box containing the number "1".
- Title:** A dropdown menu with "Mr" selected.
- Firstname:** A text box containing "Andrew J".
- Lastname:** A text box containing "Walker".
- Street:** A text box containing "16 Dovecot Close".
- Town:** A text box containing "Chelmer".
- County:** A text box containing "Cheshire".
- Post Code:** A text box containing "CH2 6TR".
- Telephone No:** A text box containing "01777 569236".
- Occupation:** A text box containing "Builder".
- Date of Birth:** A date picker showing "12/03/1952" and an "Age" field showing "52".
- Category No:** A text box containing "2" and a dropdown menu with "Senior Club" selected.
- Sex:** Radio buttons for "Male" (selected) and "Female".
- Smoker:** A checked checkbox.
- Date of Joining:** A date picker showing "03/02/1992".
- Date of Renewal:** A date picker showing "27/08/2004".
- Sporting Interests:** A text box containing "Tennis, Squash".
- Filtering and Navigation:**
 - Buttons for "Save", "Renew Membership", "Change Address", "Previous Record", and "Next Record".
 - Filter sections: "Filter By Name" (with a text box for "Enter first character(s) of last name"), "Filter after join date" (with a date picker), "Filter By Category" (with a dropdown for "Select Category Type"), "Filter Sporting Interest" (with a text box for "Enter sporting interest"), "Filter Smokers" (with a "Tick for smoker" checkbox), and "Filter by Town" (with a dropdown for "Select Town").
 - A "Filter overdue renewals" button.

Fig 4.3.2 Possible Membership form prepared ready for use from Sub Menu

4.3.5 Viewing Membership records

- In exactly the same way as above, create a new button to open your new SubMembership form. Give it a suitable caption and name such as *View Details* and `cmdView`. See Fig 4.3.1.
- Change the generated code for the `OpenForm` command to
`DoCmd.OpenForm stDocName, , , stLinkCriteria, acFormReadOnly`
- If you click on this button, your form is opened, but you cannot edit any data field, as the `AllowEdits` property has been set to `False`; that's what the built-in constant `acFormReadOnly` does.
 - o However you can still use the 'find' combo boxes and the filters, as your coding already adjusts the `AllowEdits` property when the cursor is positioned on these controls.

4.3.6 Adding new Membership records

- As before, create a new button to open your new SubMembership form. Give it a caption and name such as *Add Member* and `cmdAdd`. See Fig 4.3.1.
- Change the generated code for the `OpenForm` command to
`DoCmd.OpenForm stDocName, , , stLinkCriteria, acFormAdd`
- If you click on this button, you are presented with a 'blank' form and can enter new details. The *Save* button will save the form. Clicking on the *Next* button will move to another blank form, so that you can enter more than one record if wanted. Clicking on *Previous* will show you the records that you have added so far this session; it will not show other records in the database.
- Clicking on the 'Find' combo box will have no effect, as this action is not appropriate to adding a new record. However, the Filter buttons will still work, which may or may not be what you want to happen.

4.3.7 Deleting existing Membership records

- If you look at the allowable `datamodes` on the `OpenForm` command, you will see that there isn't one for `Delete`, as this is assumed to be part of `Edit` (i.e. changing the dataset includes removing data as well as merely amending it).
- Two possible ways of allowing for deletions are:
 - o Allow the user to delete records via the *Edit* button.
...Or...
 - o Create a *Delete* button on the sub menu. Have a hidden field on the sub menu, and put 'Delete' in here. In the SubMembership `Form_Load` event, if the form has been opened from the *Delete* button, set the button property `Visible = True`.

4.3.8 Try this: (See Figs 4.3.1 and 4.3.3)

In this section you will try out some of the ideas discussed in sections 4.3.1 to 4.3.7.

- Create a hidden unbound text box (*Visible* property set to *No*) on the Sub Menu, delete the label. Give the text box a suitable name (e.g. `txtAction`). For each of the maintenance buttons on the Sub Menu, add appropriate text to this field, e.g. for Add button, put "Add":


```
txtAction = "Add" 'put sub menu choice in hidden field on this form
```

 - o Use of 'hidden' fields on a form is a standard technique to pass information between forms.
- Create a text box on the new SubMembership form, in the heading. Give it a suitable name (e.g. `txtAction`) and suitable display properties. Delete the label. In the `Form_Load` event code:


```
txtAction = Forms![Membership Maintenance Menu]![txtAction] 'copy choice from hidden field on sub menu
```

 - o Even though these text boxes have the same name, Access can work out which is which. If you prefer, you can use different names.
 - o This will show the action in the form header as information (and a reminder) to the user. You may like to change some of the appearance properties for this box, for example, try setting the `BackStyle` to `Transparent` and the `SpecialEffect` to `flat`. Set the `Locked` property to `Yes`. Change the font to suit. Try choosing an option from your sub menu and see the effect in the SubMembership form header.
- Using the property boxes, set the *Visible* properties for the *Delete* and *Save* buttons on the SubMembership form to `False`.
- If you have implemented the Change Address and Renew Membership buttons from Exercise 2.7.1 then set the *Visible* properties for these to `False`.
- In the `Form_Load` event for the SubMembership form, enter code (possibly using a CASE statement) to do the following (test each bit out as you do it – this will help with debugging if you make a mistake):
 - o If the action is Edit:
 - Make the *Save*, *Renew Membership* and *Change Address* buttons visible.
 - o If the action is Add:
 - Make the *Save* button visible.
 - Disable/remove any 'Find' combo boxes (set the `Enabled` or `Visible` property to `False`).
 - Make the number of records invisible.
 - The various filters to the right-hand side of the form are not relevant when adding records, but it is possible to resize the form to hide these buttons. Look up the `InsideWidth` property in VBA Help. If you code `MsgBox InsideWidth` in the `Form_Load` event, you will see the value that applies to the full-sized form. So, for the Add action, code something like


```
InsideWidth = 11500
```

 (you will need to experiment to find an appropriate value for your form, and you may need to move some controls around on the form that the relevant ones can be seen).
 - o If the action is Delete:
 - Make the *Delete* button visible
- In the `Form_Current` event, code the following:


```
If txtAction = "View" Or txtAction = "Delete" Then
    Me.AllowEdits = False 'to set to view mode"
End If
```
- Now try all the buttons on the sub menu and see how it all works together. See Fig 4.4.3.

Fig 4.3.3 SubMembership form in Edit and Add modes.

4.3.9 Opening the form with a particular record

In the above sections, the form is opened giving the user access to the full set of membership records. This may not be what is required; you may wish to open the form only for records with a particular surname or membership number (the latter for example if you are simulating swiping a membership card – you cannot expect members to remember their numbers).

In section 4.3.4 you were instructed to open the form with the 'show all the records' option. If you had opted for the 'find specific data' option Access would have refused to do this as there are no linkable fields between the forms; the Sub Menu did not contain any appropriate text boxes to use as a link. If you look at the code for an open form button, you will see that there is a variable named `stLinkCriteria`, but this is not used. In order to open a form for, say, a particular surname, do the following:

- Create an unbound textbox on the Sub Menu form, call it `txtName` and set the label text to 'Enter last name of required member'.
- Add the following line before the `DoCmd.OpenForm` statement of the `View` button:


```
stLinkCriteria = "[Lastname]=" & [txtName] & ""
```

 - o This is the code created by the wizard if you choose the 'find specific data' option. (Note that "" after `[Lastname]` is ' ', and "" at the end of the line is " " – see section 3.5.1)
- If the name entered in `txtName` is Jones, the actual string that will be put in `stLinkCriteria` will be `[Lastname]='Jones'` and this will be used as a filter to open the form only with records for all members with the last name of Jones.
- If you want to be able to filter by all names with the same starting letters, change the code to


```
stLinkCriteria = "[Lastname] like " & [txtName] & ""
```

 (note "" at the end of the line is " * ")

With the letter J, the criteria would now be

```
[Lastname] like 'J*'
```

If the user does not enter anything in the txtName field, the form will open with all records (like '*') as before. This may be more useful than looking for an exact match. The user will be able to use the *Next* and *Previous* buttons to scroll through the filtered records, just as normal.

You may have noticed that the SubMembership form opens showing the full number of Membership records, regardless of how many are actually filtered via the Sub Menu. You will need to alter the line for this in the Form_Load event to

```
txtTotalMembers = DCount("[Membership No]", "Membership", _ ← note continuation over two lines
    "[Lastname] like" & Forms![Membership Maintenance Menu]![txtName] & "*"")
```

In order to use this process for the *Edit* and *Delete* buttons as well, copy the stLinkCriteria line to the Sub Menu Click events for these buttons. Better still, have a function such as that shown in Fig 4.4.4 and add the line

```
stLinkCriteria = mySetLinkCriteria
```

to each event. Then, if you wanted to change the link criteria in future, you only need to alter it in the one place, in the function. (See Exercise 4.4.4).

```
Private Function mySetLinkCriteria() as String
    mySetLinkCriteria = "[Lastname] like '" & Me![txtName] & "*"
End Function
```

Fig 4.4.4 Function to set the filter link criteria for opening the SubMembership form

You will not need any link criteria for the *Add* button. It may be useful to set the count of members to 0 (zero) initially and add 1 (one) each time a record is saved (in which case put the Visible property for the count of total members back to True).

Fig 4.4.5 Sub Menu with textbox for Lastname filter

4.4 Exercises

4.4.1 'Are You Sure?' procedure on Exit

If the user clicks on the Exit button for the Main Menu, ask an 'Are You Sure Question' with a Yes and No reply, and take appropriate actions depending upon the user response.

4.4.2 Sub Menu buttons for Renew Membership and Change Address

Instead of having the buttons for these processes on the SubMembership form, put them on the Sub Menu. A suggestion is that you open the SubMembership form for the chosen member(s) and make just the relevant fields available by setting appropriate properties on the other fields. Set the form width to a value that will not show the filters, as these will not be relevant.

4.4.3 Show count of records added

As suggested at the end of section 4.3.9, show a count of records added in each *Add* action. It is possible to add several records at a time, so it could be useful to show a count.

4.4.4 Open SubMembership form for a particular Membership Number

Using the ideas discussed in section 4.3.9, provide a facility for the user to enter a Membership No to open just the record for that member. It might be useful to use `DCount` or `DLookup` to check that there is a record for that Membership No first and open the form if the number is found, but display a suitable message if it is not found.

The form now has two textboxes in which the user can enter a value, so you will need to check which one has been used and set the link criteria accordingly.

Membership No	Lastname	Action
Null	Null	Use Lastname criteria. This will load all records.
Null	Not Null	Use Lastname criteria. This will load all records where the first character(s) match.
Not Null	Null	Use Membership No Criteria. If the record exists, this will load just the one record.
Not Null	Not Null	Error – give message to tell user to enter a value in only one of the textboxes.

The best place to put this code would be in the function `mySetLinkCriteria` (Fig 4.4.4).

4.4.5 Provide data maintenance facilities for the Stock Level table via a sub menu.

Using your Stock form from previous exercises, create a SubStock form and a Stock Maintenance sub menu for the various data maintenance facilities. Open the sub menu from the Main menu.

Move the Receive Stock facility to the sub menu.

4.4.6 Using a System Heading in a table.

Instead of coding the system heading in a `Public` constant, as in Fig 2.2.3, it could be more flexible to have a table with one row which has the system name. The user can set and change their own title using a form. Change your code as follows:

- Create a new table (`SystemHeading`).
- Create a form to show only one record for the user to enter/change the heading.
- Change `myconChelmerName` to be a public `String` variable (not a constant).
- In your Main Menu and in your new form, use the `DLookup` function to get the heading from the `SystemHeading` table and put it in `myconChelmerName`. The rest of your coding should now pick up the system heading as before.
- Change the heading via your new form, and see the new heading in the forms and reports.

PART 5 – USING EVENT CODE ON REPORTS

REVIEW OF PART 5:

In this part of the Trainer you will see...

- ...that reports have a code module and events.
- ...how to change font settings at run-time.
- ...how to calculate report totals.
- ...how to cater for 'empty' reports.
- ...how to use parameters to control the data on a report.
- ...how to use a calendar control for a date parameter.
- ...how to suppress items on a report.
- ...how to change the sort order at run-time.

See Appendix A for a summary of report events.

See Appendix I for details about the Forms Collection.

5.1 Introduction

So far we have concentrated on forms, but reports are every bit as important. Forms are often regarded by students as the most important aspect of a system (and they may be the most 'fun' bit to develop, if students enjoy interacting with their own code), and reports can be regarded as a bit of an added extra and not that important.

However, management reports are vital to the running of a business. Management may not interact day-to-day with the system (do you think that the senior Managers of the Chelmer Leisure Centre will type in details to record a new member? Or make a booking?), but they will want to receive both regular and one-off reports that tell them how the business is running and enable them to take appropriate business decisions.

Members of the public also interact with systems via reports (e.g. payslips, bank statements, insurance renewals), and students will receive computer generated fee payment demands, end-of-year result transcripts, etc.

Reports are a vital function of most systems and it is very important that they are well-designed.

You have already seen that forms have their own code module and events. You will now see that each report also has a module and events, and that these are accessed, and work, in exactly the same ways as the code modules for forms.

5.2 Report of members who joined 10 or more years ago

Membership lists can get very lengthy over time, especially if lapsed members are left on the list. Suppose that the management of Chelmer Leisure wants a report that lists all members who joined 10 or more years ago, highlighting those members whose annual membership renewal is overdue.

First create a query as in Fig 5.2.1, using (at least):

- Membership No, Title, Firstname, Lastname
- Date of last renewal
- Number of years since joining.
 - o Note how this uses the `myCalcYears` function from exercise 3.7.3 with the Date of Joining and the system date. If you have not created that function, then use the `myCalculateAge` function instead with just the Date of Joining.

You may need to amend your membership data to get a good spread of joining and renewal dates. Don't forget boundary conditions.

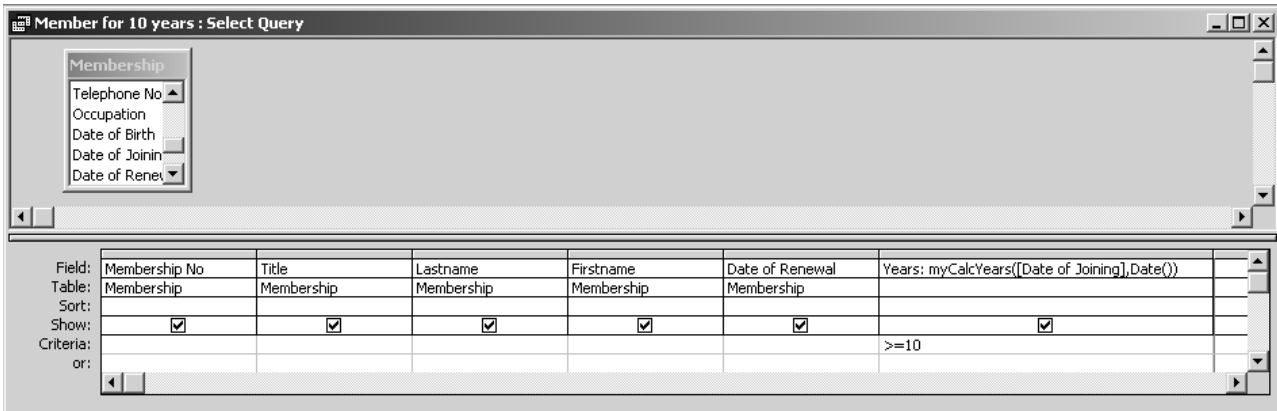


Fig 5.2.1 Query to select members who joined at least 10 years ago

Now create a tabulated report, based upon the query, to list the members in descending years order; See Fig 5.2.2.

- The heading for this report is a label with just some temporary text in it (such as 'heading'). It has been given the name lblHeading. The code:
`lblHeading.Caption = myconChelmerName`
 has been coded in the Report_Open event to put the text from the public constant in the report heading. Compare this with the similar action for putting the heading on a form, in section 2.2.1.
- Some joining and renewal dates of data have been changed for this report.



Fig 5.2.2 Basic report of members who joined 10 or more years ago.

5.3 Changing the appearance of a field at run-time

Management wants the report to highlight members who have not renewed their membership, i.e. where the date of Renewal is more than a year ago. Do the following:

- Open the report in design view and create event code for the Detail_Print event.
- Enter the code shown in figure 5.3.1 in the Detail_Print event. This code uses the myCalcYears function to check for renewal dates over 1 year ago (i.e. 2 or more years ago).
- Run the report. The lapsed members have their date of renewal highlighted in bold, red Italics. See Fig 5.4.1.
 - o You have seen in section 2.4.1 how to use the colour constants and/or the RGB function to change font on forms; the method is exactly the same for reports.
 - o Note that you must code for both lapsed and non-lapsed conditions, as the formats you set will apply for all later records unless you change them. Try removing the ELSE code and see the effect.

```
Private Sub Detail_Print(Cancel As Integer, PrintCount As Integer)
'highlight members whose annual membership has lapsed
If myCalcYears([date of renewal], Date()) > 1 Then
  [date of renewal].FontBold = True   'highlight if lapsed
  [date of renewal].FontItalic = True
  [date of renewal].ForeColor = vbRed
Else
  [date of renewal].FontBold = False  ' normal if not lapsed
  [date of renewal].FontItalic = False
  [date of renewal].ForeColor = vbBlack
End If
End Sub
```

Test no	Data	Reason for test	Expected result
1	No lapsed members in member table	No lapsed members.	All in normal black font.
2	All members have lapsed	All lapsed members.	All in bold, italic red font.
3	Mixture of lapsed and non-lapsed members, alternating through the report.	Mixture, to test fonts correctly reset.	Mixture of fonts*
4	As test 3, but first and last members listed now have different status.	As test 3, to show that first/last values do not affect the result.	As above.

* work out in advance which records you expect to show as lapsed.

Fig 5.3.1 Code to change the font at run-time, and suggested test plan.

Testing a report is very different to testing objects on a form, as a whole set of data values are normally processed at once, though it can sometimes be useful to have several versions of a table, each with one row, just to test one particular condition. If you have a table with several rows, it is useful to list the table in with your test plan and log, highlighting the tests for each row (in this example, highlighting those members whose membership has lapsed) so that you know in advance what your expected result is.

Note that it is also possible to use Conditional Formatting for this task, but the number of options is limited. Code is more flexible.

5.4 Calculating and printing totals

It is always good practice to print totals in a report footer (and, indeed, in group footers). Suitable totals here would be of all members and lapsed members shown on the report. Do the following:

- Open the report in design view and add two text boxes to the report footer (extend the area if necessary). Change the label captions to 'Total members lapsed' and 'Total members printed' and the textbox names to txtTotalLapsed and txtTotalPrinted.
- Create code for the ReportHeader_Print event. Add lines to initialise the totals to zero, e.g. `txtTotalPrinted = 0`. This will put the values in the textboxes to zero initially.
 - o The logical place to put this code may seem to be in the Report_Open event, but you get the error 'You cannot assign a value to this object' if you do this. Access does not allow values to be put in textboxes in this event.
- Go to the print event for the detail lines (Fig 5.3.1) and add code to increment the counts by 1 in appropriate places. For example, add `txtTotalLapsed = txtTotalLapsed + 1` to the code inside the IF statement where the font is set to bold, red Italics and code the equivalent for the total printed outside the IF statement. Your report should now look similar to that shown in Figure 5.4.1.

Not all report counts have to be coded; Access Help has very clear instructions on the standard counting facilities that exist for creating report and group totals. See Access Help with the keywords *report*; total, also McBride Unit 24. You will often need to count a selection of items in a report (e.g. the total of lapsed members above), and it is often best to code these totals; it may not always be possible to do them via the standard Access facilities.

The screenshot shows a report window titled "Member for 10 years". The report content is as follows:

Chelmer Leisure and Recreation Centre

List of Members who joined 10 or more years ago

Membership No	Title	Lastname	Firstname	Date of Renewal	Years since joining
20	Mr	Jones	Edward R	17/05/1996	13
2	Mrs	Cartwright	Denise	16/07/2004	13
19	Miss	Locker	Alison	13/06/1997	12
18	Mr	Locker	Liam	13/06/1997	12
16	Mrs	Robinson	Rebecca	06/12/1996	12
12	Ms	Young	Aileen	09/08/1996	12
8	Mr	Shanqali	Imran	13/09/2003	12
5	Miss	Jameson	Donna	11/09/2003	12
4	Miss	Forsythe	Ann M	12/09/2003	12
1	Mr	Walker	Andrew J	27/08/2004	12
17	Mr	Everett	Alan	05/11/1996	10
7	Mr	Harris	David J	01/02/1997	10

Total members printed: 12 Total members lapsed: 7

12 September 2004 Page 1 of 1

Fig 5.4.1 Report showing highlighted dates, plus counts

Run the tests from Fig 5.3.1 again, checking that you have the correct highlighting as before, and that you now have the correct totals for each test.

5.5 Empty Reports

What if there was no data to be printed? You can see what happens by changing the data in your Membership table so that all the joining dates are within 10 years, or (probably easier!) changing your query on which the report is based to select members who joined 20 years ago (or whatever figure will ensure that the query does not return any results).

Now if you run your report there will be a run-time failure 'You entered an expression that has no value' as your code is trying to check dates on non-existent data when calling the function `myCalcYears`.

This report only fails at this point because the code checks non-existent data. If this check were not there, the report would run but the results may not always be that clear.

Obviously you want to avoid a run-time failure and you want to make it clear on the report that there is no data to be listed.

5.5.1 Using the Report_NoData event and cancelling the report.

Create code for the `Report_NoData` event and enter the following code:

```
myDisplayWarningMessage ("There are no members to print")
Cancel = True
```

Now run the report. The message appears and the report is cancelled.

(To test that the report still works correctly with at least one record, amend your query or data accordingly and test again. You should get your report with the record(s) shown).

5.5.2 Printing an 'empty' report

In some cases, merely reporting via a message that there is nothing to print may be sufficient, but in many cases (for example regular reports) it is essential to print an 'empty' or 'null return' report. If a regular report is missing how is the user to know whether it was because there was nothing to print or whether the report had been mislaid?

Do the following:

- Declare a global variable in the report module called `bNoData`, of type `Boolean`. This will be used as a flag, to be set to `True` if there is no data for the report to print.
- In the `Report_NoData` event, delete the line to cancel the report and add the line:
`bNoData = True`
- In the `Detail_Print` event, top and tail your code with:

```
If bNoData Then
    'do nothing – the Detail_Print event appears to be invoked even though there is nothing to print!
    ' so we must prevent the code here attempting to process nothing and then failing at run time.
Else
:   existing Detail_Print event code goes in here
End If
```

Your report will now print showing totals of zero.

However, the `Report_NoData` event now appears to be called twice (I don't know why), which means that the message will be shown twice. You could therefore move the message to the `Detail_Print` event, which is only called once, or simply, do not show the message – that is probably easiest.

It might be better if your report stated explicitly that there were no records to print. Add another text box to your report footer. Delete the label. Call the box `txtNoDataMessage`. Use the property box for this new field to set the field to invisible initially. Add the following code to the `Report_NoData` event:

```
txtNoDataMessage = ("There are no records that meet the criteria")
txtNoDataMessage.Visible = True
txtTotalPrinted.Visible = False
txtTotalLapsed.Visible = False
```

Now, if you run the report with data, the report will look the same as before. If you run it with no data, the totals will be replaced by the message "There are no records that meet the criteria".

5.6 Using a query criteria parameter at run time

A more useful version of the report could be to allow the user to enter a date, rather than assuming a date 10 years before the run date. Do the following:

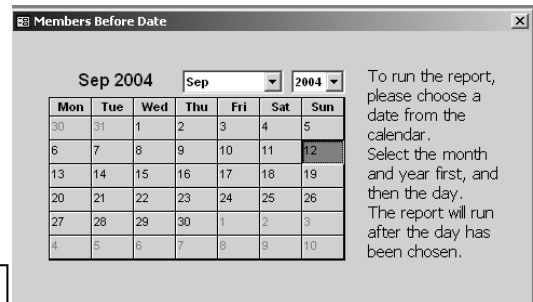
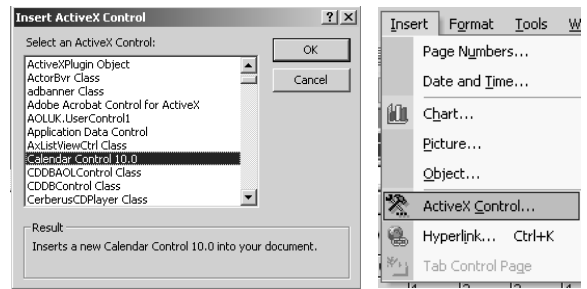
- Create a new form, not based on a table or query.
- You could use a textbox for the date, but here we will use an alternative version, a calendar control.
 - Using the *Insert* menu, add a calendar control to your form and move it to a suitable position.
 - See Fig 5.5.1.
 - Give the control the name `ocxCalendar`.
 - If you look at the calendar Value property, you will see that it is set to the date that you created the control, and this date is selected when the form is opened. If you want it to change to show a different date when the form is opened, then you will code this in the `Form_Load` event:


```
ocxCalendar.Value = Date() 'today
```

 or

```
ocxCalendar.Value = #1/1/2004# 'specific date
```

 If you want to remove this value, you can do it via the property box or code `ocxCalendar.Value = Null`



The `BorderStyle` property here is set to `Dialog`. Note that there are no max/min buttons and that the form edge is flat. You may also find the `Popup` and `Modal` form properties of use.

Fig 5.5.1 Creating a Calendar Control

- Copy your 'members for 10 years' query and create a new query, which compares the Date of Joining with the date in `ocxCalendar`, using a Forms Collection reference (see Appendix I), as in Fig 5.5.2. This query uses a copy of the Membership table, as the joining dates have been changed to test the report.

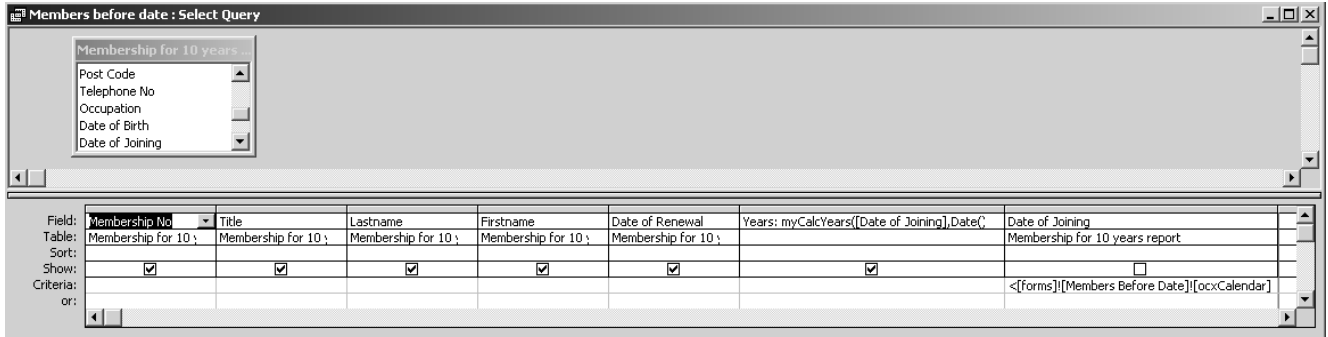


Fig 5.5.2 Query changed to reference date control on form.

- Make a copy of your 'member for 10 years' report, and change the `RecordSource` property to use the new query. Now, if you select a date on your new form and then run the report, you should see how it all works together.
 - But the report heading needs changing. To do this, give the name `lblSubHeading` to the subheading on the report, change the caption text to "List of members who joined before ", and add the following line to the `Report_Open` event:


```
lblSubHeading.Caption = lblSubHeading.Caption & Forms![Members before date]!ocxCalendar.Value
```
 - If you wanted the date to show in a different format, you can use the `FormatDateTime` function:


```
= lblSubHeading.Caption & FormatDateTime(Forms![Members before date]!ocxCalendar.Value, vbLongDate)
```
- The calendar `AfterUpdate` event is invoked when the day number on the calendar is chosen. This could be a suitable place (other than using the usual command button) to open the report. Create this event and add the following line of code:


```
DoCmd.OpenReport "Members before date", acViewPreview
```

 - Alternatively, use a command button to open the report.
- If you wanted to close the parameter form when the report was opened, then simply code the following after the line to open the report:


```
DoCmd.Close acForm, "Members before date"
```
- You could now add a button to your main menu for the report, which opens the parameter form.

5.7 Changing the sort order at run-time

The user may wish to see the same report in several different orders. Usually, we create a report with a built-in order, either by specifying an order for the underlying query (though this doesn't always guarantee that the report order is the same!) or by using Sorting and Grouping in the report design.

But it is also possible to set the relevant properties in the Report_Open event. Figure 5.7.1 shows the report property box and the values set for the OrderBy and OrderByOn properties.

Look these up in Access or VBA help (the same information appears to be in both).

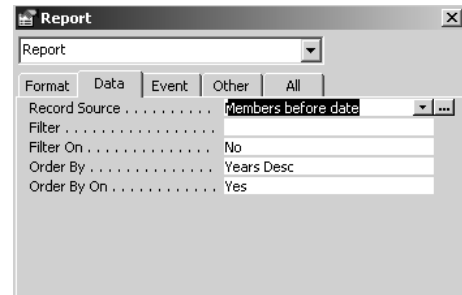


Fig 5.7.1 Report properties showing sort order at run-time

If you have used Sorting and Grouping to specify a default report order, or have specified the order when creating the report via the Report Wizard, then this order will over-ride whatever you will code. You will need to remove this sort order first (highlight the row in the Sorting and Grouping dialog box, and press Delete).

Now create two combo boxes on your parameter form, for the user to specify the fields for the sort order, and whether the order is to be ascending or descending.

- Note that the names of fields with spaces in them must be enclosed in [].
- Note how to specify two fields in an order.
- Set the default values (via the property boxes) to be one of the set values.
- Set the LimitToList properties to Yes.
- See Fig 5.7.2.
- It would be wise to add a validation for the first combo box, to ensure that the user does not leave this field Null.

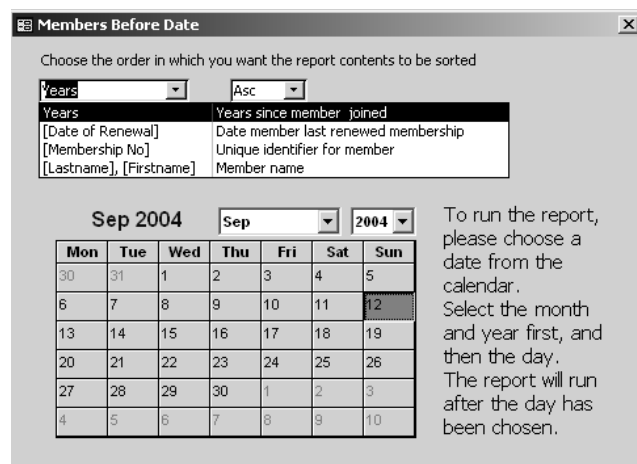


Fig 5.7.2 Sort Order combo boxes

Then add the code shown in Fig 5.7.3 to your Report_Open event.

The field lblSortOrder is a new textbox in the report header which tells the user what order the report is in; add this label to your report.

The names cboSortOrder and cboAscDesc are the names given here to the two combo boxes in Fig 5.7.2. See section 3.4.2 re the combo box Column property.

```
'set up sort order
  OrderBy = Forms![Members before date]!cboSortOrder & " " & Forms![Members before date]!cboAscDesc
  OrderByOn = True
  lblSortOrder.Caption = "Ordered by: " & Forms![Members before date]!cboSortOrder.Column(1) _
    & ", " & Forms![Members before date]!cboAscDesc.Column(1)
```

Fig 5.7.3 Code to set the sort order

You should now be able to order your report in several different ways.

The Multi-Purpose Query example database on <http://www.cse.dmu.ac.uk/~mcspence/Access.htm> has a further example of this.

5.8 Suppressing Detail Lines

Suppose the Chelmer Leisure management decide to run the report using the current date, to list all members, but they do not wish to see details of members who joined in the past year (i.e. those whose years since joining = zero).

Run your report using the current date (if you are running the report in the `ocxCalendar_AfterUpdate` event you may need to set the `calendar Value` property to `Null` to enable the user to select today), and change your data if necessary to get at least one member who has joined in the past year.

You can eliminate these records by adding a new criterion to the query, or you can do this in the report. To code this in the report you use the `Detail_Format` event, as this formats lines before they are printed. If you use the `Detail_Print` event you will get a blank line instead of no line at all. Create this event and code as shown in Fig 5.8.1.

```
Private Sub Detail_Format(Cancel As Integer, FormatCount As Integer)

    If Years = 0 Then
        Cancel = True      'don't print the line
    End If

End Sub
```

Fig 5.8.1 Suppressing lines in a report

Now run your report again, using the current date, and the details of members who joined in the past year should not be shown.

You can also use run-time parameters to decide whether or not to suppress lines, by asking the user a Yes/No question, and taking action appropriate to the answer.

5.9 Exercises.

5.9.1 Member Report

Create a report to list details of all members:

- Show the age at which the member joined (see exercise 3.7.3) and the current age.
- Highlight those members who have not renewed their membership since a given date. Use a calendar control or textbox parameter for the date, and show the date in the report header.
- Allow the user to sort on a variety of fields and show the order in the report header.
- Give the user the option to suppress details of members who have joined in the past year.
- Provide report totals showing the numbers of members in each category, in total and who have not renewed their membership since the given date.

5.9.2 Stock report

- Create a report to list all stock items that need reordering. (Re-order Level \geq Stock), highlighting all stock where the Re-order Level is less than half the Stock level.
- If there are no items to be reordered, then show this information appropriately on the report.
- Allow the user to request the report to be ordered by stock number or stock level, in ascending or descending order. Show this information in the report header.

PART 6 – EMBEDDED SQL

REVIEW OF PART 6:

In this part of the Trainer you will see how to use the DoCmd.RunSQL method to...

- ...add rows to a table
- ...delete rows from a table
- ...update rows in a table
- ...drop (delete) a table
- ...create a table
- ...include values from parameters and variables of different datatypes in SQL
- ...concatenate different parts of SQL with &
- ...embed single quotation marks within a string value

See Appendix F for some Basics of Programming (Datatypes, Literals, Loops)

See Appendix G for an overview of SQL.

See Appendix J for a list of some useful DoCmd methods.

6.1 Introduction

You should already be aware that the Query Design Window is really an SQL-generator, creating the SQL for a query. You can create queries from scratch using SQL or use the Query Design Window visual format to select what you want and let Access create the SQL for you. The SQL can be viewed and created via the *View* menu, or via the usual icon on the query design toolbar.

You have also seen the use of SQL in several sections of this document, including:

- Section 1.8.2 – UNION query for a possible documentation database.
- Section 3.2.4 – comparing the DLookup function with SQL.
- Section 3.4.1.2 – comparing the DCount function with SQL.
- Section 3.4.2 – that a combo box is based on the results of an SQL statement. (So are other objects such as list boxes and charts).
- Section 3.5 – that a filter condition has the same format as the WHERE clause of an SQL statement.

SQL underpins the features of a database, and it is essential that database programmers have, at the very least, a basic understanding of SQL.

You can run queries (that is, run the SQL code) from within VBA. You will have seen in section 1.6 the wizard code generated by the command button Wizard to run a query; this uses the DoCmd.OpenQuery method. If you wanted to run a query from within your code, one method could be to...

- ...create a query
- ...create a wizard command button on a form to run the query and generate the event code
- ...delete the command button (but note that the wizard event code will not be deleted)
- ...call the event code at the point where you want to run the query.

By using the DoCmd.RunSQL method you can code and run SQL directly within VBA. This is a very simple, easy and convenient method to use. This section will illustrate several SQL actions.

You will see in the Further VBA Trainer that you also use SQL with Data Access Objects (DAO) code.

6.2 The RunSQL method of the DoCmd object

The RunSQL Method is used to carry out the RunSQL Action of the DoCmd Object. The general format is:

```
DoCmd.RunSQL "...SQL String..." 'SQL specified within the statement
or
Dim StrSQL as String 'declare a string variable
StrSQL = "...SQL String..." 'assign the SQL statement to the variable
DoCmd.RunSQL strSQL 'run the SQL
```

The second format (where the SQL is assigned to a string variable) is the more flexible and useful format to use because...

- ...you can inspect the contents of the variable at run-time in the Debugger (see section 1.4.3) to see what is in there. This can be especially important with complex SQL and/or where you are including parameter values in the statement. See Fig 6.3.2.
- ...you can build up the SQL in different parts of the code, by assigning the relevant parts as appropriate to the string variable.

This method is used for action and data definition queries only, where the action is used to make the required change to a table. See Fig 6.2.1.

It is not used for SELECT queries as these will return a dataset. If you wanted to select a field or total from a table or query and put the result into a variable, then you can use the Domain Aggregate Functions, as listed in Appendix H and seen in sections 3.2.4 and 3.4.1.2 (and used again later in this section). If you wanted to inspect several rows of data from a table then you need to use a Recordset; this topic is discussed in the Further VBA Trainer.

Type	Query	SQL	Description
Action	Append	INSERT INTO ...	Add rows
	Delete	DELETE ...	Delete rows
	Make-table	SELECT ... INTO ...	Create one table from another, making a copy of the data (e.g. for an archive)
	Update	UPDATE ...	Update rows
Data Definition	Create table	CREATE TABLE ...	Create new (empty) table
	Create index	CREATE INDEX ...	Create new index on a table
	Alter	ALTER TABLE	Change a table structure (e.g. to add new fields)
	Delete table	DROP TABLE	Delete a table and all its data
	Delete index	DROP INDEX	Delete an index from a table

Fig 6.2.1 Action and Data Definition Queries

The DoCmd.RunSQL method works only on MS Access databases. The examples shown here apply only to our Chelmer Leisure database, but the method can also be used to affect tables in another database; see VBA Help.

When you run queries that change a table, you will get various Access confirmation messages, as shown in Fig 6.2.2.

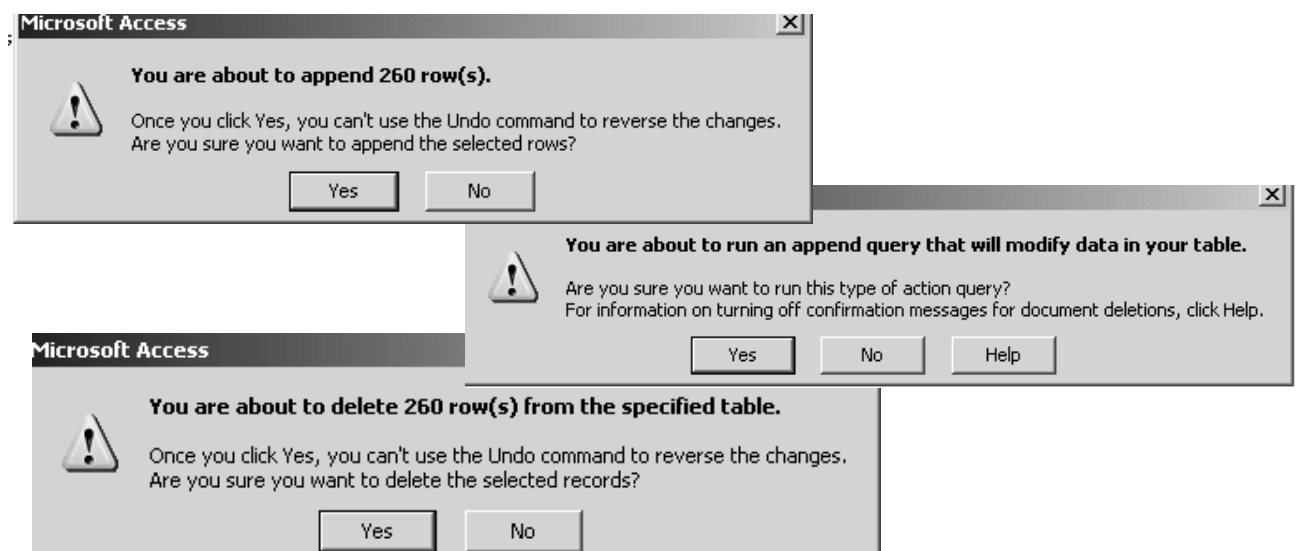


Fig 6.2.2 Examples of Access confirmation messages

You can suppress these via Tools→Options, as shown in Fig 6.2.3. Untick the required box(es) in the Confirm section.

But be careful – this option appears to apply to the machine, not the application! This means that any option set here will apply to all applications run on the same machine, but do not apply to the application itself. This seems very strange to me...

Another (possibly better) method is to code
`DoCmd.SetWarnings False`
 just before a `DoCmd.RunSQL` statement and then code

`DoCmd.SetWarnings True`
 immediately afterwards.

This method will not suppress errors caused by incorrect SQL. It also applies just to the application, so is more convenient for the user, though it requires more coding by the programmer.

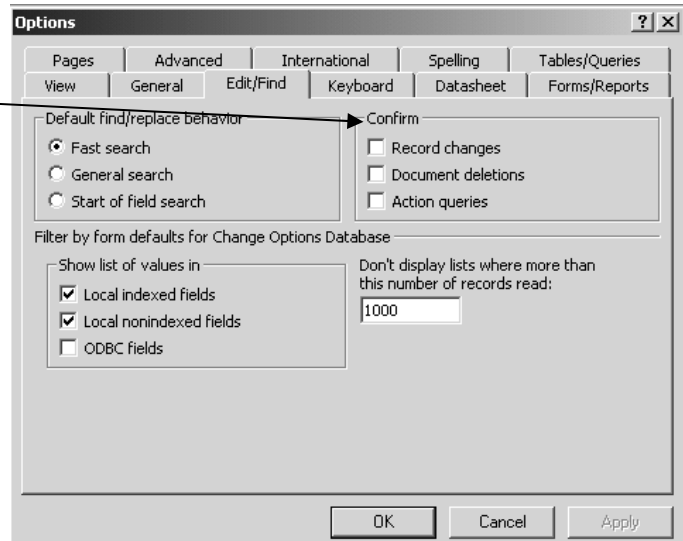


Fig 6.2.3 One way of suppressing the Access confirmation messages

6.3 Adding a row to a table

It could be useful to have a combo box for the Title field on the Membership form, so as to ensure consistency of the data (and spelling) entered here. But if the user is restricted to the combo box entries they would not be able to enter any new titles. By using embedded SQL it is possible to add to the list at run-time.

Do the following (see Fig 6.3.1):

- Create a new table, give it a suitable name such as Title, and enter some titles.
- Change the datatype of the Title field in the Membership table to use a look-up wizard based on the new table.
 - o Set the `LimitToList` property to `Yes`, to prevent the user entering anything other than one of the entries in the list.
- Delete the existing Title field on the form and add the new definition from the field list. Adjust the tab order.
- Try entering a title that is not in the list. You should see the message shown in Fig 6.3.1; this is the effect of setting the `LimitToList` property to `Yes`.

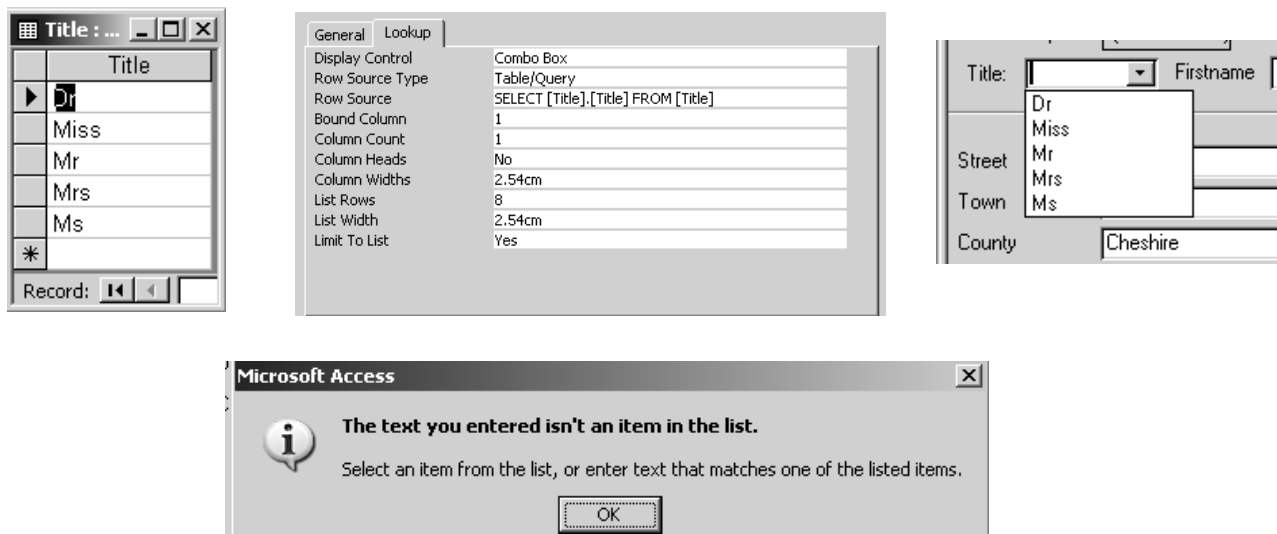


Fig 6.3.1 Setting up a combo box for the Title field on the Membership form and the message that is displayed if you enter a title that is not in the list

If you look at the property box for the Title combo box in the Membership table you will see that there is an event for 'On Not in List'. This event is invoked when the user enters a title not in the combo box list, and we can use this event to update the table with the new title.

Create the event for the title combo box on the Membership form and enter the code shown in Fig 6.3.2.

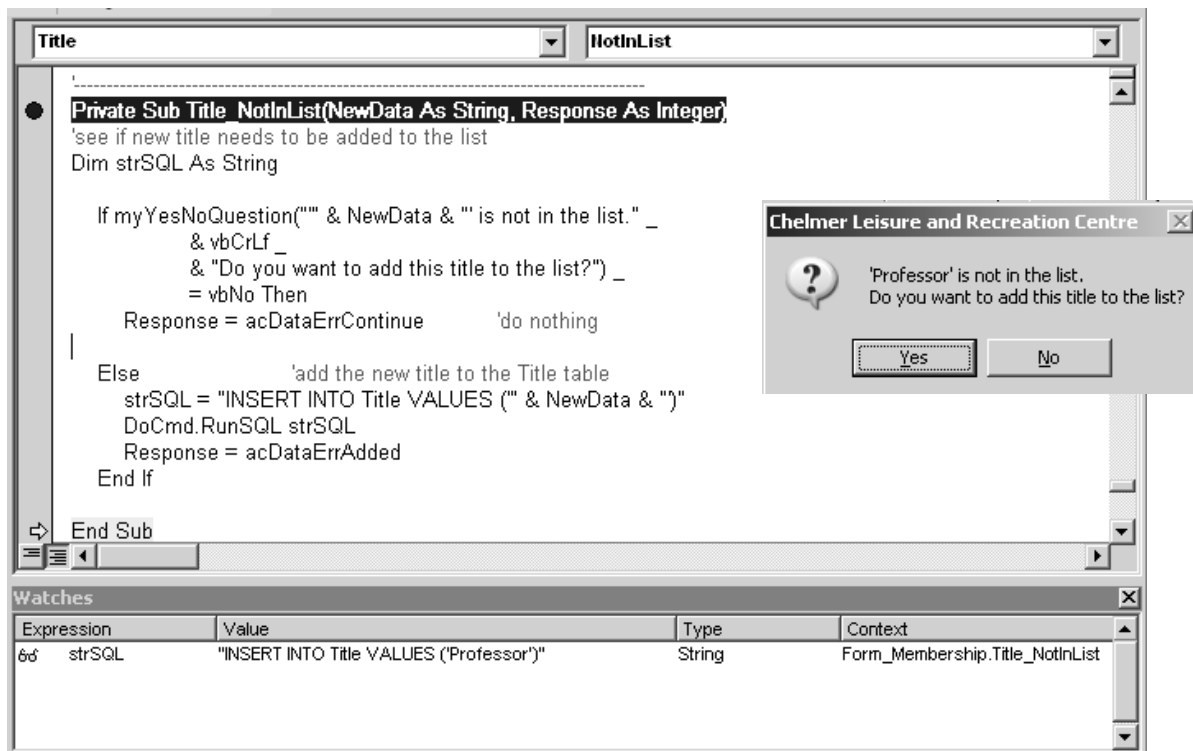


Fig 6.3.2 Code to ask a question of the user and add a row to the Title table if user replies Yes. This shows the code in the Debug window, with a Watch window for the contents of strSQL.

The code in Fig 6.3.2 is explained below:

Private Sub Title_NotInList(NewData As String, Response As Integer)

- Procedure header for a NotInList event for the Title field on the form.
 - The NewData argument contains the value that the user has just entered in the field.
 - The Response argument is for the code to use to tell Access what to do next.

'see if new title needs to be added to the list

- Just a comment.

Dim strSQL As String

- Declares a String variable to store the SQL statement.

```

If myYesNoQuestion("'" & NewData & "' is not in the list." _
    & vbCrLf _
    & "Do you want to add this title to the list?") _
    = vbNo Then

```

- Uses the myYesNoQuestion function (see section 1.7.2) to ask the user whether this is a new title or not (it could, of course, be a typing error).
- Note that this question uses the value in NewData as part of the question. It is shown with single quotation marks around it for emphasis.
 - "' – is the string for the single quotation mark at the start of the user text.
 - & NewData – adds the text that the user has typed in.
 - & "' is not in the list." – adds the single quotation mark at the end of the user text, plus the rest of the sentence.
- The text of the question is split over two lines, using the built-in constant vbCrLf (new line).
- The statement is split over four lines of code, using space+underline at the end of the first three lines.
- Each element of the statement is joined with the concatenation character &.

Response = acDataErrContinue **'do nothing**

- Tells Access to carry on and refuse to accept the new value. The user will have to enter a value again. The Access error message in Fig 6.3.1 will not be displayed.

Else **'add the new title to the Title table**

- The user wishes to add this new title.

strSQL = "INSERT INTO Title VALUES (" & NewData & ")"

- Puts an SQL statement into the string variable. Uses the value in NewData.
 - o The quotation marks are organised: "INSERT ... VALUES (" & NewData & ")"
- See the contents of the Watch window in Fig 6.3.2: "INSERT INTO Title VALUES ('Professor')"
- As for the message displayed for the user, the content of NewData has single quotation marks around it, this time because it is a String value.
 - o A numeric value would not need any quotation marks. A date/time value would need # marks around it. There are examples with different datatypes in the following sections.

DoCmd.RunSQL strSQL

- Runs the SQL statement and adds the value to the table.
- If you wanted to code this line without using a string variable, then you would code:
DoCmd.RunSQL "INSERT INTO Title VALUES (" & NewData & ")"

Response = acDataErrAdded

- Tells Access that the new value has been accepted and the user will now be allowed to carry on.

End If

- End of the IF statement.

End Sub

- End of the sub procedure

There is a third value that could have been used for Response, **acDataErrDisplay**. This tells Access to display the standard error message as shown in Fig 6.3.1.

6.4 Updating a row in a table

Suppose the Chelmer Leisure management want to introduce a new membership category of 7 with a very low annual membership fee of £5.00, just for students.

It is easy to add a new category via the Membership Category form, but not so easy to determine which existing members are students. The staff could go through a membership list, selecting just records where occupation = 'Student', and change each category manually, but this could be time-consuming and error-prone. However, it is simple to effect such a change via embedded SQL.

Do the following:

- Remove the validation check in the Membership table that restricts the Category No to the range 1 to 6.
- Change the Category No field on the Membership form to be a combo box based on the Membership Category table. Set the LimitToList property to True.
 - o This is a much better way of validating the Category Number entered into the Membership record, as it uses the range actually in the Membership Category table, and will not require any further changes to the system should the management wish to introduce more categories in the future.
- Create a new 'Change Membership Categories' form, not based on any table, with three unbound textboxes and an unbound command button, and the code as shown in Fig 6.4.1.
- Add a button to the Membership Category table to open this form. See Fig 6.4.1.

```

Option Compare Database
Option Explicit

Dim strWhere As String      'global variable - used in two places

'-----
Private Sub Form_Load()
'copy Category No into this form
'move cursor to Occupation field

    txtCategoryNo = Forms![Membership Category]![Category No]
    txtOccupation.SetFocus

End Sub

'-----
Private Sub txtOccupation_AfterUpdate()
'display count of members with this occupation

    strWhere = "Occupation = " & Forms![Change Membership Categories]!txtOccupation & ""
    txtNum = DCount("[Membership No]", "Membership", strWhere)

End Sub

```

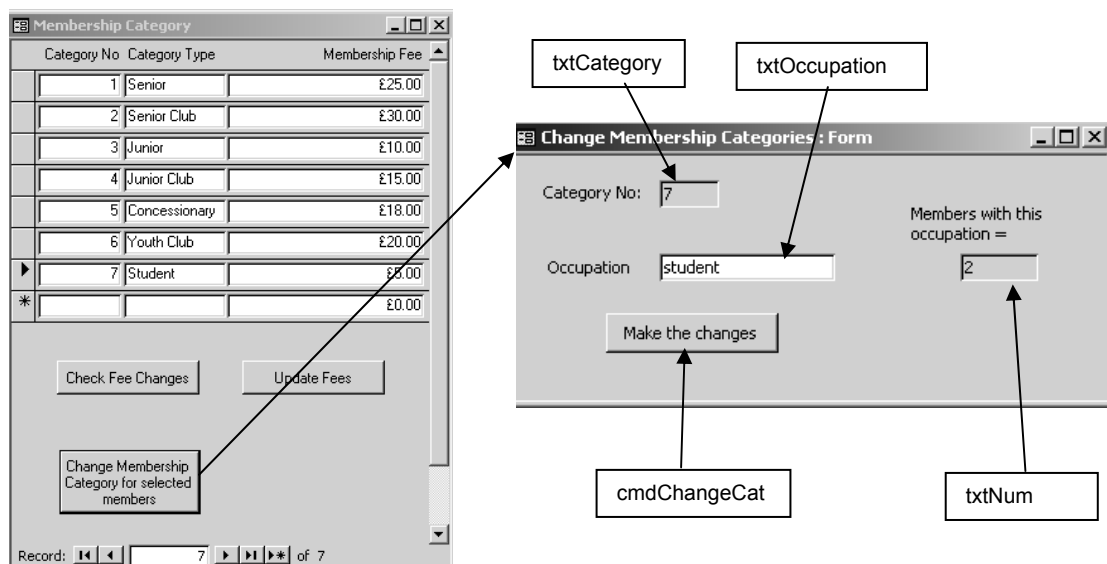


Fig 6.4.1 Change Membership Categories form and opening code

You should be able to understand the code shown in Fig 6.4.1:

- When the new form is opened, the Category No for the current record in the Membership Category form (note the position of the record selector) is copied to the new form and the cursor is positioned in txtOccupation ready for the user to enter the required occupation.
- When the occupation has been entered the code uses the Domain Aggregate DCount function (see section 3.4.1.2) to count up the number of matching records in the Membership table, and puts the result in txtNum.
 - o At run-time, the value in strWhere would be: "Occupation = 'student'"
 - o Note the single quotation marks around the string value 'student'.

Now add the code shown in Fig 6.4.2. This is the code to use embedded SQL to update the Membership table. Check your table and see that the records have now been updated.

- At run-time, the value in strSQL would be:
 - o "UPDATE Membership SET [Category No] = 7 WHERE Occupation = 'student'"
- The Category No is numeric so does not need quotation marks (or anything else) around it.
- Note that this code reuses the WHERE condition set up in strWhere, as the same SQL format is required for the Domain Aggregate Functions. This also ensures that the count and the action are both working with the same table rows.

```

Private Sub cmdChangeCat_Click()
'use embedded SQL to update the Membership table

Dim strSQL As String

    strSQL = "UPDATE Membership SET [Category No] = " & txtCategoryNo & " WHERE " & strWhere
    DoCmd.RunSQL strSQL

End Sub

```

Fig 6.4.2 code to update the membership table.

This is just a very simple example of using UPDATE. It is a moot point whether in this case one should check just for a match with 'student' or look for 'student' anywhere in the Occupation field by coding

strWhere = "Occupation LIKE "*" & Forms![Change Membership Categories]!txtOccupation & "*" as the occupation could have been entered as 'University Student'. At run-time, the value in strSQL would then be:

```
"UPDATE Membership SET [Category No] = 7 WHERE Occupation LIKE "*student*"
```

But this method would also pick up occupations such as 'Student Careers Advisor'. You will need to think carefully about the implications of any method that you use for a real system; this is just a simple example to demonstrate the use of embedded SQL for UPDATE which also uses parameter values from a form.

6.5 Creating and Dropping tables

It is often useful to solve a problem by creating a temporary table, adding the required data, processing it and then deleting (dropping) the table afterwards. The Further VBA Trainer has a large example to create a Booking form that looks like a diary page, and that uses a temporary table to reorganise the bookings data into the required format.

The example to be shown now does not relate to the Chelmer Leisure Centre, but is just a simple example to create 6 lottery numbers, creating a temporary table to store the numbers before displaying them in a list box.

The form is shown in Fig 6.5.1. The list box is an unbound list box (created without the wizard) called lstNumbers with the RowSourceType property set to Table/Query in the property box. The *Close Form* button is a wizard button to close the form.

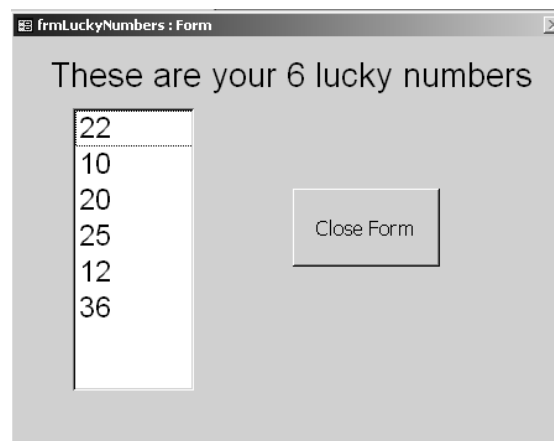


Fig 6.5.1 the Lucky Numbers form.

The code is shown in Fig 6.5.2. This code, although for a simple application, introduces several new concepts. It also uses the Rnd built-in function to calculate random numbers in the required range; look this up in VBA Help and Appendix H.3, it is very easy to use.

```

Option Compare Database
Option Explicit

'=====
Private Sub Form_Load()
"most of the work is done in this event

'define constants for range to be used.
'-----
    'This is better practice than using the literal values in code later
    'It is easy to see what the range is, and to change it if wanted.
Const intLower = 1          'lowest number in range
Const intHigher = 49       'highest number in range
Const intNosRequired = 6   'number of numbers required to be calculated

'define variables for the random number calculation
'-----
Dim intScaleFactor As Integer 'use this with Rnd function to calculate lucky number
Dim intLuckyNo As Integer    'used to store each random number from Rnd function

'initialise variables
'-----
    intScaleFactor = intHigher - intLower + 1 'calculate scale factor to use throughout
                                           'simplifies the calc for Rnd later. – check this in VBA help
    Randomize ' initialise the sequence - will be based on system timer

'get and display the required numbers
'-----
    DoCmd.RunSQL "CREATE TABLE tblNumbers (Num NUMBER)" 'create temporary table

    Do Until DCount("Num", "tblNumbers") = intNosRequired

        intLuckyNo = Int((intScaleFactor * Rnd) + intLower) 'calc next lucky number
        If DCount("Num", "tblNumbers", "Num = " & intLuckyNo) = 0 Then ' we have a new number
            DoCmd.RunSQL "INSERT INTO tblNumbers VALUES (" & intLuckyNo & ")" 'add to temp table
        Else
            'do nothing - already have this number
        End If

    Loop

    'bind listbox to tblNumbers - this will show the numbers
    lstNumbers.RowSource = "SELECT * FROM tblNumbers"

End Sub

'=====
Private Sub cmdClose_Click()
On Error GoTo Err_cmdClose_Click

    lstNumbers.RowSource = "" 'remove binding to tblNumbers before deleting table
    DoCmd.RunSQL "DROP TABLE tblNumbers" 'delete temporary table
    DoCmd.Close

Exit_cmdClose_Click:
Exit Sub

Err_cmdClose_Click:
MsgBox Err.Description
Resume Exit_cmdClose_Click

End Sub

```

Fig 6.5.2 Code to create and drop a temporary table, and calculate and display lottery numbers

Explanation of the **Form_Load** code in Fig 6.5.2:

- Most of the action occurs in this procedure.
- The random number sequence is first initialised.
- A temporary table called `tblNumbers` is created, with just one field called `Num` of datatype `NUMBER`, as it will be a numeric field.
 - o Datatypes that can be used on SQL include: `NUMBER` for numeric data; `CHAR(n)` for string data where `n` = the length of the string field; `DATE` for date/time values.

- The code then has an example of a DO UNTIL ... LOOP to keep repeating the calculation until the required number of numbers has been calculated.
 - The loop condition compares the count of rows in tblNumbers (worked out using the DCount function) with the required number. Initially, of course, DCount will return zero.
 - The Rnd function is used to calculate the next number.
 - The DCount function is used to count up the number of rows that already have this number.
 - If the count is zero then we have a new number and it is added to the table, using the embedded SQL INSERT statement.
 - As the number is numeric, it does not need to be enclosed in anything.
 - Check VBA Help and Appendix F.3.3 for more information on loops.
- When the loop has finished, the RowSource property of the unbound list box lstNumbers is set to SQL that selects all the numbers from the table, so that the numbers will be displayed in the list box on the form.
- The form will now stay up, with the numbers showing, until the user closes the form by using the command button (the form close button in the top right-hand corner has been disabled).

Explanation of the `cmdClose_Click` code in Fig 6.5.2:

- The binding to tblNumbers in the RowSource property of lstNumbers is removed so that the table can now be deleted.
 - This will also clear the contents of the list box, which is why this action is not put in the Form_Load event.
 - If you attempt to delete the table without clearing the RowSource property first you get the error: "The database engine could not lock table tblNumbers because it is already in use by another person or process". This means that you cannot delete a table that is bound to an object on an open form.
- The temporary table tblNumbers is then dropped (deleted) using an embedded SQL DROP statement.
 - Note that an SQL DELETE statement refers to the deleting of rows in a table, so the same word cannot be used for the deleting of the table itself. DROP is used to delete tables, or indexes of tables.
- The form is then closed.

This example has just a simple table with one column. If you wanted to create a table with more columns, with different datatypes, you would code:

```
DoCmd.RunSQL "CREATE TABLE tblExample (ColNum NUMBER, ColText Char(20), ColDate DATE, ColTime DATE)"
DoCmd.RunSQL "INSERT INTO tblExample VALUES(3, 'ABC', #20/3/2004#, #15:00#)"
```

ColNum	ColText	ColDate	ColTime
3	ABC	20/03/2004	15:00:00

Field Name	Data Type	Description
ColNum	Number	
ColText	Text	
ColDate	Date/Time	
ColTime	Date/Time	

Field Properties	
General	Lookup
Field Size	20
Format	
Input Mask	
Caption	
Default Value	
Validation Rule	
Validation Text	
Required	No
Allow Zero Length	Yes
Indexed	No
Unicode Compression	No
IME Mode	No Control
IME Sentence Mode	None

Fig 6.5.3 Example of using CREATE and INSERT with several fields of different datatypes

Section 6.6 has an example of using INSERT for a table that has an AutoNumber primary key.

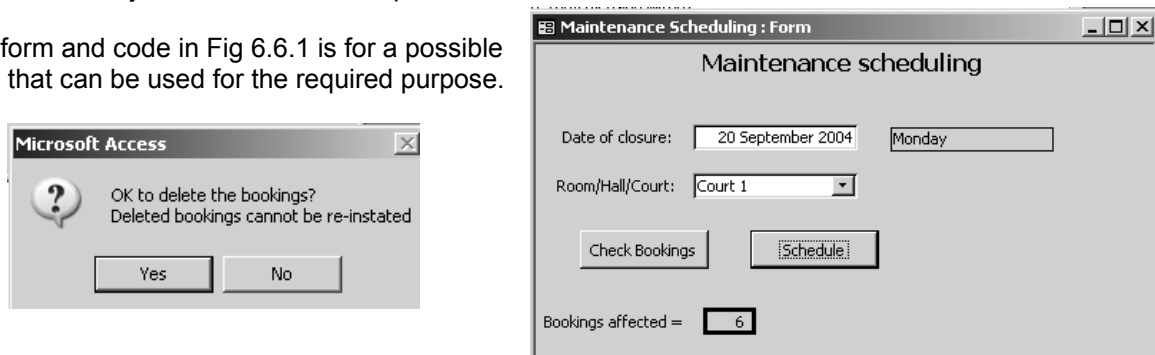
6.6 Deleting a row from a table

Suppose that the Chelmer Leisure management want a facility to allow them to take a room out of use for a day, for maintenance reasons. A possible way to do this could be:

- List all bookings for that room on that day
- Make suitable arrangements to book another room for the same date and time, or contact the class/member if this is not possible.
- Delete the bookings
- Create new bookings for the maintenance (in order to stop the slots being booked by anyone else).

You should know how to create a query to list the affected bookings, so the example here will concentrate on just the last two bullet points above.

The form and code in Fig 6.6.1 is for a possible form that can be used for the required purpose.



```

Option Compare Database
Option Explicit

Dim strCriteria As String
'used from different subs in SQL for DELETE and for DCount
'-----
Private Sub cmdSchedule_Click()

Dim strSQL As String          'for SQL for RunSql action

Dim intCount As Integer      'loop counter
Dim dtBkgTime As Date       'time variable for loop
Const conMembNo = 1         'used to signify maintenance
Const conBkgStartTime = #9:00:00 AM# 'time for first booking in a day

'show count of matched records on the form
myShowCount

'Ask if it's OK to carry on and delete the bookings
If MsgBox("OK to delete the bookings?" _
& vbCrLf & "Deleted bookings cannot be re-instated", _
vbYesNo + vbQuestion) = vbNo Then
'do nothing
Else
strSQL = "DELETE * FROM Bookings WHERE " & strCriteria
DoCmd.RunSQL strSQL 'this will delete the records
'later - code from Fig 6.6.2 will go in here *****
End If

End Sub

'-----
Private Sub myShowCount()

'set up criteria for locating the bookings
strCriteria = "Date = (#" & txtDate & "#)" _
& " AND [Room/Hall/Court] = " & txtRoomHallCourt & """

'Show the number of records found in a textbox on the form
txtNumBookings = DCount("[Booking No]", "Bookings", strCriteria)

End Sub

```

Fig 6.6.1 Form and code for deleting bookings from Bookings table

Explanation of code in Fig 6.6.1:

- The user should enter the date and the room for the maintenance, then choose either the *Check Bookings* button to see the bookings affected (code is not shown here), or the *Schedule* button to delete the bookings.
 - o The code should also, of course, perform suitable validations on the parameter values, but these have been omitted here as they are not the purpose of this example.
- The sub procedure `myShowCount` puts the WHERE criteria for the `DCount` function in the variable `strCriteria`; this is a global variable as it will be used later for the SQL as well. Note the use of # for a date.
 - o At run-time, `strCriteria` would contain:
 "Date = (#20/09/2004#) AND [Room/Hall/Court] = 'Court 1'"
 You can see this by using the Debugger (section 1.4.3).
 - o The value returned by the `DCount` function is put in a textbox on the form.
 - o This code is in a separate procedure so that it can be used from the command button to list the bookings and the command button to schedule the maintenance.
- The code behind the *Schedule* button does the following:
 - o The purpose of the variable `strSQL` should be obvious. The remaining four variables are used by Fig 6.6.2 below.
 - o The procedure `myShowCount` is called to show the count of matching records on the form.
 - o The user is asked whether or not they wish to delete the records, with a reminder that deletions cannot be undone.
 - o If the user replies Yes, then an embedded SQL DELETE statement is used to delete the records. Note the re-use of the WHERE condition in `strCriteria`.

Finally, what needs to be done is some method of stopping anyone making bookings for the room and day, as they are now free as far as the database is concerned. The code in Fig 6.6.2 shows a method of doing this, which assumes for the purposes of this example that Member No 1 has been reserved for maintenance; that is, it is a 'dummy' member record. This method also serves to demonstrate some useful coding.

The database currently only has two types of bookings (Member and Class) but could be extended to include such things as maintenance, public holidays, etc.

```
'Finally, create bookings for all times for that day and room for maintenance
'this code assumes that Membership No 1 is used for maintenance purposes.

dtBkgTime = conBkgStartTime   'for the first booking time in a day
For intCount = 9 To 20       'booking times 09:00 to 20:00
  strSQL = "INSERT INTO Bookings ([Room/Hall/Court], " _
    & "[Member/Class], " _
    & "[Membership No], [Date], [Time]) " _
    & "VALUES (" & txtRoomHallCourt & ", " _
    & "Yes," _
    & conMembNo & ", " _
    & "#" & txtDate & "#)," _
    & "#" & dtBkgTime & "#)"
  DoCmd.RunSQL strSQL         'add the booking record for scheduled maintenance
  dtBkgStartTime = DateAdd("h", 1, dtBkgStartTime) 'add 1 hour to time
  Next intCount

MsgBox ("Maintenance scheduled OK") 'confirmation message
```

Fig 6.6.2 Code to book all time-slots for the room and date

Explanation of code in Fig 6.6.2:

- The code needs to make bookings for all times from 09:00 to 20:00 (assumed here to be the times for which rooms can be booked). This implies that a loop should be used. The loop type that seems appropriate is a FOR ... NEXT loop, counting from 9 to 20. Use VBA Help and Appendix F.3.3 to find out more about FOR loops.
- Prior to the loop starting, the variable `dtBkgTime` is set to a constant value representing 9 AM.
- The code inside the loop is as follows:
 - o `strSQL` is set to an INSERT statement with values for appropriate fields in the new Bookings table row. At run-time this will look like:
 "INSERT INTO Bookings ([Room/Hall/Court], [Member/Class], [Membership No], [Date], [Time]) VALUES ('Court 1',Yes,1,#20/09/2004#,10:00:00#)"
 - Earlier examples of the SQL INSERT statement were for simple tables with just one field, so the statement did not specify the field names.

- This example is more complex as
 - (a) only a selection of fields has data entered, and
 - (b) the Bookings table has an AutoNumber primary key.
- Only the fields that are needed to make the booking are used on this SQL:
 - the room (string datatype)
 - the fact that it is a member booking (Yes/No datatype)
 - the member number (numeric datatype)
 - the date (date/time datatype)
 - the time (date/time datatype)
- The SQL therefore lists the fields that are going to have values entered into them, and then lists the values in the same order.
- The primary key field of Booking No (an AutoNumber datatype) is not in the list; Access will calculate the key value automatically.
- o The SQL is then run.
- o The built-in DateAdd function is used to add one hour to the time in dtBkgTime for the next cycle round the loop.
- o The loop will stop when intCount passes the value 20.
- An information message is displayed after all the maintenance bookings have been made.

Possible problems. Using embedded SQL is a powerful but simple method of making changes to tables. The things that trip students up are usually:

- Omitting to put quotation marks around string values.
- Omitting to put # around date/time values.
- Misspelling table or field names. This may give you the run-time error 2001, 'you cancelled the previous operation'. Not exactly intuitive!
- Missing out spaces needed between elements (e.g. before/after words such as WHERE, AND, OR).
- Missing an & needed to join two elements.
- Missing space+underline at the end of a line when continuing code to the next line.
- Putting calculated dates in SQL in UK format. Access Help states: *"You must use English (United States) date formats in SQL statements in Visual Basic. However, you can use international date formats in the query design grid"*
See VBA FAQ 15 on <http://www.cse.dmu.ac.uk/~mcspence/Access.htm> for more information and for a method of catering for this. Also see Fig 8.3.7.

6.7 Exercises

6.7.1 Add values to Town and County combo boxes at run-time.

Turn the Town and County fields on the Membership form into combo boxes, and allow the user to add new values as required at run-time by using embedded SQL INSERT statements.

6.7.2 Record class sales

Tutors are allowed to take a selection of stock to a class, and record afterwards how many of an item they have sold.

- Add a button to your stock form to allow the tutor to record the number of items sold.
- Validate as appropriate, including that this is not greater than the current stock level.
- Use an embedded SQL UPDATE statement to update the stock amount in the Stock Level table.
 - o As the Stock Level table has a name made up of two words, you must remember to use square brackets around it:
UPDATE [Stock Level] SET ...

PART 7 – MISCELLANEOUS

REVIEW OF PART 7:

In this part of the Trainer you will see...

- ...how to create and use a tabbed form.
- ...how to use the DoCmd.TransferSpreadsheet method to import /export data.
- ...further use of embedded SQL.
- ...information regarding backups, compacting etc.
- ...how to link to an external database.
- ...how to prepare your database for distribution.
- ...using the WITH statement.

See Appendix J for a list of some useful DoCmd methods.

7.1 Introduction

This Part of the document discusses various items that do not naturally belong in another Part.

Many of the items discussed here do not use VBA, but are mentioned for completeness.

7.2 Using tab controls on forms

“With the tab control, you can construct a single form or dialog box that contains several different tabs, and you can group similar options or data on each tab’s page. For example, you might use a tab control on an Employees form to separate general and personal information.”

MS Access 2002 – VBA Help

The tab control is very simple to use via the form design toolbox. See Fig 7.2.1.

If you look at a tab property box, you will see that each is called a ‘page’ and has a set of events for which you can add code. Just as for any other control, you can change the default name to one of your own and add a suitable caption. See Fig 7.2.1.

By clicking on the outer edge of the control, you will see the property box for the full control. Note that you can set the MultiRow property to True if you have a lot of tabs.

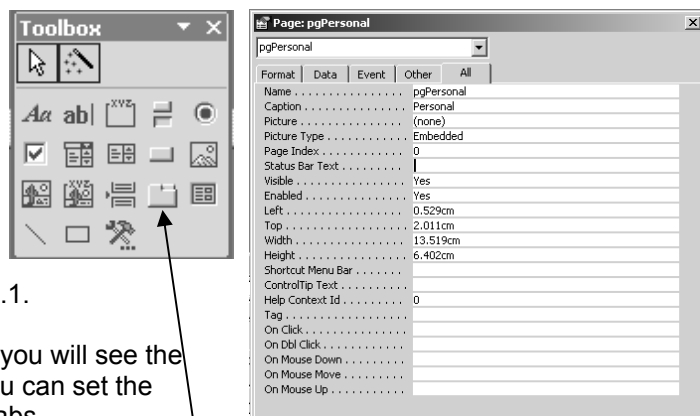


Fig 7.2.1 Toolbox icon, and property box for a tab on a tab control

You will have seen tab controls already – for example, a property box, such as that in Fig 7.2.1, has a tabbed control on the form.

The Membership form has rather a lot of information on it, plus a whole load of buttons and other controls; see Fig 3.5.2 and the exercises in section 3.6. By using tab controls, you can show just the information that you wish at any time and make the form less cluttered. It can also be useful for restricting information that can be seen in a system where different users have different security levels (set the tab Visible property to True to hide the page). The Further VBA Trainer discusses methods of setting security levels for different users.

Do the following: (see Figs 7.2.3 and 7.2.4 for the finished form)

- Create a copy of your Membership form and call it something like 'Membership with tabs'.
- Check your code and change all references to the form name from *Membership* to [*Membership with tabs*].
 - o For example, any forms collection reference such as those used in the various DCount and DLookup functions used for filter counts will need to be changed. If you omit these changes here you will get the error message shown in Fig 7.2.2 (which is not exactly a helpful message!). *Edit* → *Replace* is useful, but you must be careful to distinguish between the form name (which has changed) and the table name (which has not).

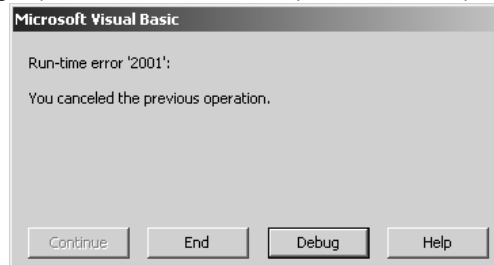


Fig 7.2.2 error if you refer to the wrong form name

- Extend the form downwards and add a tab control (this toolbox control does not have a wizard).
 - o You will get two pages by default, but if you want to add another then right-click on one of the tabs and choose Insert Page. A new page will be added to the right of all the pages.
 - o To change the page order, right-click on a tab and choose Page Order, then set the order that you want.
 - o To delete a page, right-click on the specific page and choose Delete Page.
- Call your tabs pgPersonal and pgOther with captions *Personal* and *Other*.
- Use cut and paste to copy controls from the detail section of your form onto each tab page (dragging the controls doesn't seem to work). Then move things around and adjust the sizes of the tab controls and the form as necessary. The various buttons and controls still work as before – try them out and see.
- You don't have to use the pages for data. Look at the page for *Filters* in Fig 7.2.4.
 - o You may need to re-link the various events to the controls (use the property boxes), but after that they should still work as before – try them out and see.
- You can also use subform data on the form. Look at the page for *Classes Joined* in Fig 7.2.4.
- By default, the page with PageIndex = 0 will be displayed, but you can change this by using the SetFocus method, either for the page itself or for a field on the page. For example, pgOther.SetFocus or [Category No].SetFocus will both cause the *Other* tab to be shown.
 - o Moving between records will keep the same page on display unless you have a SetFocus statement in the Form_Current event that changes the focus to another page.

Fig 7.2.3 Membership form with tab control

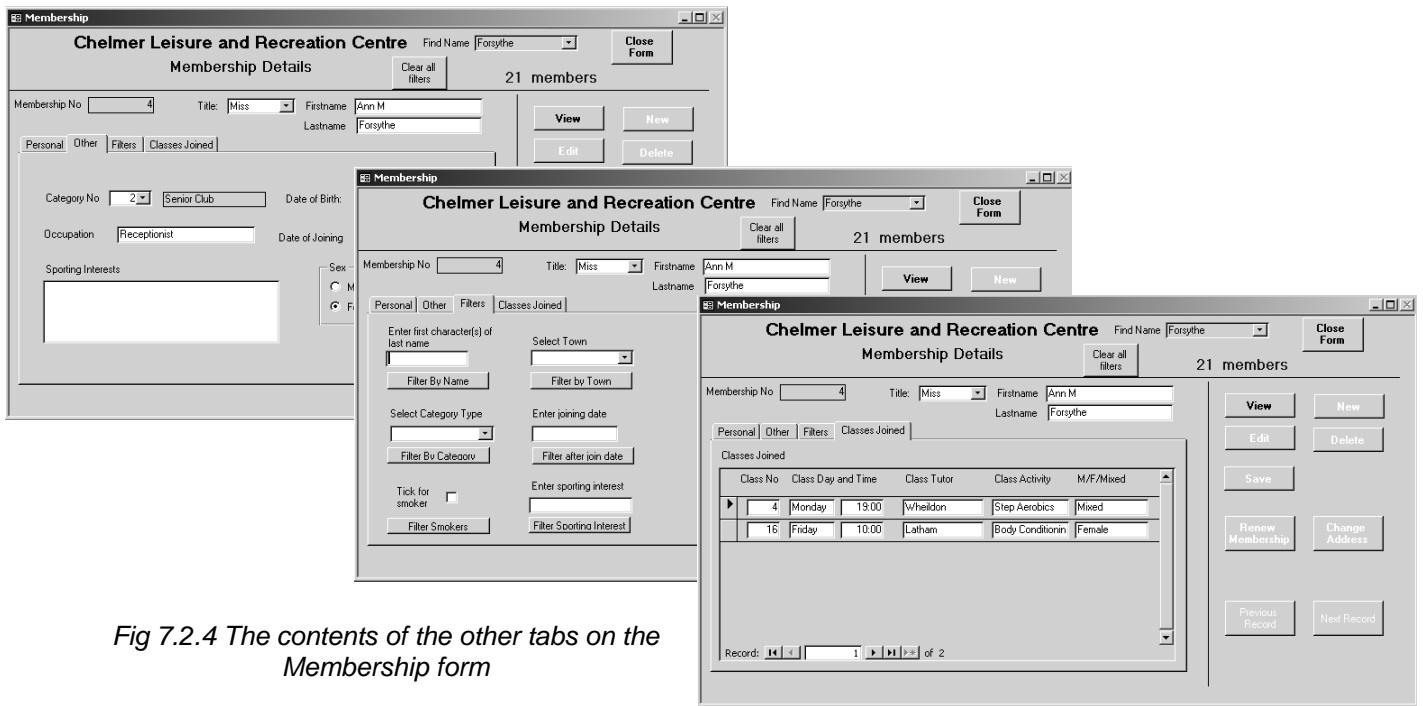


Fig 7.2.4 The contents of the other tabs on the Membership form

7.3 Importing/Exporting spreadsheet data

Suppose that class attendance registers are recorded in spreadsheets on a laptop by each Class Tutor, and that the data is then to be added to the database to keep a record of class attendance for each member. The example described here will show how to use DoCmd.TransferSpreadsheet to import the data into a table in the database.

7.3.1 The TransferSpreadsheet method

This method is used to transfer data in from a spreadsheet, or to create a spreadsheet from data in a table. It uses several parameters (see Fig 7.3.1 and VBA Help), and is coded as shown below (extract is from VBA Help):

```
DoCmd.TransferSpreadsheet(TransferType, SpreadsheetType, TableName, FileName, HasFieldNames, Range, UseOA)
```

Parameter	Import from spreadsheet	Export to spreadsheet	Comment
TransferType	acImport This is the default if this parameter is omitted.	acExport	Specifies the type of transfer. A third option is acLink, for linking to a data source.
Spreadsheet Type	The default assumes an Excel spreadsheet.		There are many values allowed here; see VBA Help.
TableName	The name of the table into which you want to import the data.	The name of the table or query from which you want to export data. You cannot specify an SQL statement.	Imported data is always appended to the named table if the table exists; otherwise a new table is created. Exported data will create a new file or a new sheet in an existing file, according to the version of the software.
FileName	The full path of the file used, as a String expression.		
HasFieldNames	True if the data area has a heading row. False (default) otherwise.	Irrelevant and ignored. Exported data will always have a heading row.	Named columns in an imported spreadsheet do not have to be in the same order as in the destination table. Access will match the names.
Range	A defined range of cells or a named range. Can be prefixed by the sheet name. If omitted then the whole sheet is imported.	Irrelevant, but must be omitted or the export will fail.	
UseOA	This is not explained anywhere in Help, but does not seem to be needed (?)		

Fig 7.3.1 Parameters used for TransferSpreadsheet method

7.3.2 Importing, using a named range and named columns

Stage 1 – Create an Excel spreadsheet as shown in Fig 7.3.2.

- The file is called *Class Attendance.xls*
- The sheet for the data is called *Details*. Other sheets in the file can contain other information if wanted.
- The shaded area has been given the name of *DataArea*
 - Check Excel Help for details on creating, viewing and amending named rows. Use keywords *Name*; *Range*; and look at the item for “Name cells in a workbook”.
- The first row is a heading row.

Stage 2 – Create a table as shown in Fig 7.3.2.

- This is the table into which the data is to be read.

The name of the spreadsheet file and the database table are the same in this example, but they can be different.

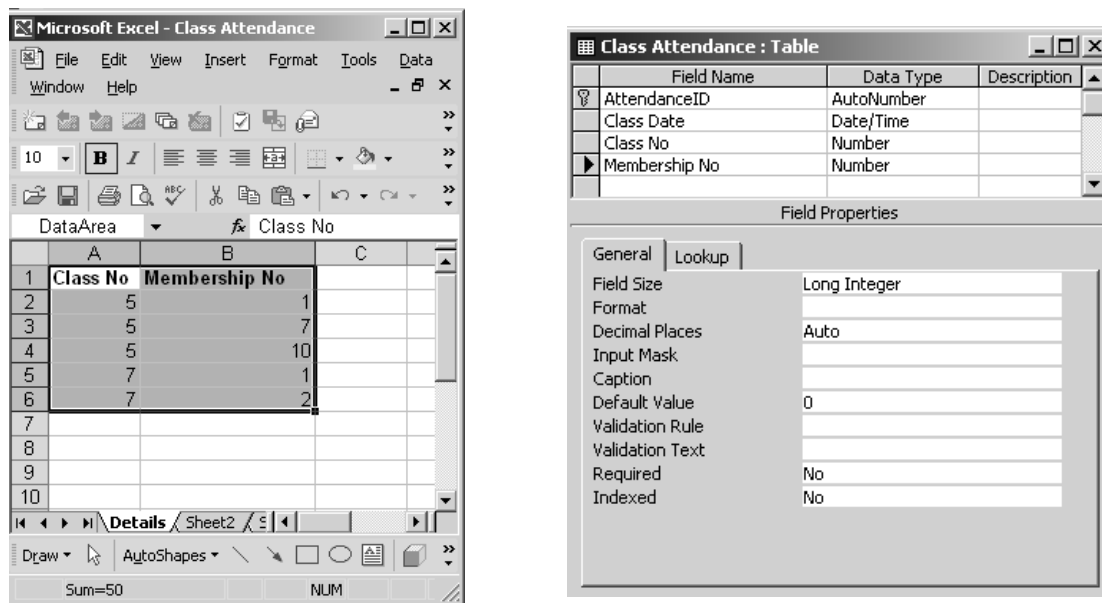


Fig 7.3.2 spreadsheet data and table in database into which data will be imported.

In order to keep this example simple, the spreadsheet data shown includes the Class No, as the same Tutor may run more than one class on the same day. However, it would be just as valid (and probably better) to require a separate sheet for each class, and for the tutor to select the class (possibly from a list) at run-time, with embedded SQL used to add the Class No to each new table row.

The table has the following columns:

- AttendanceID
 - This is an AutoNumber primary key. It is not shown on the spreadsheet data as Access will allocate this value at run-time.
- Class No
 - The value in here will be the value from the spreadsheet cell. The spreadsheet and table columns must have the same name (see section 7.3.4).
- Class Date
 - The example to be used here will request the tutor to enter this value at run-time. The date is then added to each new table row using embedded SQL, so that you can see how it is done. See Fig 7.3.3.
- Membership No
 - The value in here will be the value from the spreadsheet cell. The spreadsheet and table columns must have the same name (see section 7.3.4).

Stage 3 – create a non-wizard command button on your Main Menu to allow the Tutors to record class attendance. Call it cmdClassAttendance. Create a click event for the button as shown in Fig 7.3.3.

```
Private Sub cmdClassAttendance_Click()
'read in data from spreadsheet
'Access will work out the table AutoNumber key
'input data validations omitted for simplicity

Dim dtClassDate As Date

dtClassDate = InputBox("Please enter date of class")    'ask for date
myImportData "Class Attendance"                      'read data from spreadsheet into table
DoCmd.RunSQL "UPDATE [Class Attendance] SET [Class Date] = #" & dtClassDate & "# WHERE [Class Date] IS NULL"
myDisplayInfoMessage ("Data successfully added")

End Sub
```

Fig 7.3.3 Code for cmdClassAttendance button Click event

Explanation of the code in Fig 7.3.3:

Dim dtClassDate As Date

- Just a declaration of a variable, to convert the input data to a date datatype.

dtClassDate = InputBox("Please enter date of class") 'ask for date

- Ask user for the date. This example uses InputBox purely for simplicity here and does not perform any validations on the data (if any) that has been entered. A project or live system should use a text box or calendar control and perform appropriate validations.

myImportData ("Class Attendance") 'read data from spreadsheet into table

- Calls a new procedure (not written yet, so code will not compile yet) to read the spreadsheet.
 - o The procedure requires just one parameter, being the (same) name of the spreadsheet and the table.
 - o This code has been put into a separate procedure as you may find it useful to use/adapt if you need to read several different sheets into database tables in the same system.
 - It also demonstrates the usefulness of using separate procedures.

DoCmd.RunSQL "UPDATE [Class Attendance] SET [Class Date] = #" & dtClassDate & "# WHERE [Class Date] IS NULL"

- The procedure myImportData does not delete any data from the Class Attendance table, so the new rows are added to any existing data. These rows will not have anything in the Class Date field.
- This DoCmd.RunSQL statement shows how to add a value to the new rows in the table.

myDisplayInfoMessage ("Data successfully added")

- Just a feedback message to the user.

Stage 4 – In a separate Access module, create the procedure shown in Fig 7.3.4

```
Public Sub myImportData(prmName As String)
'read in spreadsheet into table of the same name
'sheet has a named data area

'constants
Const strPath = "A:\"           'path for the input spreadsheet
Const strSheetRange = "DataArea" 'name of range on sheet for the data – see Fig 7.3.2

'variables
Dim strFullPath                'path and name for spreadsheet

'set up full path - assumes sheet is called prmName.xls
strFullPath = strPath & prmName & ".xls"

'read the named range into the table
DoCmd.TransferSpreadsheet acImport, , prmName, strFullPath, True, strSheetRange

End Sub
```

Fig 7.3.4 Procedure to read in data from a named range on a spreadsheet

Explanation of the code in Fig 7.3.4:

Public Sub myImportData(prmName As String)

- Procedure header, with one parameter, the parameter value being used for both the spreadsheet name and the table name, which are assumed to be the same in this example.
 - o Use two parameters if the names are different and adjust the code accordingly.

Const strPath = "A:\"

'path for the input spreadsheet

Const strSheetRange = "DataArea"

'name of range on sheet for the data

- As the data is physically carried on some form of device other than a hard disk, the drive for the device needs to be specified here. The code here assumes that the data is on a floppy disk which the machine recognises as A:
 - o If the data was in a separate area on the network, on a drive called R:, in a folder called Class Data, then the code would be:


```
Const strPath = "R:\Class Data\"
```
 - o If the data was in the My Documents folder of the current machine, then the code would be:


```
Const strPath = ""
```
 - o If the data was in a folder called Class Data within the My Documents folder of the current machine, then the code would be:


```
Const strPath = "Class Data\"
```
- It is probably best to require that the user must always put the data in the same area, to save mistakes at run-time.

Dim strFullPath

'path and name for spreadsheet

- Variable to contain the full pathname. No need to put this in a variable, but it is useful to code this way as you can examine the contents at run-time in case of errors, using the Debugger.

strFullPath = strPath & prmName & ".xls"

- Joins the path, the filename and the spreadsheet extension.
- At run-time, this will contain "A:\Class Attendance.xls"

DoCmd.TransferSpreadsheet acImport, , prmName, strFullPath, True, strSheetRange

- This line will import the data.
 - o acImport is used (although it is also the default) to specify that this operation is to import data.
 - o The type of spreadsheet is not specified, so the default of an Excel spreadsheet is assumed.
 - Note that you must include the comma-separator when an argument has been omitted.
 - o The value in prmName is used to name the destination table.
 - o The value in strFullPath is used to tell Access where to find the spreadsheet.
 - o The sheet contains a heading row, so the 5th parameter specifies True.
 - o The value in strSheetRange is used to specify the data range on the sheet.
 - This value does not tell Access which sheet to look in, but this is not necessary as range names must be unique in the file, so Access can work out where to look.
 - If you wanted to specify the sheet, then it would seem logical to code:


```
Const strSheetRange = "Details!DataArea"
```

 but this generates the error shown in Fig 7.3.5. I have no idea why this doesn't work with a named range, or why the exclamation mark (!) becomes a dollar sign (\$).

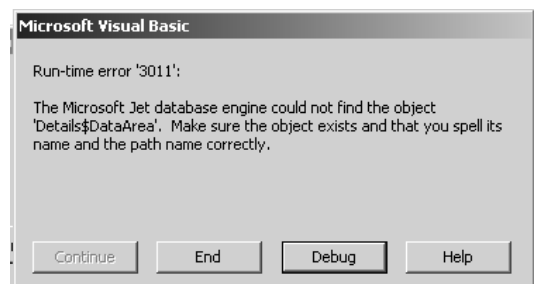


Fig 7.3.5 error message when referencing named range plus sheet name

Stage 5 – try it all out

With your spreadsheet file in the area specified by strPath, click on the new *Record Attendance* button on your main menu, enter a date, and see what happens. You should get the message that the data has been added. Look at the table and see the data, with the date now in the rows.

Try again with a new date, and check the result. The new rows should have the new date and the existing rows should be unchanged.

7.3.3 Importing, using un-named columns

Delete the heading row from your spreadsheet and change the value for 5th parameter on the DoCmd.TransferSpreadsheet command from True to False.

When I try this I get the error message shown in Fig 7.3.6. I have no idea why.

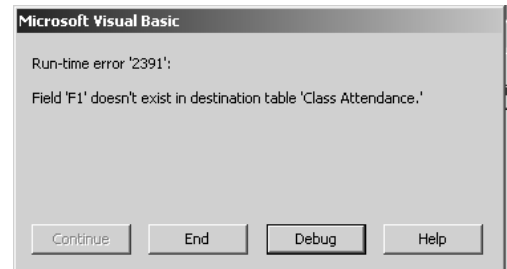


Fig 7.3.6 Error when using un-named columns

The same error will also occur when using named columns if there is a column in the spreadsheet that is not also in the table. The reference to 'Fn' in the message does not mean spreadsheet cell or column F or Fn, but means that the nth column has a name that is not also in the table.

7.3.4 Importing, using a defined range

Rather than using a named range you could have the range defined on the sheet.

Change your spreadsheet to delete the named area DataArea (check how in Excel Help) and add three new rows above the data, as shown in Fig 7.3.7.

This sheet now has two sets of data which means that the sheet is opened and read twice:

- 1) To get the data range specified in A1:A2 (assumed to be a fixed range).
- 2) To read the data in the range specified by cell A2.

It is possible to use a macro in Excel to calculate the range specified in cell A2, so that this adjusts automatically. This may be preferable to requiring the Tutor (who may not be very familiar with Excel) to check and adjust a named range. These cells can then be locked and/or hidden; Access will still be able to read them.

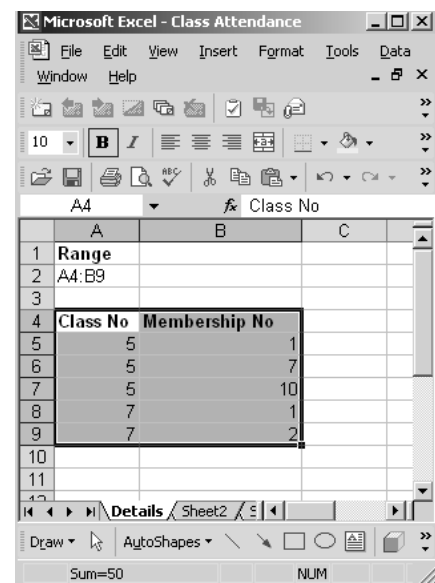


Fig 7.3.7 Defining the range on the sheet

You will need to create a new table called *CellRange* with just one text column called *Range*. The value in cell A2 will be read into here.

Rename your *myImportData* procedure as something different (for example *myImportData1*) and create a new *myImportData* procedure as shown in Fig 7.3.8.

The code is explained below:

```
Const strPath = "A:\"
Const myconCellRange = "A1:A2"
Const myconSheet = "Details"
```

- The first constant is the same as before.
- The second constant defines the fixed range on the sheet used to define in turn the data area.
- The third constant specifies the name of the data sheet within the file. If this is not used than Access assumes that the data is on the first sheet, which is fine as long as no-one inserts a sheet in front of the data sheet. So it would be prudent to specify a named sheet in this case.

```
Dim strSheetRange As String
Dim strCellRange As String
Dim strFullPath
```

- Variables to store ranges and the full path name for the file.

DoCmd.RunSQL “DELETE * FROM CellRange”

- As DoCmd.TransferSpreadsheet will add to any existing data in the table, we need to delete any previous (and thus unwanted) values from the CellRange table.

strFullPath = strPath & prmName & “.xls”

- Same as before.

strCellRange = myconSheet & “!” & myconCellRange

DoCmd.TransferSpreadsheet acImport, , “CellRange”, strFullPath, True, strCellRange

strSheetRange = myconSheet & “!” & DLookup(“Range”, “CellRange”)

- The first line combines the name of the sheet with the fixed range. At run-time, strCellRange will contain “Details!A1:A2”. Unlike the problem mentioned in Stage 4 of section 7.3.2, this works.
- The second line reads in the value in cell A2 and puts it in the CellRange table.
- The third line uses the DLookup function to get the value from the CellRange table, and combine it with the name of the sheet. At run-time, strSheetRange will contain “Details!A4:B9”.

DoCmd.TransferSpreadsheet acImport, , prmName, strFullPath, True, strSheetRange

- This reads the same spreadsheet again, this time to get the data in the second range and to put it in the required destination table..

```
Public Sub myImportData(prmName As String)
'read in spreadsheet into table of the same name

'constants
  Const strPath = "A:\"           'path for the input spreadsheet
  Const myconCellRange = "A1:A2" 'fixed cells for cell range on each file
  Const myconSheet = "Details"   'need to specify sheet, otherwise assumes first sheet

'variables
  Dim strSheetRange As String    'sheet range
  Dim strCellRange As String     'cell range
  Dim strFullPath                'path and name for spreadsheet

'remove existing data from cell range table
  DoCmd.RunSQL "DELETE * FROM CellRange"

'set up full path - assumes sheet is called prmName.xls
  strFullPath = strPath & prmName & ".xls"

'first get the cell range info into the CellRange table
  strCellRange = myconSheet & "!" & myconCellRange
  DoCmd.TransferSpreadsheet acImport, , "CellRange", strFullPath, True, strCellRange
  strSheetRange = myconSheet & "!" & DLookup("Range", "CellRange")

'then use the cell range to read the specified range into the table
  DoCmd.TransferSpreadsheet acImport, , prmName, strFullPath, True, strSheetRange

End Sub
```

Fig 7.3.8 Reading in data from a range specified on the sheet

7.3.5 Exporting data

This is simpler than importing as you do not need to match column headings in a spreadsheet with a table, nor do you need to have some way of specifying the data range.

A system may have one or more functions that display query result dynasets; it could be useful to offer the user the option of saving the data to a spreadsheet.

Look back at the example in section 1.6, where the query result showing the effect of changes that may be applied to the Membership Fee is displayed. Add the following line to the cmdCheckFees_Click() event, after the line for the DoCmd.OpenQuery statement.

```
myExportData stDocName, "CheckFees"
```

- myExportData is a new public procedure (see Fig 7.3.9), which requires two parameters.
 - o The first parameter is the name of the table or query from which the data is to be exported. This will therefore be the name already in stDocName in the wizard RunQuery code.
 - o The second parameter is for the filename of the spreadsheet.

```

Public Sub myExportData(prmQueryName As String, prmFileName As String)
'ask if the user wants to export the data to a spreadsheet, etc.

'Const strPath = "A:\"           'path for the input spreadsheet
Const strPath = "VBA stuff\Trainer V5\"
Const myconExportMsg = "Do you want to save this file to a spreadsheet?"

Dim strFullPath As String
Dim strDateTime As String
Dim strFileName As String

If myYesNoQuestion(myconExportMsg) = vbYes Then

    strFullPath = strPath & "Export\" 'assumes all files put in Export folder

    strDateTime = Format(Date, "dd-mm-yyyy") & "_" & Format(Now(), "hh-mm-ss")
    strFileName = prmFileName & "_" & strDateTime

    DoCmd.TransferSpreadsheet acExport, , prmQueryName, strFullPath & strFileName

    myDisplayInfoMessage "data has been saved as Excel spreadsheet:" _
        & vbCrLf _
        & strFullPath & strFileName

End If

End Sub

```

Fig 7.3.9 Code for procedure to export data to a spreadsheet

Explanation of code in Fig 7.3.9:

Const strPath = "A:\" 'path for the output spreadsheet
Const myconExportMsg = "Do you want to save this file to a spreadsheet?"

- strPath serves the same purpose as before, and here assumes that the output is to be on a floppy disk.
- The second constant is simply to be used for a message/question box.

Dim strFullPath As String
Dim strDateTime As String
Dim strFileName As String

- strFullPath serves the same purpose as before, this time for the full path where the output file is to be written to.
- strDateTime will be used to store the system date and time. If it is required to ensure that each output file is a separate file, then there needs to be some way of ensuring that the files have unique names; using the system date and time as part of the filename could be a useful way of achieving this.
 - o It could also be useful to add the user's log-in ID to the filename; the Further VBA Trainer shows how you can access this ID.
- strFileName will be used to store the full path and filename of the output file.

If myYesNoQuestion(myconExportMsg) = vbYes Then

- Ask the user whether or not they wish to export the data. See Fig 7.3.10

strFullPath = strPath & "Export\" 'assumes all files put in Export folder

- This line starts to set up the value in strFullPath. The code assumes that all exported data is to be put in a folder called 'Export'.
 - o It is possible to ask the user where the data file is to go, but if the user enters the wrong information the export may fail.

strDateTime = Format(Date, "dd-mm-yyyy") & "_" & Format(Now(), "hh-mm-ss")

strFileName = prmFileName & "_" & strDateTime

- Use the built-in Format function to format today's date and time as a string.
- Add this new value to the name for the file.

DoCmd.TransferSpreadsheet acExport, , prmQueryName, strFullPath & strFileName

- Export the query dataset to the spreadsheet.

**myDisplayInfoMessage “data has been saved as Excel spreadsheet:” _
& vbCrLf _
& strFullPath & strFileName**

- Tells user where to find the spreadsheet file. See Fig 7.3.10.

End If

- Simply terminates the If.

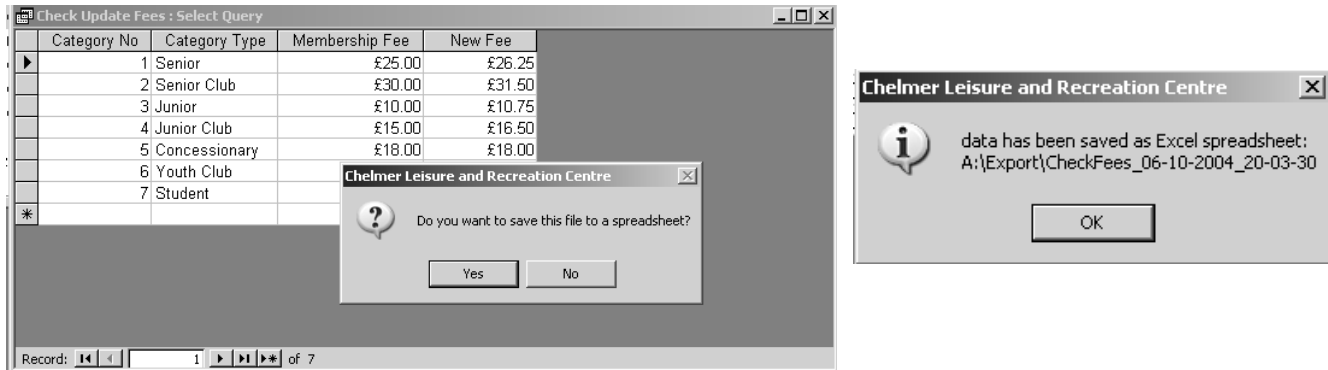


Fig 7.3.10 Saving the Update Fees query result to a spreadsheet.

If you have added Category Type 'Student' (Category No = 7) as used in section 6.4 you will need to need to change your code for your myUpdateFee function to cater for category No 7.

7.3.6 Some errors you may encounter

If the error has an identifiable error number, then you should be able to trap this via an On Error statement in your code. You could then give an alternative error message. Or you could take appropriate action, such as asking the user to input the filename again, or choose whether to cancel or continue. Error-trapping is discussed briefly in this document and at more length in the 'Further VBA' Trainer.

(a) Run-time error 3044: '<path>' is not a valid path. Make sure that the filename is spelled correctly and that you are connected to the server on which the file resides.

This occurs when importing or exporting data and could mean one (or more) of the following:

1. The path in the code is incorrect.
2. The filename in the code is incorrect.
3. You are not connected to the network (if using a common network area).
4. You do not have permissions to access the network area (if using a common network area).

(b) Run-time error '3011': The Microsoft Jet Database Engine could not find the file '<path>'. Make sure the object exists and that you spell its name and the pathname correctly.

This occurs when importing data and normally means that the file does not exist in the folder specified. See also Fig 7.3.5.

(c) Invalid use of Null.

This occurs when importing data and can mean that a spreadsheet has a blank row, a blank field where data is expected, or the specified area includes blanks at the end. But this error doesn't always seem to happen for this condition. It may depend on the datatype of the field in the table.

(d) Run-time error '2391': Field 'Fn' doesn't exist in destination table '<name>'.

This occurs when importing data. See Section 7.3.3.

(e) Run-time error '3051': File is locked for editing.

This occurs when importing data, if the file is currently in use by another user. If the file is not vital to the process, you could trap the error and give the user the opportunity of cancelling or continuing.

7.4 Backups, Compacting, etc

Read 'Using Access97' by Roger Jennings, published by QUE, pages 922-924 for some very good advice. There should be a copy of the book in the library.

7.4.1 Making Backups

Do not forget the importance of providing backup and retrieval facilities for your database. For yourself, save your database on more than one backup disk, and keep them in separate places. It is recommended that you have a cycle of disks, with documentation to say at what stage each backup was taken and what has been done since the backup. Keep printouts of code so that you can recreate it if you lose it. For most students, these precautions will not be needed. But you can't guarantee that your database will be safe – remember, 'it could be you!'

For a commercial system, backup, retrieval and logging procedures are vital. Any system that you develop, be it a project or a system for a 'real' user, must have these procedures in place.

7.4.2 Compacting your database (keeping the size down)

Look at *Tools* → *Database Utilities* for information on repairing and compacting databases. Also note that databases that have records added to, and deleted from, them may need defragmenting at intervals to speed up access and to reduce the size.

While you are developing your system, you may be experimenting with data, queries, forms and the like. Access will grab extra space for each change that you make, but the unwanted bits are still in your database, which can grow pretty large very rapidly. It is helpful periodically to repair and compact your database as this will free up space and reduce the overall size. Access 2002 introduced the facility to perform automatic compact and repair procedures on a database, whenever the database is closed. Look at *Tools* → *Options*, and then at the *General* tab. Also see the 'New features for Access 2000' page of <http://www.cse.dmu.ac.uk/~mcspence/Access.htm> .

Access needs adequate space on your machine to undertake this operation, as it creates a second version of your database, copying over just the non-deleted bits, and deletes the original. You can even lose your database if this operation goes wrong! (It's happened to me...). See Access FAQ 27 on the Frequently Asked Questions page of <http://www.cse.dmu.ac.uk/~mcspence/Access.htm> .

It may be wise not to use 'Compact on Close' and to make a back-up before attempting to compact your database, just in case...

7.5 Linking to an external database (separating data from the rest)

So far, in this database, the tables, queries, code, etc have all been contained in the same physical file. However, in practice, tables and the rest (the application) tend to be separated, with the tables on a server and the application on the local machine(s). The database with the tables is usually known as the 'back-end' and the database with the application is usually known as the 'front-end'.

This has various advantages:

- Other applications can run on the same tables (by linking to the tables required).
 - For example, there could be an application for the Chelmer Leisure staff who register membership details, make bookings etc, and another application for Management to use for various management reports and statistics. The two applications could both use the same back-end database.
- An application can be maintained and tested 'off-line', using a copy (not the real thing!) of the live data for the final testing.

- Distributing and installing a new version of an application is simpler (as it does not affect the data, unless the table structure has been altered).
- You could replace a VBA application by a VB application if wanted (I think).
- Backing-up the data can be done without having to back up the entire application each time.
- Back up versions can be tested – it's a bit late to find out that the device is faulty when you need to restore from a backup following loss or corruption of the main back-end database.
- This facilitates use of an application over a network with each user having a copy of the front-end linked to the same back-end on a network server.
- Users cannot change back-end table design via a front-end application.

Look at *Help* for further information regarding linking external data and database objects.

Splitting a database is very simple to do. The following shows firstly a DIY method (here purely so that you can see how to set up links should you need this in future and in case the Database Splitter wizard is not installed on your machine) and secondly how to use the Database Splitter wizard.

7.5.1 DIY method

Do the following with your Chelmer Leisure database:

- Make two copies of your Chelmer Leisure database and call them CL Code and CL Data.
 - Open the CL Data database and delete all queries, reports, forms, code modules – i.e. leave only the tables remaining. This will be the back-end database.
 - Open the CL Code database and delete all the tables. This will be the front-end application database.
- With the CL Code (front-end) database open at the tables tab of the database window link to the tables in the CL Data (back-end) version of the database by:
 - *File* → *Get External Data* → *Link Tables*
 - Select the required database (your CL Data (back-end) database) in the *Link* dialog box
 - Click *Link*
 - Choose 'select all' and click *OK*
 - That's it – the tables will now appear in the database window, but with an arrow to the left hand side of the table icon. This arrow means that the table is a linked table – i.e. it is not part of the database file, but links to a table in another physical file. See Fig 7.5.1.
- Open one of the tables in design view.
 - Access will give a message telling you that you cannot alter the table structure within a linked database – you can only alter the table structures within your CL Data database.
 - Look at the property box for the table and note the entry under 'Description' – this gives the full path to the linked database, plus the name of the table in that database.
 - You can also see the link path in a 'control tip' box by moving the cursor over the table name in the database window. See Fig 7.5.1.
- Try running some of the queries, reports, forms etc in your CL Code database – they should all work exactly as before.
- Repair and compact the two new (CL Data and CL Code) databases, to free up unused space.
 - If you have set the Main Menu to open automatically when the database is opened (see section 4.2.4) you will get a message to the effect that the main menu is not available when you open the CL Data database. Use *Tools* → *Startup* to remove this option.

Only the raw data tables and look-up tables need to be in a back-end database. Any intermediate or temporary tables are probably best defined in (or definitions and data moved to) the front-end application in order to avoid conflicts of use.

For example, the CellRange table from section 7.3.4 is now in the back-end database. This table would be best as a local table as different users may be reading different spreadsheets and there could be a conflict if two users are doing this at the same time.

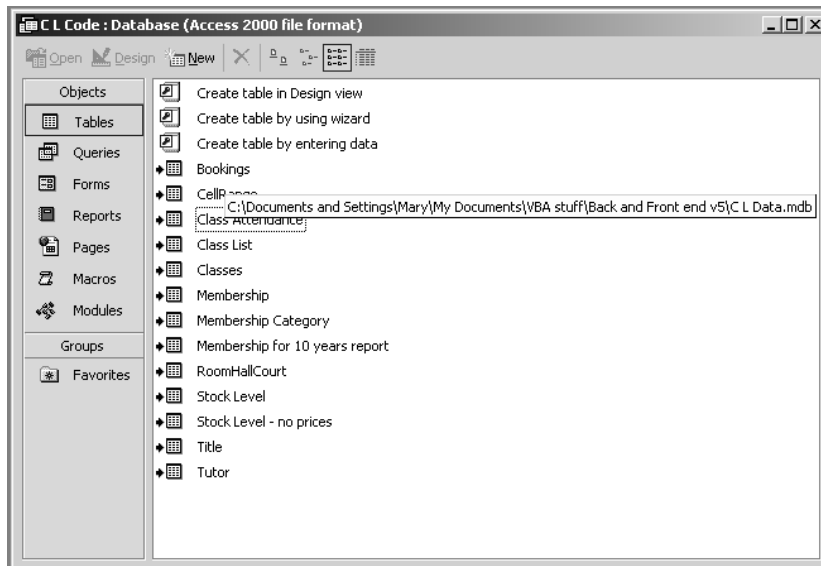


Fig 7.5.1 CL Code (front-end) showing links to back end tables in CL Data

7.5.2 Using the Database Splitter wizard

This is much simpler than doing it yourself, if you have the wizard installed. Do the following

- Make a copy of your database (as the wizard will change it)
- Access the wizard via *Tools* → *Database Utilities* → *Database Splitter*
- Choose the required directory when prompted
- Click on the *Split* button and the wizard will do the rest

Look at the contents of your database directory.

- You will see a new database with the name <myname>_be.mdb. This is the back-end database with just the tables. Open it up and look at the contents.
- Your original database has now been changed to be a front-end database. Look at the tables – these are now links to your back-end database as in Fig 7.5.1. Everything else is the same. You may need to repair and compact it to free up unused space.

7.5.3 Re-linking after a back-end has been moved or renamed

If you look at *Tools* → *Database Utilities* you will see an option for *Linked Table Manager*. Use this to update links if the back-end database location is moved or renamed. If the wizard is not installed, then it is simple to do it manually using *File* → *Get External Data* → *Link Tables* - see section 7.5.1; you will need to delete the old links first.

7.5.4 Multi-User access to a back-end.

Many (most?) database systems will be used by several people, each with a copy of the front-end application linked to the same back-end tables. So, what happens when two (or more) people wish to access and update the same record?

Using *Tools* → *Options* you can specify what you want to happen. The *Advanced* tab contains the dialog for you to specify the options that you want. The settings shown in Fig 7.5.2 are the default settings.

MS Access Help contains information about what these various settings mean. Use the Keyword *share*, and look at items titled *Share a database* and *Set options for a shared database*.

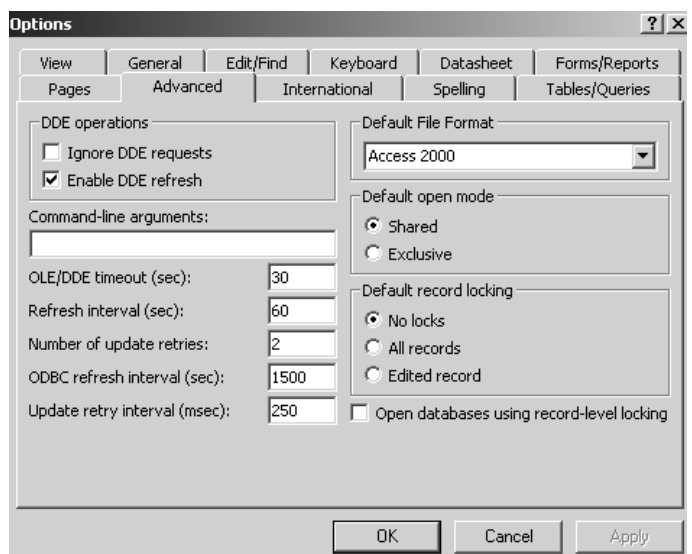


Fig 7.5.2 How to specify sharing options

7.6 Preparing your database for distribution

7.6.1 Removing the database window and menu bars

Before you experiment with any of this, make a back-up copy (or two) of your database so far. In a distributed version you would not normally want the user to be able to get at any design features or tables. This could also apply to you.

Look at the *Tools* → *Startup* dialog box shown in Fig 7.6.1 and discussed in Fig 7.6.2. You specify here the options that you want for your distributed version. If you right-click on an item you will get a little *What's This* button, and if you click on that you will get a brief explanation of the function of the option.

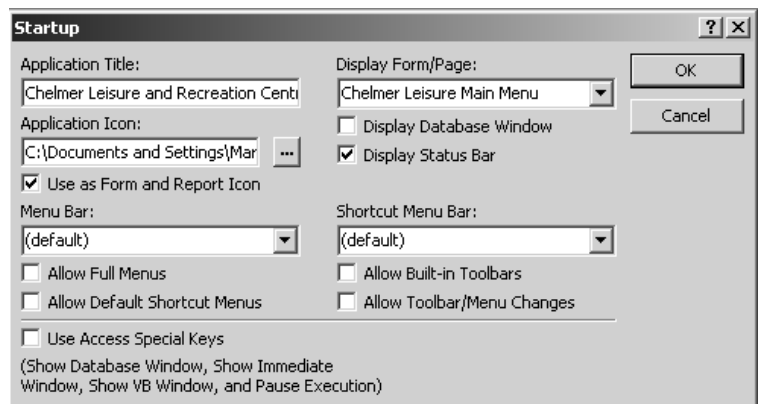


Fig 7.6.1 Suggested options for distributed version

Option	Restart needed?	Comment
Application Title	No	Replaces the default text of 'Microsoft Access' in the bar at the top of the application window. Note that you cannot reference any public constant such as myconChelmerName in here.
Application Icon	No	Replaces the default MS Access icon in the bar at the top of the application window. Must be a .ico or .bmp file.
Menu Bar	Yes	Replaces the default MS Access menu with the specified menu of your own. (Look up <i>Create a Toolbar</i> in Access Help).
Allow Full Menus	Yes	Specifies whether or not you want the user to have access to all menus and be able to make changes to the database structure.
Allow Default Shortcut Menus	Yes	Specify whether or not you want the user to use right-click to see the default shortcut menus.
Allow Special Keys	Yes	Specify whether or not you want the user to be able to use special shortcut keys to give access to various design features.
Display Form/Page	Yes	Specify the form or page that you want to be displayed automatically when the application is opened. Leave blank if you do not want a form/page displayed.
Display Database Window	Yes	Specify whether you want the database window to show (normal when developing) or be hidden (normal for a distributed version).
Display Status Bar	Yes	The status bar displays various information messages, including field comments in table design when the field is bound to a control on a form. The text displayed is from the StatusBarText property of the control on the form. Use this option to specify whether or not you want this bar to show.
Shortcut Menu Bar	Yes	Specify a custom shortcut menu to use for forms and reports in place of the default MS Access bar.
Allow Built-in Toolbars	Yes	Specify whether or not you want the default MS Access toolbars to be available. These allow access to design views, so would not normally be wanted for a distributed version
Allow Toolbar/Menu Changes	Yes	Specifies whether or not the user can make changes. If cleared, then the user cannot use right-click or the View menu to access design toolbars; this is what you would normally want for a distributed version.

Fig 7.6.2 Discussion of options

Fig 7.6.3 shows (in a minimised window) the effect of applying the options shown in Fig 7.6.1. This screenprint shows what will be seen when the database is first opened. Note the menu bar, the application title and the application icon. The database window is hidden (and not just behind the main menu!). Right-clicking on the menu bar or the menu form has no effect.

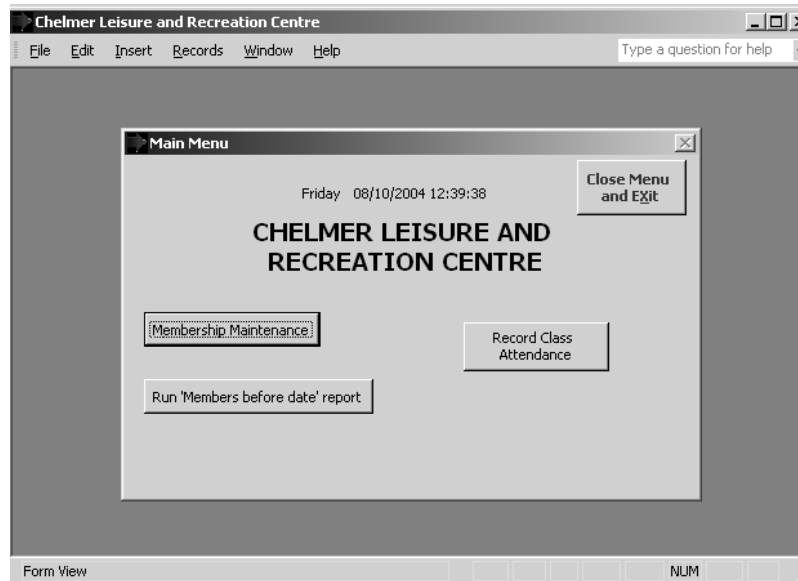


Fig 7.6.3 Effect of options

For further information, check MS Access help.

If wanted, you can also use the same icon for a desktop shortcut for the file.

- Create a shortcut
- Right-click the shortcut to see the properties
- Click the 'change icon' button
- Select the required picture.

Tip: If you remove the menus and have forgotten to make a back-up copy first, you can normally override the Startup options by pressing the SHIFT key down when you open the database. But if the user is also familiar with this, then they can also use this to get at the table and query design. As stated in section 7.5, if the tables are in a back-end database, then the table designs can only be viewed from the front-end, they cannot be changed.

7.6.2 Creating MDE (Microkernel Development Environment) files

VBA is an interpreted language; as you have been using it so far each statement has been compiled and run whenever you have activated the event. This is not efficient for a 'live' system.

With some other languages, such as COBOL, VB, Java, C++, you can create .exe (executable code) files. With VBA, a .mde file is the equivalent of this.

In a .mde file, all code is stored in compiled format, so the code is not available to the user, or to the developer, nor can the forms, reports or code modules be modified. It should also be smaller (as raw code has been removed) and run more efficiently.

To create an MDE version, do the following:

- Create a copy of your code database ('CL Code exe', for example). The conversion of a database to .mde format works on the existing version of the database so Help advises that you make a copy.
 - o Your file format must be for the version of Access used to create the MDE file; if you are using Access 2002 (or 2003) you must convert it to that format first.
 - Use *Tools* → *Database Utilities* → *To Access 2002 file format*.
 - Note that this means that all users must have a version of Access that can read the chosen file format.

- Using the copy with the correct file format, do:
 - o *Tools*→*Database Utilities*→*Make MDE File*
 - o Confirm the name and directory and click on Save.
- Close Access and check the folder where the MDE file should be. You will see a database with a new icon, as shown in Fig 7.6.4. This is the MDE file.

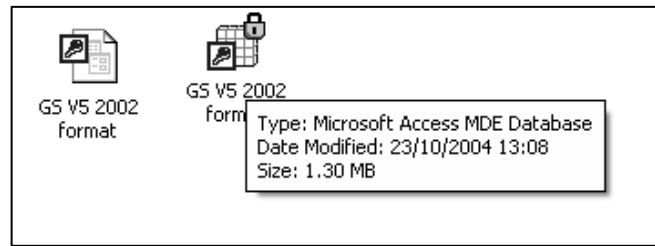


Fig 7.6.4 Icon for normal database and for MDE file

- Open the MDE file.
 - o Try amending forms, reports, macros, data access pages or code modules – the *design* and *new* boxes are greyed out, so that the designs and code cannot even be viewed let alone changed.
 - o But you can change table and query designs.
- You can set Startup options (See section 7.6.1) with an MDE file, but can still use the SHIFT key to over-ride these.
- For further information, look up *mde* in the Access Help Answer Wizard.

7.7 The WITH statement

Access 2002 Help states:

*“The **With** statement lets you specify an object or user-defined type once for an entire series of statements. **With** statements make your procedures run faster and help you avoid repetitive typing”.*

The statement has not been used earlier in this Trainer as those readers who are struggling with VBA may not want an added complication, even though the With statement is straightforward to use. Those of you who can cope with VBA (and who have read this far) may like to know about it, and use it in your own applications. Some VBA Help examples use With.

7.7.1 Using a single With statement

The With statement is useful if you are referencing many properties of an object. Instead of repeating the object name for each property, you can set it just once using With. Fig 7.1.1 shows the code from Fig 3.2.5 as it is in that figure, and rewritten using With.

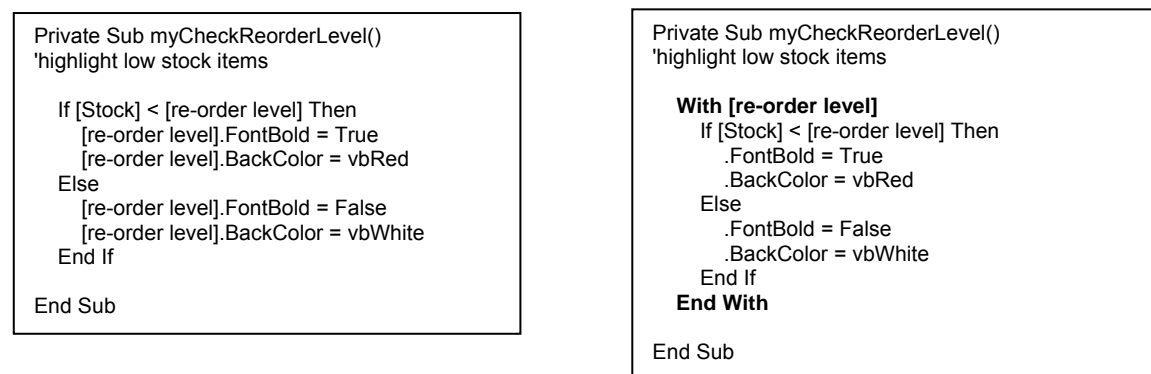


Fig 7.7.1 Comparison of the same code with and without the With statement (original code is in section 3.2.2.2)

Points to note:

- The With block starts with a With <object name> statement and ends with an End With statement.
- You now refer to the object properties by coding the full-stop and then the property name.
 - o The little pop-up prompt box (see Fig 1.1.2) is provided as before.
 - o You can still code <object name>.<property>, where <object name> refers to the object on the With block, as before (a bit pointless if within the With block, but Access does not object and the code appears to run OK). For example, you could still code
[re-order level].BackColor = vbRed
 - o If you want to set a property for another object within the With block then you must specify this in full. for example, if you wanted to set the forecolor on the Stock field to red when the stock level is low (not forgetting to put it back to black when it is OK), then you must code
Stock.ForeColor = vbRed

Further information and advice from Access 2002 Help:

- *“Once a With block is entered, <object> cannot be changed. As result you can’t use a single With statement to affect a number of different objects.”*
- *“In general it’s recommended that you don’t jump into or out of With blocks. If statements in a With block are executed, but either the With or End With statement is not executed, a temporary variable containing a reference to the object remains in memory until you exit the procedure.”*

7.7.2 Using nested With statements

With statements can also be nested within each other. Access 2002 Help states:

“you can nest With statements by placing one With block within another. However, because members of outer With blocks are masked within the inner With blocks, you must provide a fully qualified object reference in an inner With block to any member of an object in an outer With block.”

Look at Fig 7.7.2. This is the code from Fig 3.6.8 rewritten to use With statements, one nested within the other. It’s a bit contrived, as there is no need to use nesting with this example, as two separate blocks could be used instead, however it serves to illustrate the point.

```
Private Sub IstMember_Db1Click(Cancel As Integer)

    With IstMember
        If .Column(5) = True Then
            txtSex = "Male"           'this will be used by IstClass query
        Else
            txtSex = "Female"
        End If

        txtLastname = .Column(1)    'show member lastname on form

        With IstClass
            .Requery                'requery IstClass and...
            .Visible = True         '...make it visible
            'any reference to IstMember in here must be fully qualified
        End With 'for IstClass

    End With 'for IstMember

End Sub
```

Fig 7.7.2 code from Fig 3.6.8 rewritten to show how to nest With statements

7.7.3 Using With statements with parameter objects

With statements can also be used to reference objects passed as parameters. Fig 7.7.3 shows a very simple example of a form where label properties are set in a sub procedure that has the label name passed to it at run-time. This is not a particularly useful example in itself, but it serves to illustrate the principle.

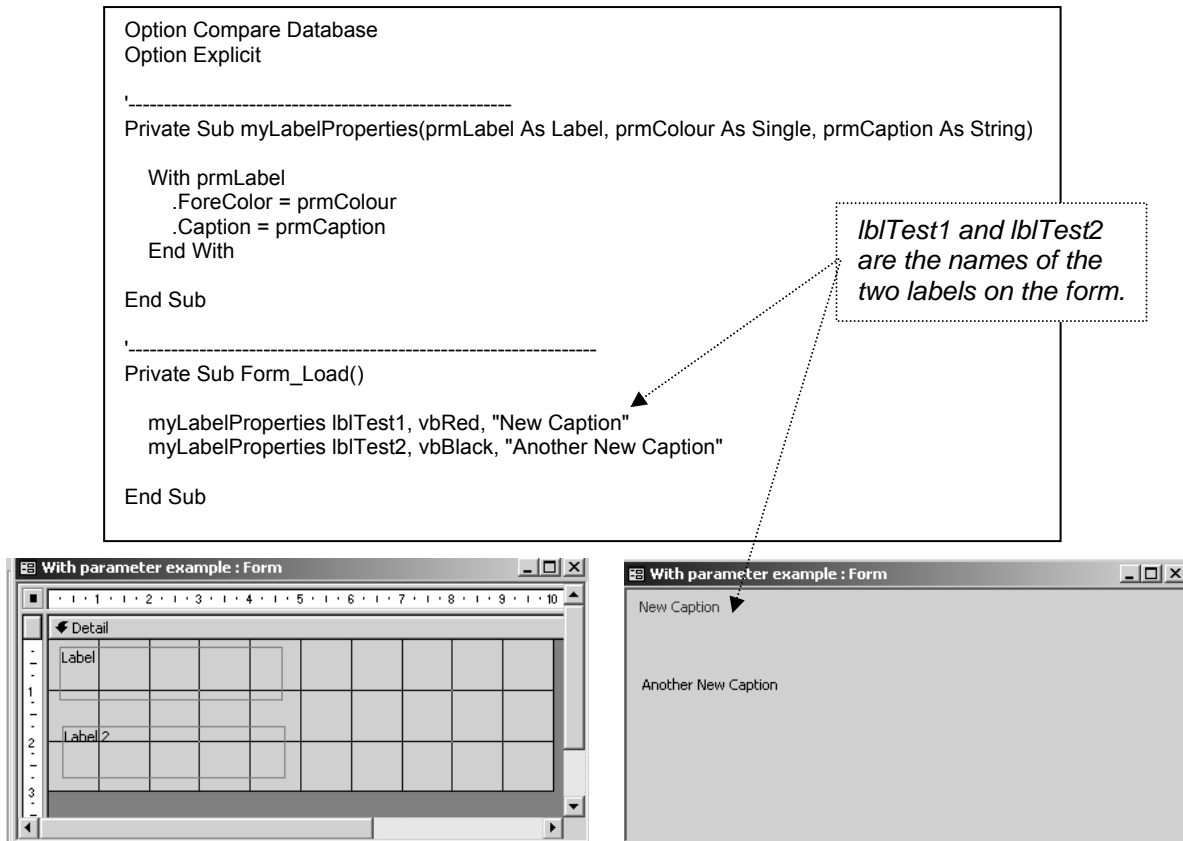


Fig 7.7.3 Example form with code that sets label properties via a sub procedure using With

7.8 Exercises

7.8.1 Member bookings tab on Membership form

Add a new tab to the membership form of Fig 7.2.3, and show all bookings made so far this year for the member.

7.8.2 Record Class Sales

Provide a facility to allow Class Tutors to record sales on a spreadsheet for the application to read and update the Stock Level table.

7.8.3 New Chelmer Application

Open a new database and link it to the back-end tables.

- Create a query and a report.
- Add a facility to display the query result and save it to a spreadsheet if the user so wishes.

This will demonstrate that you can use the same set of tables for more than one front-end application.

7.8.4 Use the With statement

Change your code in Fig 5.3.1 to use the With statement.

PART 8 – WORKED EXAMPLES OF BOOKING PROCEDURES

In this part of the Trainer you will see...

- ...worked examples illustrating use of some of the VBA discussed earlier in this document.
- **Example 1 is for Member Bookings of Courts.**
 - Uses list boxes to show free/booked courts. The query for each list box is based on an outer join query with a table listing all allowable booking times, which selects bookings for a given date. The Court list boxes can be adjusted to show all information for a court (booked and free slots) or free slots only.
 - Double-clicking on a free slot in a Court list box starts the booking process by putting the details in the booking record.
 - Double-clicking on a Member list box allows the user to choose the member for the booking.
 - Double-clicking on a booked slot allows the user to delete the booking.
 - Two methods of checking for simultaneous double-bookings are shown.
- **Example 2 is for Class Bookings of Rooms/Halls.**
 - Allows the user to write several booking records at once simply by specifying the start and end dates. A class runs on the same day each week, at the same time.
 - Shows a method of listing results in weekday (Sun, Mon, Tue, etc) order.
 - Shows how to write temporary records to a table, using embedded SQL (including a method of coping with UK/USA date formats in SQL), then uses this table for an outer join query.
 - The user chooses the class by double-clicking on a list box, then entering the start and end dates for the run of the class, and choosing the room.
 - Deletions can be done one-by-one in a similar manner to that for Example 1.
- **Example 3 shows how to set up a diary page booking form.**
 - Uses a Crosstab query based on an outer join query to select bookings for a required date.
 - A method is demonstrated of specifying a form parameter criterion for the Crosstab query.
 - Conditional Formatting is used to distinguish between booked and free slots on the Booking diary page form.
 - A method is shown of catering for the situation where the Crosstab query does not have all the columns that the form is expecting.
 - The user can then click on slots to make and delete bookings, and the form is refreshed to show the changes.

See <http://www.cse.dmu.ac.uk/~mcspence/Access.htm> for some more example databases, including booking items (cars, hotel rooms, etc) for a period of days.

8.1 Introduction

In this document so far you have seen how to use VBA to improve various aspects of the functionality of the Chelmer Leisure database system: writing private/public procedures; data maintenance; automatic calculations; validation of input data; searching for, and filtering, records; using combo and list boxes; using menus; using form parameters; improving reports. This Part of the Trainer will show how to use some of the techniques shown so far in a further function for the database.

One of the main functions of the Chelmer Leisure and Recreation Centre is the booking of Courts and Classes, but the only booking procedure so far available is that of the very simple booking form in Unit 16 of *McBride*. This form is not user friendly: it does not assist with data entry; it does not show the user the room availability; and it does not prevent double-bookings.

Many students have booking systems for Projects and Placements, so these examples here are to demonstrate some ways that you may find useful if you have to implement such a system, and to illustrate further uses of the VBA that you have seen so far. They are primarily intended for students on year 2 or higher, but keen year 1 students may also find the examples of interest.

The methods illustrated here use a mixture of standard Access point-and-click features and VBA. The solutions are not complete, fully-working solutions, but are partial solutions to demonstrate the ideas, leaving you to finish, or to adapt, them for your own Projects.

The examples here are based on several assumptions, guessed at from information provided in *McBride*:

- Courts are booked by Members only; Sports Halls and Fitness Suite are booked by Classes only (see page 240 of *McBride*).
- All rooms can be booked between 09.00 and 20.00 (9 a.m. to 8 pm) inclusive, on all days of the week, for periods of one hour only.
 - o A system that allowed the user to make several bookings at the same time (on the same booking number) would need another table, being the 'many' end of a one-to-many relationship with the Bookings table. The design of the Chelmer database has not got such a table.

Before you start, add the Public constants shown in Fig 8.1.1 to an Access module. These constants will be used in some of the later examples. Using constants in this way will ensure consistency, and is better practice than using literal (in this case String and Boolean) values in the code.

```
Public Const myConMember = True    'Yes
Public Const myConClass = False   'No
Public Const myconCourt1 = "Court 1"
Public Const myconCourt2 = "Court 2"
Public Const myconCourt3 = "Court 3"
Public Const myconFSuite = "Fitness Suite"
Public Const myconSHall1 = "Sports Hall 1"
Public Const myconSHall2 = "Sports Hall 2"
```

Fig 8.1.1 Public Constants for use in later code.

8.2 Example 1 – Member Bookings for Courts

8.2.1 Create table of booking times.

In order to list all free and booked slots, and also to make it easier for the user to select a valid time, a table of allowable times will be needed. Create a table for all the possible booking times as shown in Fig 8.2.1.

For a system where bookings are to be every 15 (or 20 or 30...) minutes, simply set the table rows as 09:00, 09:15, 09:30, etc.

If breaks (lunch-time, etc) are needed, then simply miss these off.

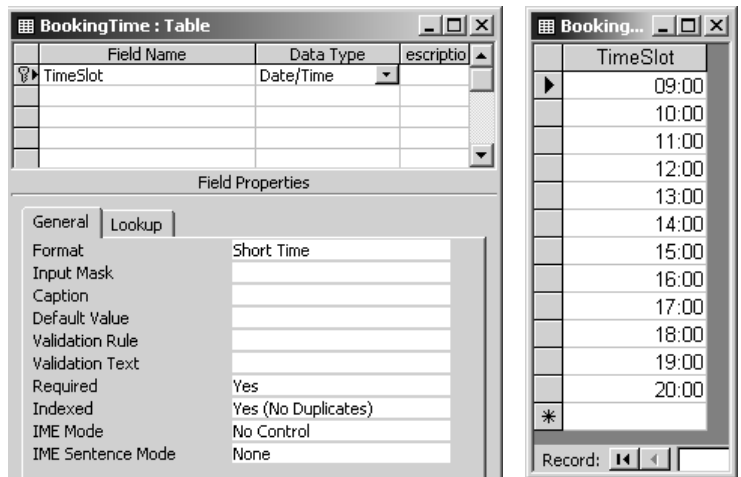


Fig 8.2.1 BookingTime table, in design and datasheet views.

8.2.2 Start the Court Bookings form.

Create a simple wizard form called CourtBookings based on all the fields from the Bookings table, except for the Class No.

Add an unbound text box called txtDate to the form header.

In the Form_Load event code (see section 2.6.1):

```
'open form in 'new record' mode
DoCmd.GoToRecord , , acNewRec
```

This form will be developed into the form that will allow the user to see court availability, choose members, and make/delete Member court bookings.

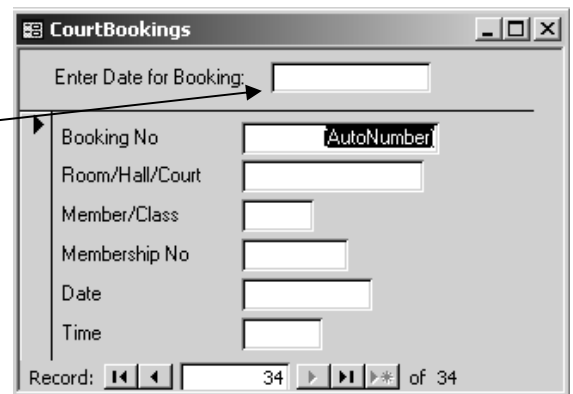


Fig 8.2.2 Initial CourtBookings form

8.2.3 Create Query to see Court availability.

Create a simple inner join (i.e. normal) query to list bookings for Court 1 for the date in the header of the new CourtBookings form. Use a date of 13/5/1996 as this matches data in McBride for the Bookings table. This query will show the booked slots for the date, but not the free slots.

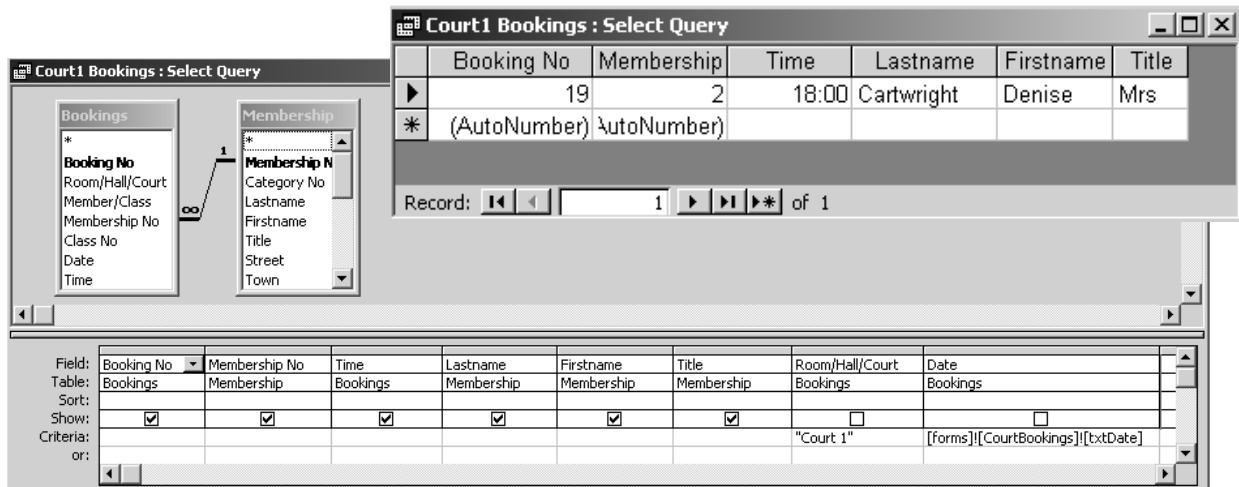


Fig 8.2.3 Inner Join (normal) query to see bookings for the date.

To see the free slots as well, we need an Outer Join query; see Access FAQ 18 *What is an Outer Join?* on <http://www.cse.dmu.ac.uk/~mcspence/Access.htm>. This where the BookingTime table from section 8.2.1 comes in.

Create a second query as shown below, using the BookingTime table and the *Court1 Bookings* query. Create a join between the tables (joining the times) and right-click on the join.

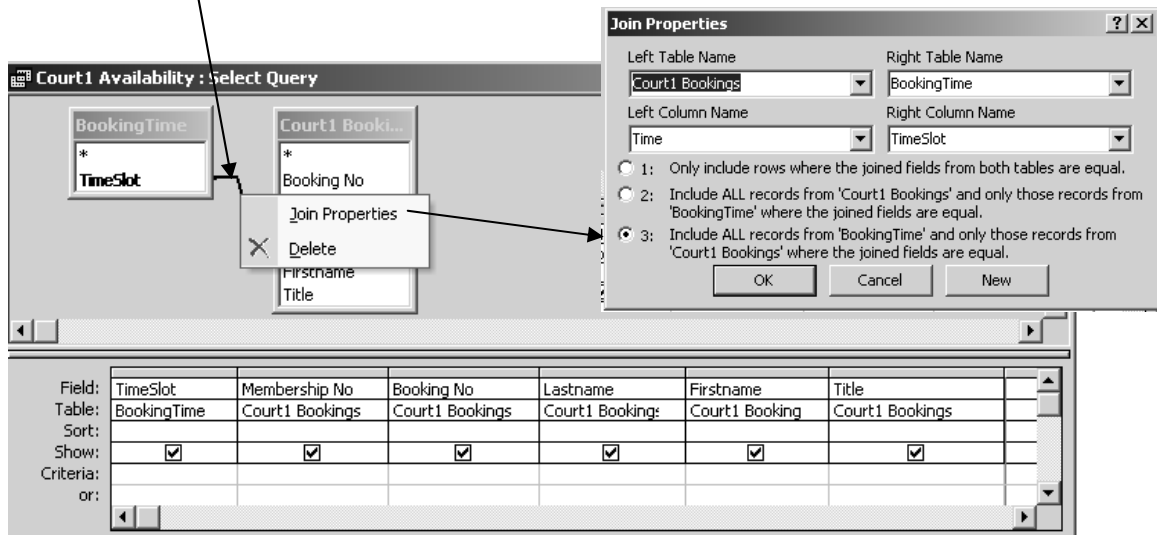


Fig 8.2.4 Creating an Outer Join query

Select Join Properties and choose the option shown; you want to see all booking times, with matches where bookings exist. Note that the line that joins the tables will now have an arrow pointing to the Court1 Bookings query (see Fig 8.2.8). Save the query as *Court1 Availability*.

Now if you run this query (with the CourtBookings form open and a date of 13/5/1996 entered in txtDate) you will see a row for all the timeslots with bookings where they apply. See Fig 8.2.5. Outer joins are extremely useful in cases like this. Have a look at the SQL and compare it with that for a 'normal' query (an inner join is the default).

TimeSlot	Membership	Booking No	Lastname	Firstname	Title
09:00					
10:00					
11:00					
12:00					
13:00					
14:00					
15:00					
16:00					
17:00					
18:00	2	19	Cartwright	Denise	Mrs
19:00					
20:00					

Fig 8.2.5 Result of Outer Join query to list free and available time slots.

8.2.4 Use the Outer Join query for a form list box.

Add a list box to your form and base it on the *Court1 Availability* query. When prompted by the wizard, choose not to save any of the fields; code will be used to put list box contents in form fields in a double-click event for the list box (the wizard does this on a single-click event).

Set field properties as shown

Property	Value
Name	lstCourt1
Control Source	Table/Query
Row Source	Court1 Availability
Column Count	6
Column Heads	Yes
Column Widths	1.3cm;0cm;0cm;2.549cm;0cm;0cm
Bound Column	1
Default Value	
IME Hold	No
IME Mode	No Control
IME Sentence Mode	None
Validation Rule	
Validation Text	
Status Bar Text	
Visible	Yes
Display When	Always
Enabled	No
Locked	No
Multi Select	None
Tab Stop	Yes
Tab Index	6
Left	7.143cm
Top	0.529cm
Width	3.899cm

Fig 8.2.6 Court list box properties.

The query has many fields for the booking details, as they will be needed in subsequent code, but they are not all required to be seen in the list box.

See section 3.6 for information regarding list box properties.

8.2.5 Do more on the CourtBookings form and the Court1 Availability query.

Set various form properties and controls as shown in Fig 8.2.7.

txtDate – used for user to enter date for booking, and by the queries for the court list boxes

lstCourt1 – list box based on query Court1 Availability. Disabled in field property.

Booking table bound fields set to Locked = Yes, BackStyle = Transparent and SpecialEffect = Sunken, as these fields will not be used for user data entry. Move Member/Class to the end of the group and set Visible to No and Default to Yes, as this field will always be for member bookings.

txtCourtParam – textbox that will be used by the Court1 Availability query to select all slots or just the free slots. It will contain the values "All" or "Free". Remove the label and set the Visible property to No. Set the DefaultValue property to "Free".

cmdCourts – non-wizard button that will be used to toggle lstCourt1 (and the other listboxes when added later) between seeing all slots and seeing just the free slots.

Fig 8.2.7 The CourtBookings form so far, in Design View

Change the *Court1 Availability* query to reference txtCourtParam as shown in Fig 8.2.8.

The criterion for the Lastname field is:

Like IIf([forms]![CourtBookings]![txtCourtParam]="All","*",Null)

At run-time, if txtCourtParam = "All" the criterion is set to Like "*" so that all rows are selected. Otherwise the criterion is set to Null, so that only rows where Lastname is Null (the free slots) will be selected.

The IIf function is extremely useful in situations like this where you want to set a variable criterion. Look at VBA Help for more information.

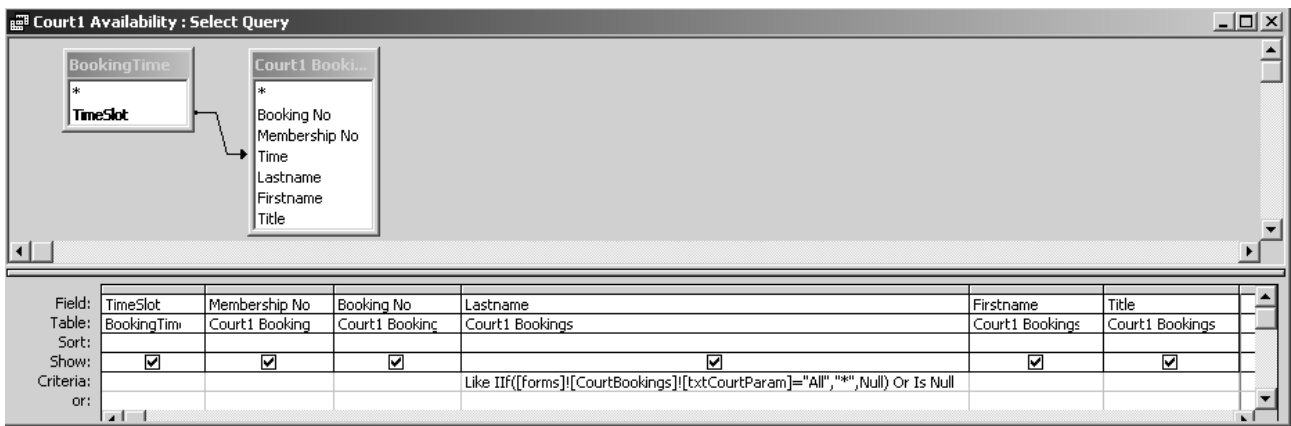


Fig 8.2.8 The query using the value in txtCourtParam to select all slots or just the free slots.

8.2.6 Viewing Court availability.

So far, only one line of VBA has been coded (section 8.2.2) plus the use of the IIf function in section 8.2.5. The form has been set up with a date parameter txtDate and a list box lstCourt1 that uses a query based on the txtDate and the value in txtCourtParam.

Add the code shown in Fig 8.2.10 to your code module for the CourtBookings form. Now you can enter a date in txtDate, see the free slots for the date in lstCourt1, and click on the cmdCourts button to toggle between seeing all the slots and just the free slots.

Add list boxes for Courts 2 and 3, using the method shown in the previous sections for Court 1 and check the code out with these new list boxes. Uncomment the lines in Fig 8.2.10 for lstCourt2 and lstCourt3.

Add a list box for the members with provision for a filter, as described in section 3.6.3. Set the Enabled property to No. This list box will be used later for making bookings. Choose not to save the Membership No.

Your form should now look like that in Fig 8.2.9, and you can enter a date and see availability and bookings for each of the Courts. If you want to exit without making a booking (not that you have any choice as yet), simply close the form.

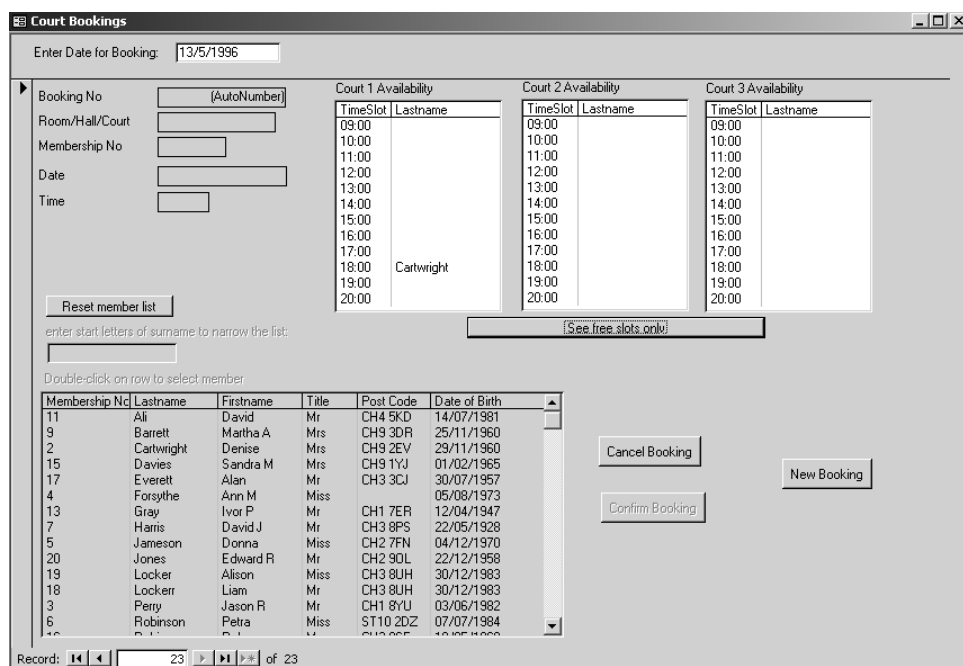


Fig 8.2.9 the CourtBookings form after a booking date has been entered and showing all bookings.

```

Option Compare Database
Option Explicit

'constants for toggling list boxes between Free and All slots
Const myconSeeFree = "See free slots only"
Const myconSeeAll = "See all slots"
Const myconFree = "Free"
Const myconAll = "All"

'-----
Private Sub Form_Load()

'open form in 'new record' mode
DoCmd.GoToRecord , , acNewRec

End Sub

'-----
Private Sub cmdCourts_Click()
'allow toggling between free and all timeslots
'just one button for all list boxes

With cmdCourts
If .Caption = myconSeeAll Then 'seeing just free slots?
txtCourtParam = myconAll 'change to seeing all slots
.Caption = myconSeeFree 'parameter is used by queries 'Courtn availability'
Else
txtCourtParam = myconFree 'change to see free slots only
.Caption = myconSeeAll
End If
myRequeryAll 'requery list boxes to show new contents
End With

End Sub

'-----
Private Sub txtDate_AfterUpdate()
'user has entered date for booking - set up each listbox for the date

myEnableList lstCourt1
myEnableList lstCourt2
myEnableList lstCourt3

End Sub

'-----
Private Sub myEnableList(prmList As ListBox)

'Enable Court listboxes and set button caption etc.
cmdCourts.Enabled = True
cmdCourts.Caption = myconSeeAll
txtCourtParam = myconFree

'requery listbox to show slots for the required date.
With prmList
.Requery
.Enabled = True
End With

End Sub

'-----
Private Sub myRequeryAll()

lstCourt1.Requery 'requery listboxes to show new contents
lstCourt2.Requery
lstCourt3.Requery

End Sub

```

These may be better as Public Constants in an Access module, as they then can also be used for Class bookings See section 8.3.6.

Already coded in section 8.2.2

Separate procedure at end of this coding.

Calls common procedure. Date validation not coded yet. Won't compile until lstCourt2 and lstCourt3 are created.

Common procedure, so it can be used for all court list boxes.

Common procedure, also used when booking made or deleted in later sections. Won't compile until lstCourt2 and lstCourt3 are created.

Fig 8.2.10 Code to use the date in txtDate to enable and use the Court1 list box.

Note how having common procedures can really save you a lot of time. Note also that you can pass various objects (such as list boxes) as parameters; see the prompt list when you type As to choose the datatype.

See section 7.7 for information about the With statement.

A possible test plan for the work done so far (just viewing Court availability) is shown in Fig 8.2.11.

Note that the date of 13/5/1996 is used as this has a Court 1 booking in the data in McBride. Eventually, the date entered in txtDate will be validated, and dates in the past will be rejected, but for now we are just checking the working of the list boxes.

Test No	Data	Reason for Test	Expected result
1	-----	Initial state of form when opened.	<ul style="list-style-type: none"> Booking table fields show as locked, flat and grey. Cannot see Member/Class field. All three Court list boxes and associated buttons are disabled. Member list box and controls disabled. The query parameters are not visible. Courts command button states "See All". Form opened ready for new record.
2	13/5/1996	Enter date for booking (this has one booking for Court 1 in McBride data), see Court list boxes, toggle to see free and all slots for Court 1.	<ul style="list-style-type: none"> All list boxes enabled when date entered. Court 1 list box shows all except booked slot. Court 2 and 3 list boxes show all slots as free. Can click on Courts command button to see contents of Court 1 list box change to show/hide bookings. Caption of button also changes to reflect the current situation.
2	More bookings for various times for all three courts, for 13/5/1996.	Add bookings directly to Bookings table before opening form. Testing operation of all court list boxes.	<ul style="list-style-type: none"> All court list boxes show appropriate free slots (matches the new data in the Bookings table). Clicking on Courts button toggles between seeing all and seeing free slots for each Court. Booked slots all show correct Lastname.
3	-----	Close form	No record added to booking table when user merely views court availability.

Fig 8.2.11 Possible test plan for viewing Court Availability on CourtBookings form. Incremental development and testing is a useful technique for complex functions.

8.2.7 Making a Member booking for a Court.

Now we are almost able to make these bookings.

Bookings will be made by the user...

- ...entering the required date
- ...double-clicking on a free slot in the required Court list box
- ...double-clicking on the required row in the Member list box.

See fig 8.2.12 for how the form will look when a booking is made, and Fig 8.2.13 for the new code to be added to the CourtBookings form.

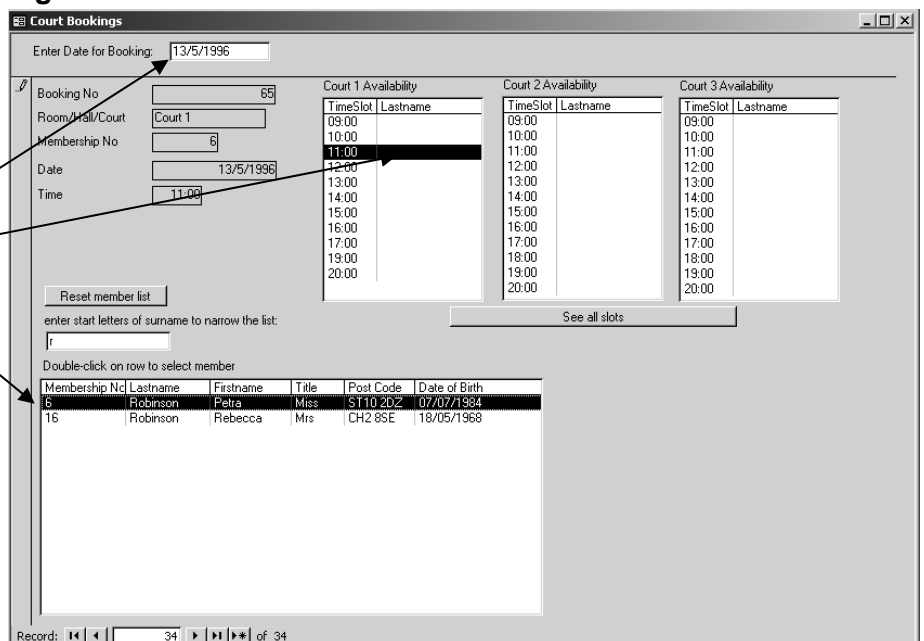


Fig 8.2.12 making a Member Booking for a Court

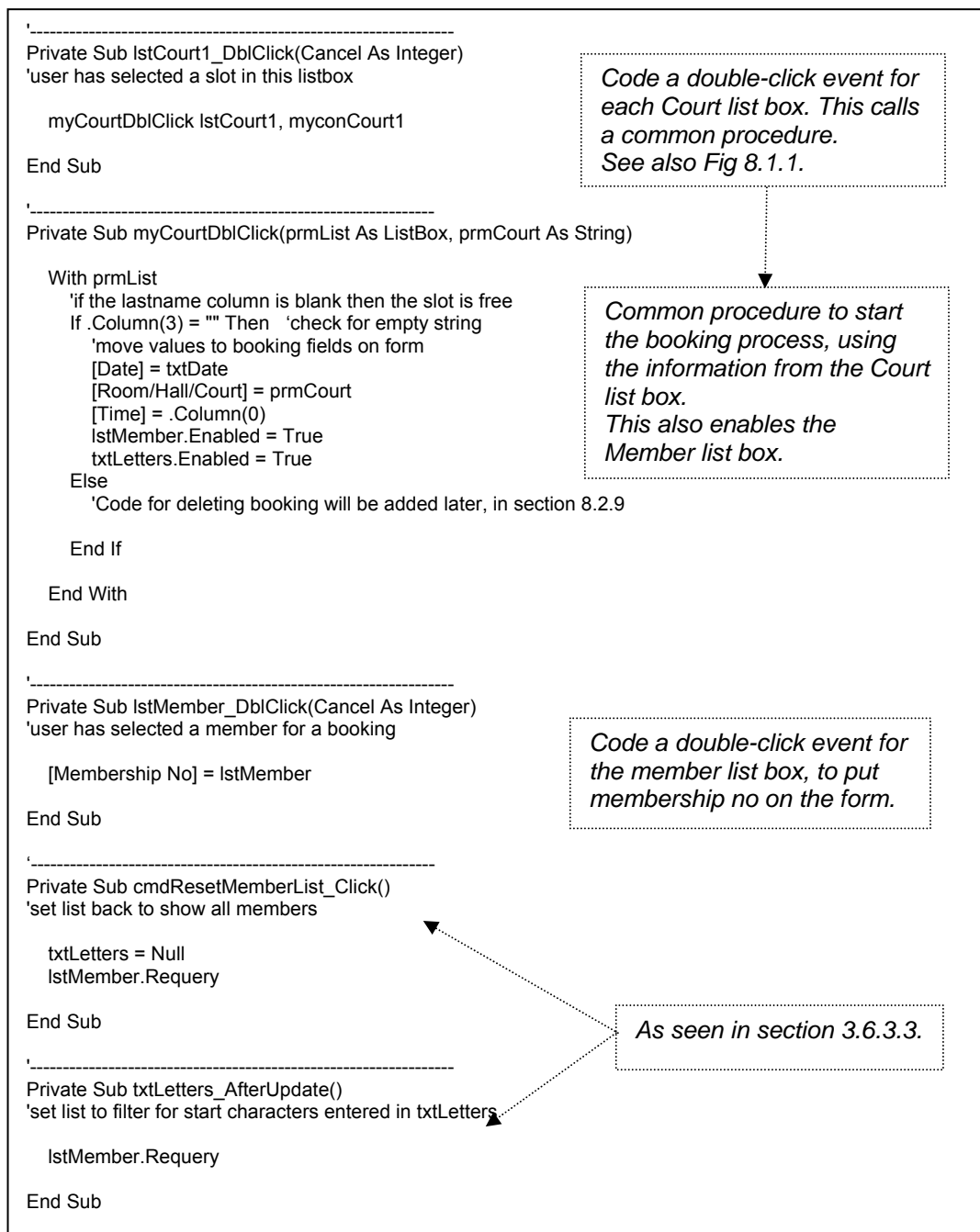


Fig 8.2.13 Code to make Member Bookings for Courts.

See section 3.6 for information regarding list box properties, including the Column property.

The user can now make Court bookings for Members by following the simple procedure shown by Fig 8.2.12. The form is bound to the Bookings table so you can create a wizard save button (give it the label *Confirm Booking* and the name `cmdConfirm`) to save a booking. Make the button disabled for each new booking, enable it when the user double-clicks on the member list box and disable it (moving the focus away first, perhaps to the date in the header or to a *New Booking* button) in the *Confirm Booking* button click event; this should ensure that it is only enabled when the full set of data is in the form fields and that it cannot be clicked twice. See section 8.2.8.1.

The booking number is visible on the form so that the user can give this information to the member for reference if required. If not, then it could be set invisible.

8.2.8 Preventing double-bookings.

The form opens showing the availability at that instant of time. However, in a multi-user system it could be quite possible that another user is looking for Court availability for the same day, and opens the Court Bookings form at the same time. Thus, two users are simultaneously looking at the same information, and could well make a booking for the same room on the same day and at the same time. At present there is nothing to stop this happening. The standard Access record and form locking facilities are not really appropriate here, as...

- ...each booking is for a new record, so record locking won't stop double-bookings.
- ...locking a form means only one user can make a booking (or even simply check availability) at a time; this is not good practice, especially in a system where booking activity may be high.

So, you will need to check for this situation yourself, via code, and take appropriate action if the required booking has been made in the few seconds/minutes since the Court Bookings form was opened or last refreshed.

It may also be useful to provide the user with a report listing double-bookings (if any) where the booking date is greater than or equal to today; this will highlight any possible cases where the code hasn't worked, or some other situation has occurred that has allowed a double-booking.

8.2.8.1 Using DCount.

In the `cmdConfirm` click event (see end of section 8.2.7), add code to count up the number of bookings for the required court, date and time. If the count is zero, then the slot is still free and you can go ahead and save the booking. If the count is not zero, then there is a booking already for this slot which has been made in the past few seconds/minutes (if the code is all working correctly!).

```

Private Sub cmdConfirm_Click()
'wizard code to Save record
'plus changes to check for simultaneous double-bookings shown in bold
On Error GoTo Err_cmdConfirm_Click

Dim strSQL As String
Dim intCount As Integer
strSQL = "[Room/Hall/Court] = " & [Room/Hall/Court] & "" _
    & " AND [Date] = #" & [Date] & "#" _
    & " AND [Time] = #" & [Time] & "#"
intCount = DCount("[Booking No]", "Bookings", strSQL)

If intCount = 0 Then 'slot is free
DoCmd.DoMenuItem acFormBar, acRecordsMenu, acSaveRecord, , acMenuVer70
cmdNew.SetFocus 'move focus away so can...
cmdConfirm.Enabled = False '...disable confirm button
myRequeryAll 'requery Court list boxes – own proc in Fig 8.2.10
lstCourt1.Enabled = False 'stop user from clicking on boxes again and attempting to save...
lstCourt2.Enabled = False '...as this will change the booking in the bound fields
lstCourt3.Enabled = False
lstMember.Enabled = False
myDisplayInfoMessage "Booking confirmed"
Else 'slot has just been booked
myDisplayWarningMessage "This slot has just been booked by another member" & vbCrLf _
    & "Please choose another slot or cancel"
End If

Exit_cmdConfirm_Click:
Exit Sub

Err_cmdConfirm_Click:
MsgBox Err.Description
Resume Exit_cmdConfirm_Click

End Sub

```

These two items are in variables so that you can check them in the Debugger.

Wizard line to save the record.

See section 8.2.10 for a possible method of allowing the user to make several bookings for the same date.

Fig 8.2.14 Code (in bold) showing how to use DCount to check for a double-booking

To test a double-booking choose the date, court, time and member, then go to the Booking table and add a booking for the same date, court and time, close the Bookings table, go back to the Bookings form and then click on `cmdConfirm`.

8.2.8.2 Setting a unique index.

It is also possible to get Access to do the check for you. Add a new index called BookingsSlot to the Bookings table, as shown in Fig 8.2.15.

Set the value for the Unique property to Yes (the default is No).

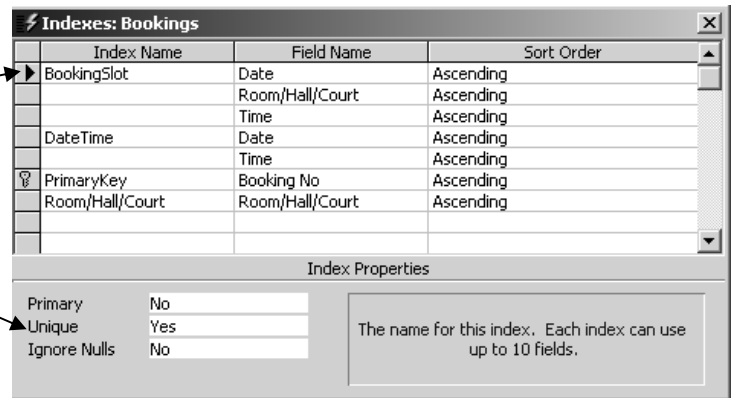


Fig 8.2.15 Setting a unique index for the booking slot.

Now, if you use just the wizard code to save a record, Access will check for a double booking. However, the message that Access comes back with is not that user-friendly; see Fig 8.2.16.

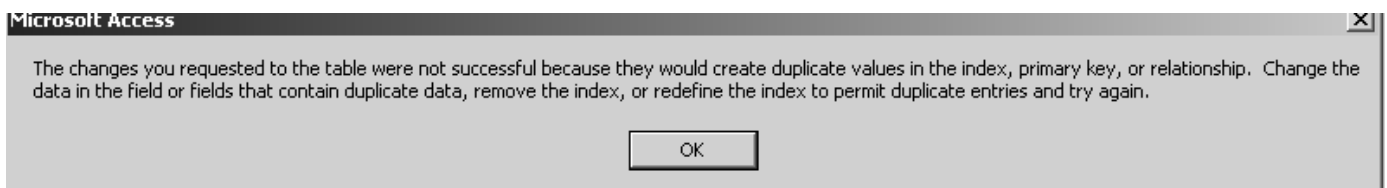


Fig 8.2.16 Access standard message for a duplicate key error

Fig 8.2.17 Shows how you can add to the wizard code to check for this message and replace it with one of your own. This is probably the best method to check for simultaneous double-bookings as Access code is likely to be more accurate than yours or mine!

```

Private Sub cmdConfirm_Click()
'wizard code to Save record
'plus code to intercept Access error for double-booking
On Error GoTo Err_cmdConfirm_Click

DoCmd.DoMenuItem acFormBar, acRecordsMenu, acSaveRecord, , acMenuVer70
cmdNew.SetFocus 'won't get here if save fails
cmdConfirm.Enabled = False 'disable confirm button
myRequeryAll 'requery Court list boxes – own proc in Fig 8.2.10
lstCourt1.Enabled = False 'stop user from clicking on boxes again and attempting to save...
lstCourt2.Enabled = False '...as this will change the booking in the bound fields
lstCourt3.Enabled = False
lstMember.Enabled = False
myDisplayInfoMessage "Booking confirmed"

Exit_cmdConfirm_Click:
Exit Sub

Err_cmdConfirm_Click:
'MsgBox Err 'code this first to see the error number then code the rest
If Err = 3022 Then 'duplicate key error
myDisplayWarningMessage "This slot has just been booked by another member" & vbCrLf _
& "Please choose another slot or cancel"

Else
MsgBox Err.Description
End If
Resume Exit_cmdConfirm_Click

End Sub

```

This code is the same as in Fig 8.2.14.

Test first with just this MsgBox statement, to see what the error number is. Then remove it (or comment it out, as here).

Fig 8.2.17 Code (in bold) showing how to intercept and replace Access duplicate key error message.

See Part 2 of the Further VBA Trainer for more information about error-handling.

8.2.9 Deleting a Member Booking

This is now easy.

In Fig 8.2.13, in the procedure `myCourtDbfClick` where there is a comment about deleting a row, replace the comment with the code shown in Fig 8.2.18.

```
'this is a booked slot
'confirm delete - show member details (number and name)
strDelete = "Delete booking for Member No: " & .Column(1) & vbCrLf _
    & .Column(4) & " " & .Column(3) & " (" & .Column(5) & ")"
If myYesNoQuestion(strDelete) = vbNo Then
    myDisplayInfoMessage "Booking kept on file"
Else
    strSQL = "DELETE * FROM Bookings where [Booking No] = " & .Column(2)
    DoCmd.SetWarnings False 'suppress Access messages
    DoCmd.RunSQL strSQL
    DoCmd.SetWarnings True
    myDisplayInfoMessage "Booking deleted"
    myRequeryAll 'own procedure that requeries each list box – see Fig 8.2.10.
End If
```

Fig 8.2.18 Code to Delete a Booking

See section 6.6 for further information about using embedded SQL to delete a row from a table.

Fig 8.2.19 shows the form and the 'Delete booking?' question, which uses some of the hidden list box fields to make a more useful message.

The screenshot shows the 'Court Bookings' form with the following components:

- Form Fields:** Enter Date for Booking (13/5/1996), Booking No (AutoNumber), Room/Hall/Court, Membership No, Date, Time, and a 'Reset member list' button.
- Availability Tables:**
 - Court 1 Availability:**

TimeSlot	Lastname
09:00	
10:00	
11:00	Robinson
12:00	
13:00	
14:00	
15:00	
16:00	
17:00	
18:00	Cartwright
19:00	
20:00	
 - Court 2 Availability:**

TimeSlot	Lastname
09:00	
10:00	
11:00	Davies
12:00	
13:00	
14:00	
15:00	
16:00	
17:00	
18:00	
19:00	
20:00	
 - Court 3 Availability:**

TimeSlot	Lastname
09:00	
10:00	Jameson
11:00	Gray
12:00	
13:00	
14:00	
15:00	
16:00	
17:00	
18:00	
19:00	
20:00	
- Member List:** A table with columns: Membership No, Lastname, Firstname, Title, Post Code, Date of Birth. The 6th record is highlighted: 6, Robinson, Petra, Miss, ST10 2DZ, 07/07/1984.
- Dialog Box:** 'Chelmer Leisure and Recreation Centre' with a question mark icon and the text: 'Delete booking for Member No: 6 Petra Robinson (Miss)'. It has 'Yes' and 'No' buttons.

Fig 8.2.19 Deleting a Member Booking

8.2.10 Finally...

There are still a number of things that need to be done to make this procedure work correctly in all situations, and these are left for you to do (design, code and test), as very similar situations have been covered already in this Trainer. These are the sort of things that you would have to do in a Project. Some suggestions are shown in Fig 8.2.20.

Item	Comments
Validate the date in txtDate (see section 3.3.1).	<ul style="list-style-type: none"> • Check that the value is not Null, is a valid date, and is equal to or after today. You could allow the user to view past bookings, but not make/delete past bookings. • You could also provide a button for the user to use a calendar control to select the date, putting the chosen date into txtDate. See section 5.6. • It could also be useful to put the day of week in words for txtDate on the form. See section 4.2.1. • Check that the Centre is open on this date. One way is to have a new ClosureDate table and use DCount Or DLookup to see if the given date is in the table.
A <i>Cancel Booking</i> (wizard Undo) button could be useful if the user wants to exit without saving.	See the end of section 2.5 and exercise 2.7.2.
A <i>New Booking</i> button (wizard New Record) is useful to allow the user to make several bookings at a time (no need to close and reopen the form).	<ul style="list-style-type: none"> • Make the button enabled/visible only when the current booking has been saved/deleted/cancelled. • Use the form Current event to set default settings for each new booking. • Set the Focus to the date at the top of the form.
Check that the booking time is possible for bookings for today.	<p>If the date for the booking or deletion is today's date, then check to see if the time on which the user has clicked can be booked/deleted or not, and take appropriate action if it cannot (message and cancel, for example).</p> <p>The following line should do the check: If CDate(txtDate) = Date And CDate(.Column(0)) < Time Then 'cannot book...</p> <p>See also VBA FAQ 16 <i>Why don't the Date and Time functions work in my module?</i> on http://www.cse.dmu.ac.uk/~mcspence/Access.htm</p> <p>You could code this check in a public function that you then use to determine the criterion for the time in the Court availability queries, or in the myCourtDbClick procedure.</p>
Ask an 'Are You Sure?' question before confirming and deleting bookings.	<ul style="list-style-type: none"> • Ask the question in the appropriate button Click event. • Save/Delete if the user replies Yes, and requery the three Court List boxes so that the change is reflected on the form. • Otherwise cancel the changes (use Undo, or call the <i>Cancel Booking</i> button code).
Trap unsaved changes and give the user a chance to confirm or cancel a booking.	<ul style="list-style-type: none"> • See section 2.5.2. • Remove the X in the top right-hand corner of the form and add a wizard Close Form button.
Tidy the form up; I have concentrated so far on functionality rather than what the form looks like. Some things you may like to consider are listed here.	<ul style="list-style-type: none"> • Add suitable headings (corporate and form). See section 2.2.1. • Show today's date and time. See section 4.2.1. • Perhaps add help for the user to show how to fill in the form. • Remove record selectors, max/min buttons, navigation bar, etc. • Change sizes of fields so that they are appropriate to the data. • And there will be others I haven't thought of.

Fig 8.2.20 Suggestions for improvements to the Court Booking facility.

A possible test plan is shown in Fig 8.2.21.

Test No	Data	Reason for Test	Expected result
1	<i>Selection of invalid and valid booking dates.</i>	<i>Check validations. See section 3.3.5 for some ideas.</i>	<ul style="list-style-type: none"> • <i>Messages as appropriate for errors, and user cannot move on.</i> • <i>Valid date accepted, user can move on and Court list boxes are enabled.</i>
2	Booking Date = date in the future, which currently has no bookings at all	Can a booking be made for each Court? Make several bookings for each Court for that day, for several members. Check operation of function.	<ul style="list-style-type: none"> • All three Courts are fully available for that day. • Can make bookings for each Court. • Each new booking is reflected accurately in the Court list box. • <i>Form is refreshed after each booking to ensure previous data is removed.</i> • <i>Cannot save a record without a Membership No.</i> • Bookings table shows all new bookings correctly recorded. • <i>Buttons are enabled/disabled appropriately.</i>
3	Booking Date = as above, but now has the bookings from test 2.	Can bookings be deleted from each Court? Delete all the bookings made in test 1.	<ul style="list-style-type: none"> • All three courts correctly show bookings from test 2. • Can delete each booking • Court list boxes are refreshed accurately. • All deleted bookings are removed from Bookings table. • <i>Buttons are enabled/disabled appropriately.</i>
4	Booking date = as above.	Alternate making and deleting bookings.	All works correctly. <i>Buttons are enabled/disabled appropriately.</i>
5	<i>Booking date = system date.</i>	<i>Can the user make bookings after the current time? Click on slots that are:</i> <ul style="list-style-type: none"> • <i>Before now</i> • <i>The same as the current hour.</i> • <i>After the current hour.</i> 	<ul style="list-style-type: none"> • <i>Cannot book.</i> • <i>Cannot book – or could code to allow bookings in first 5-10 mins.</i> • <i>Booking OK.</i>
6	-----	<i>Can user start a booking then cancel it?</i>	<ul style="list-style-type: none"> • <i>Can use Cancel button to cancel booking.</i> • <i>Fields on form are cleared.</i> • <i>No record is written to Bookings table.</i>
7	-----	Check confirm booking button.	<ul style="list-style-type: none"> • Yes – booking saved and list boxes refreshed. • No – booking cancelled/ignored (depending on how you have handled this)
8	-----	Check multi-user double-booking situation. If possible, also test on a common server area with a colleague on another machine.	<ul style="list-style-type: none"> • Start a booking. • Add booking for same court, date and time directly into the booking table. • Complete the booking. Confirm. • Message 'already booked' and possible cancel. • User can cancel or book for another date/time.
9	-----	<i>Start a booking and move to new booking or close without saving.</i>	<i>Situation is trapped and user is given option to save or cancel. Both options work correctly in each situation.</i>

Fig 8.2.21 Possible test plan for making and deleting bookings
Items in italics are those you will have to code as extras to the code demonstrated so far.
This is not an exhaustive list.

8.3 Example 2 – Class Bookings for Rooms/Halls

These bookings could be done in exactly the same way as for Member Court Bookings; however, as Classes typically run each week on the same day at the same time for a number of weeks, a different approach has been taken here. This could be a more appropriate method for the user and also allows different features to be demonstrated.

The approach taken is...

- ...user selects a class from a list.
- ...user enters dates between which the class is to take place. The Class Day and Time are taken from the Class details.
- ...room/hall availability for the weeks of the dates given is displayed for information, and to allow deletion of a class.
- ...a series of class bookings are written to the Booking table, using embedded SQL.

8.3.1 Start the ClassBookings form

The first thing that needs to be done for this form is to add a list box based on the Classes table. A useful order could be the Class Activity then the Class Day, but the Class Day will be sorted in alphabetical order not day of week order.

It's easy to sort in day of week order, but this requires another table, to give a numeric ID to each day name. See Fig 8.3.1.

The numbers given to each weekday are the same as those used by Access for the constants vbMonday etc, as returned by the Weekday function.

(If you had a table with actual dates, and wanted to list them in weekday order, then you could simply use the Weekday function to get the weekday number for each date).

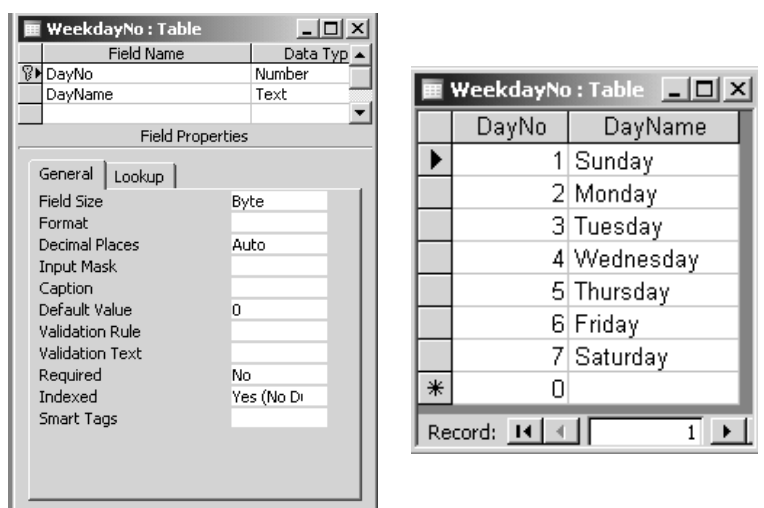


Fig 8.3.1 The WeekdayNo table

Now you can create a query to join the Classes and the WeekdayNo tables, to list the details in an appropriate order, as shown in Fig 8.3.2.

You will need to create the join between the two tables; you do this in the query design view.

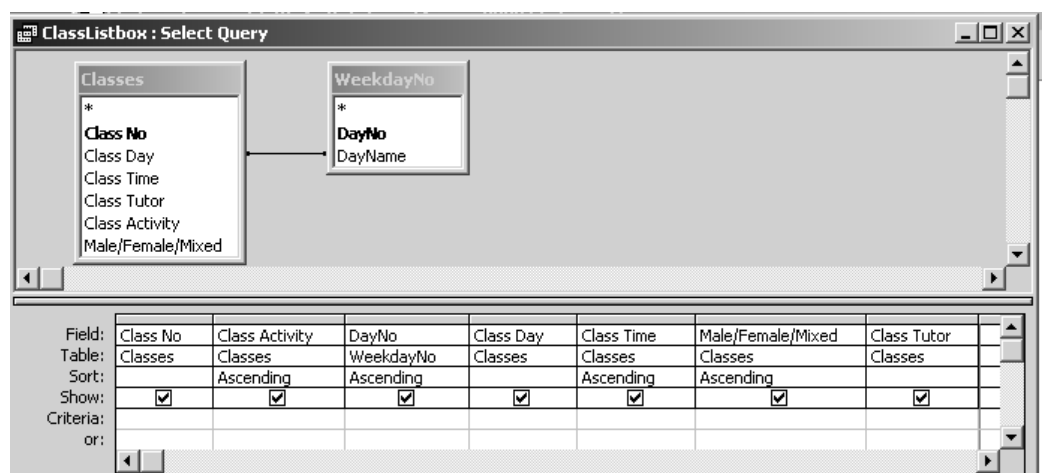


Fig 8.3.2 The Classes list box query, to order by day of week.

Now create an unbound form, and add a list box `lstClass` to it based on the `ClassListBox` query from Fig 8.3.2. set the width for the `Day No` field in `lstClass` to zero, to hide the field on the form.

It would be useful to provide filters for `lstClass` to select elements within the `Class Activity` and for `Male/Female/Mixed`, so add the following:

- To your form (see Fig 8.3.3)
 - o A textbox called `txtActivity`
 - o A combo box called `cboMFM` with `LimitToList = Yes`. Set the `DefaultValue` property to "All".
 - o A non-wizard command button `cmdReset` to reset the list box.
 - o These filters will work in exactly the same way as you have seen already in sections 3.6 and 8.2. You should know to create and code for them by now.
- Two criteria to your `ClassListBox` query:
 - o For the `ClassActivity` column:
Like "*" & [forms]![ClassBookings]![txtActivity] & "*"
 - o This will select all `Class Activities` where the text in `txtActivity` occurs anywhere in the field.
 - o For the `Male/Female/Mixed` column:
Like If([forms]![ClassBookings]![cboMFM]="All", "*", [forms]![ClassBookings]![cboMFM])
 - o This will create a criterion of 'Like "*" ' if the combo box contains "All", and 'Like "Male" ' if the combo box contains (for example) "Male". See also section 8.2.5.

Your form should now look like that shown in Fig 8.3.3.

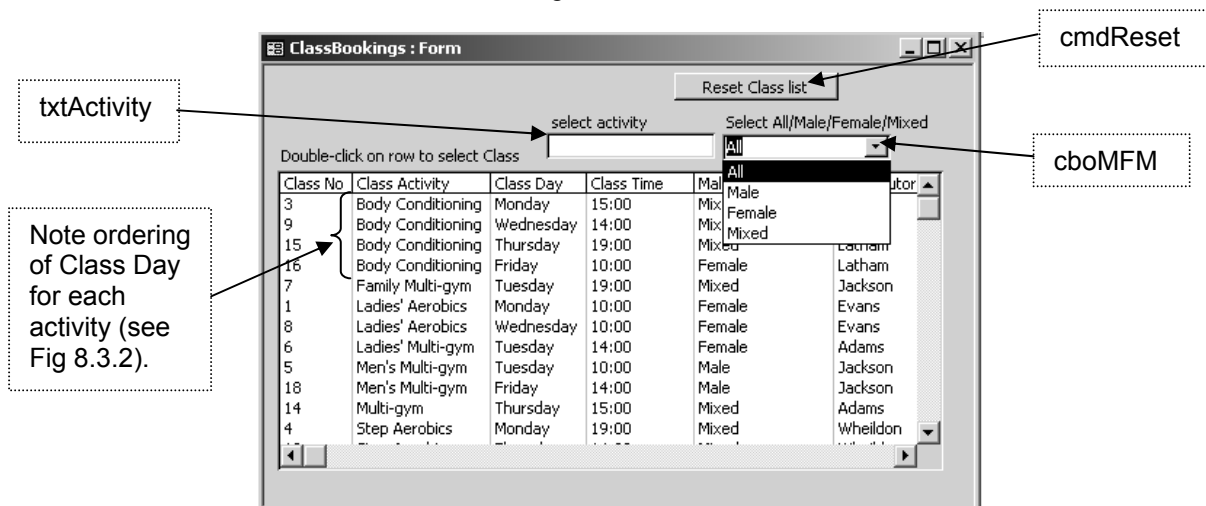


Fig 8.3.3 The `ClassBookings` form with the class list box.

8.3.2 Selecting a class

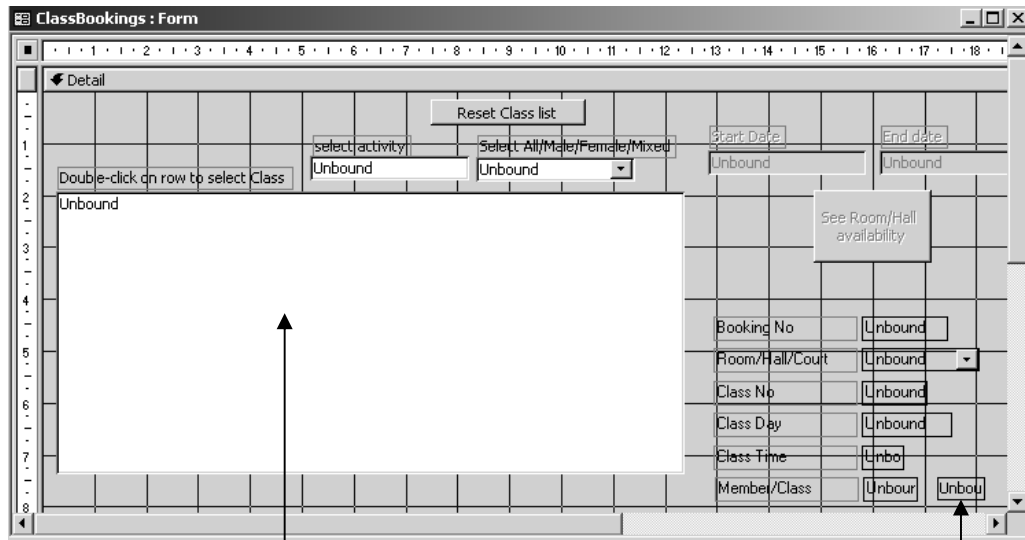
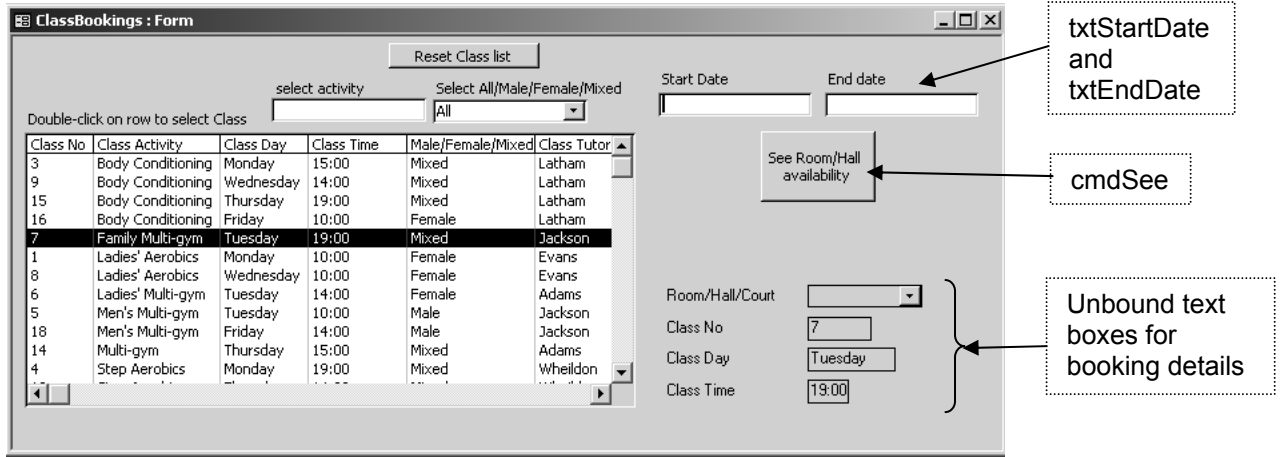
Add a couple of textboxes (`txtStartDate` and `txtEndDate`) for the user to use for the dates between which the class is to be booked. These will be disabled initially but enabled (and with `txtStartDate` given the Focus) when the user double-clicks on the class list box to choose a class.

Add a *See Room/Hall Availability* button `cmdSee`, being a non-wizard button with no code as yet.

Add also some text boxes to record the details of the selected class. Make the `Room/Hall/Court` field a Value List combo box with the room names, with `LimitToList = Yes`. This form is not bound to any table or query; these fields will be used later in embedded SQL.

See Fig 8.3.4 for the form so far. Note that the `Visible` property for some of the fields is set to `No`, as the fields are not required for the user to see.

See Fig 8.3.5 for the code to add for a double-click event for the `Class list box` `lstClass`. You should now be able to select the required class and see the details in the unbound text boxes.



lstClass – based on ClassListBox query, but does not show all the fields (column width set to zero for hidden columns).

txtDayNo – used to store the day number from the ClassListBox query, for use in section 8.3.6.

Fig 8.3.4 The form so far, with the Class list box, date fields and fields for values.

```
Private Sub lstClass_DblClick(Cancel As Integer)
'put chosen class values in fields on the form.

With lstClass
    [Class No] = .Column(0)
    [Class Day] = .Column(3)
    [Class Time] = .Column(4)
    txtDayNo = .Column(2)
End With

'enable date fields ready for user entry
txtStartDate.Enabled = True
txtStartDate.SetFocus
txtEndDate.Enabled = True
cmdSee.Enabled = True

End Sub
```

Note that these column numbers refer to the corresponding value in the list box. You may need to adjust these to match how you have created your list box.

Button enabling is here for now. But code later will fail if user does not enter any dates – see section 8.3.6.

Fig 8.3.5 Code when user double-clicks on the Class list box

8.3.3 Creating a BookingDate table.

We know that the class runs on a specified date at a specified time. What the user wants to see now is the availability for the weeks for which they want the class to run. For example, if the class runs on a Wednesday and the user enters the dates 25th May 2005 and 29th June 2005 (both dates are Wednesdays) then they want to see if the class is booked for all Wednesdays in that range at the required time.

An outer join with a BookingDate table can be used to do this, in an exactly similar manner to that used in the previous section for the BookingTime table. But it was easy to create the BookingTime table as the times for each day's bookings were known. The contents of the BookingDate table will vary from booking to booking.

Do the following:

- Create a BookingDate table as shown in Fig 8.3.6.
- Create an Access module with the code shown in Fig 8.3.7. This code will take two given dates and create rows in the BookingDate table for each week between the two dates.
- Create a Click event for the cmdSee button and add the following line:

```
myCreateDateTable CDate(txtStartDate), CDate(txtEndDate)
```

Now you can double-click on the Class list box to select a class, enter start and end dates, click on the cmdSee button and see the required dates in the BookingDate table. These dates will vary for each pair of start/end dates.

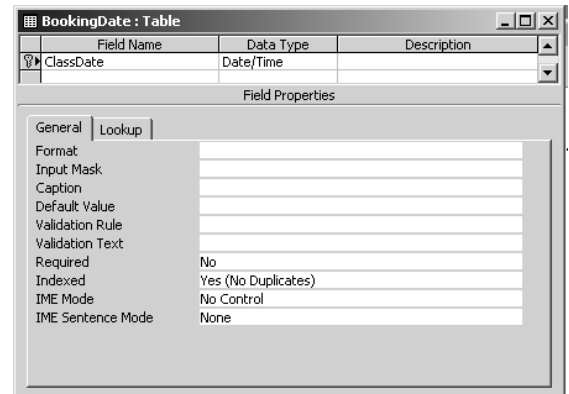


Fig 8.3.6 The BookingDate table.

```
Option Compare Database
Option Explicit

'-----
Public Sub myCreateDateTable(prmStartDate As Date, prmEndDate As Date)
'add rows to the BookingDate table to correspond with the dates the user requires

Dim dtDate As Date
Dim strSQL As String

'first delete any previous rows from the table
DoCmd.SetWarnings False 'suppress Access messages
strSQL = "DELETE * FROM BookingDate"
DoCmd.RunSQL strSQL

'starting from the start date, add a row for each week
dtDate = prmStartDate
Do Until dtDate > prmEndDate 'stop after the end date
    strSQL = "INSERT INTO BookingDate VALUES(#" & myUSADate(dtDate) & "#)"
    DoCmd.RunSQL strSQL
    dtDate = dtDate + 7 'add 7 days
Loop

DoCmd.SetWarnings True 'reinstate Access messages

End Sub

'-----
Public Function myUSADate(prmUKDate As Date) As Date
'Convert date from dd/mm/yyyy (UK) format to mm/dd/yyyy (USA) format
'dates in SQL must be in USA format

Dim strDate As String

    strDate = Month(prmUKDate) & "/" & Day(prmUKDate) & "/" & Year(prmUKDate)
    myUSADate = CDate(strDate)

End Function
```

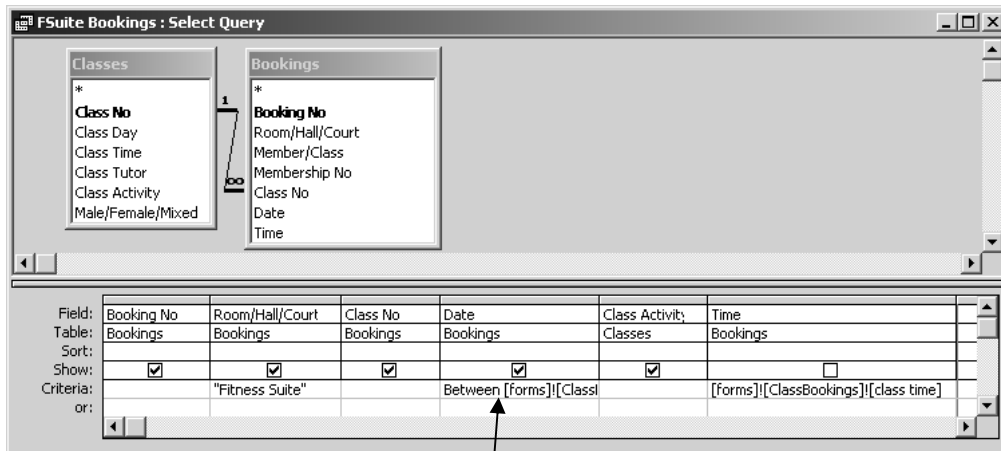
See VBA FAQ 15 [Why does a calculated date give the wrong result in an SQL statement?](http://www.cse.dmu.ac.uk/~mcspence/Access.htm) on <http://www.cse.dmu.ac.uk/~mcspence/Access.htm>

Fig 8.3.7 Code to add dates to the BookingDate table

8.3.4 Seeing Class availability

Now that we have the BookingDate table, with the dates required for the bookings, we can create queries similar to those in section 8.2.3 to see the class availability for those dates at that time.

The two queries for the Fitness Suite are shown below. Queries for the two Sports Halls can be created in exactly the same way.



Between [forms]![ClassBookings]![txtStartDate] And [forms]![ClassBookings]![txtEndDate]

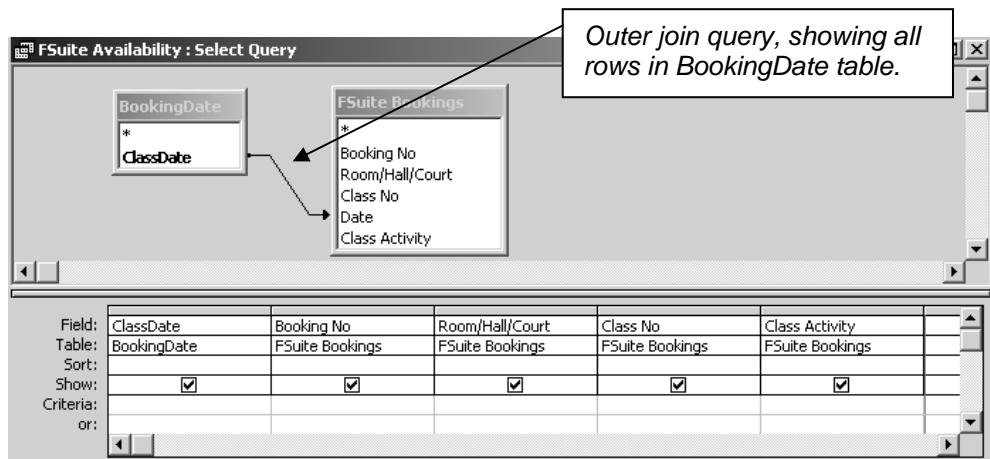


Fig 8.3.8 Queries to show class availability for the chosen dates and time.

You should now be able create list boxes (lstFSuite, lstSHall1, lstSHall2) for each of the rooms, just as in section 8.2.4. Use the queries shown in Fig 8.3.8, and select all the columns, but only show the columns for the Date and the Class No. Set the list boxes to invisible initially, as they will pick up previous values in the BookingDate table when the form is loaded.

Add the code shown in Fig 8.3.9 to the click event for cmdSee.

```
Private Sub cmdSee_Click()
'Create the date list in the BookingDate table

myCreateDateTable CDate(txtStartDate), CDate(txtEndDate)
lstFSuite.Requery 'requery each room availability
lstSHall1.Requery
lstSHall2.Requery
lstFSuite.Visible = True 'make the list boxes visible
lstSHall1.Visible = True
lstSHall2.Visible = True
[Room/Hall/Court].Locked = False
[Room/Hall/Court].SetFocus
[Room/Hall/Court].Dropdown 'user to select room from the list
End Sub
```

Public procedure from Fig 8.3.7.

See section 3.6.

Fig 8.3.9 Code to create dates and show room availability.

Now, if the user selects a class, enters two dates then clicks on `cmdSee`, the form should look like that in Fig 8.3.10. The user now has to select the room from the drop down list.

The screenshot shows the 'ClassBookings : Form' window. At the top, there is a 'Reset Class list' button. Below it are two text boxes for 'Start Date' (9/5/1996) and 'End date' (30/5/1996). A 'select activity' dropdown is set to 'All', and a 'Select All/Male/Female/Mixed' dropdown is also set to 'All'. A 'Double-click on row to select Class' instruction is present above a table of class details. The table has columns: Class No, Class Activity, Class Day, Class Time, Male/Female/Mixed, and Class Tutor. Row 15 is selected. To the right of the table is a 'See Room/Hall availability' button. Below the table is a 'Room/Hall/Court' dropdown menu showing 'Fitness Suite', 'Sports Hall 1', and 'Sports Hall 2'. Further down are 'Class No', 'Class Day', and 'Class Time' text boxes. At the bottom, there are three tables showing availability for 'Fitness Suite', 'Sports Hall 1', and 'Sports Hall 2'. Each table has columns for 'ClassDate' and 'Class No'.

Class No	Class Activity	Class Day	Class Time	Male/Female/Mixed	Class Tutor
3	Body Conditioning	Monday	15:00	Mixed	Latham
9	Body Conditioning	Wednesday	14:00	Mixed	Latham
15	Body Conditioning	Thursday	19:00	Mixed	Latham
16	Body Conditioning	Friday	10:00	Female	Latham
7	Family Multi-gym	Tuesday	19:00	Mixed	Jackson
1	Ladies' Aerobics	Monday	10:00	Female	Evans
8	Ladies' Aerobics	Wednesday	10:00	Female	Evans
6	Ladies' Multi-gym	Tuesday	14:00	Female	Adams
5	Men's Multi-gym	Tuesday	10:00	Male	Jackson
18	Men's Multi-gym	Friday	14:00	Male	Jackson
14	Multi-gym	Thursday	15:00	Mixed	Adams
4	Step Aerobics	Monday	19:00	Mixed	Wheildon

ClassDate	Class No
09/05/1996	
16/05/1996	
23/05/1996	
30/05/1996	

ClassDate	Class No
09/05/1996	
16/05/1996	
23/05/1996	
30/05/1996	

ClassDate	Class No
09/05/1996	
16/05/1996	15
23/05/1996	
30/05/1996	

Fig 8.3.10 The ClassBooking form so far, showing room availability for the selected class at the dates and time required.

8.3.5 Making the Bookings and checking for double-bookings.

Add a non-wizard command button with the caption *Confirm Bookings* and the name `cmdConfirm`. Set it to disabled initially, and enable it when the user chooses a room (use the `AfterUpdate` event for the `Room/Hall/Court` text box). Create a `Click` event for the button and add the code shown in Fig 8.3.11.

Points to note:

- Even though the unique index `BookingSlot` (see Fig 8.2.15) has been created, it does not appear to cause a trappable error (i.e. where you can code an `On Error` statement) when inserting records using embedded SQL in this way.
 - You must therefore code your own check for a double-booking and report the fact to the user.
 - However, if you do attempt to write a record that causes a duplicate key then the SQL statement is simply ignored.
- The `myUSADate` function is used again (see Fig 8.3.7) for the calculated date.
- The bookings are written out in a loop, using the same loop controls as when creating the dates for the `BookingDate` table.
 - A better (safer) method may well be to use Data Access Objects (DAO) code to read the `BookingDate` table into a `Recordset` and use the same dates as on there. `Recordsets` are discussed in the Further VBA Trainer.
- The room list boxes are queried after each set of bookings to show the new state of affairs.

```

Private Sub cmdConfirm_Click()
'make the bookings - non-wizard code

Dim dtDate As Date
Dim strSQL As String
Dim intCount As Integer

dtDate = txtStartDate 'start with the first date
DoCmd.SetWarnings False

Do Until dtDate > CDate(txtEndDate)
'insert each booking into the Booking table
'must check first for double-booking
strSQL = "[Room/Hall/Court] = " & [Room/Hall/Court] & "" _
& " AND [Date] = #" & myUSADate(dtDate) & "#" _
& " AND [Time] = #" & [Class Time] & "#"
intCount = DCount("[Booking No]", "Bookings", strSQL)

If intCount = 0 Then 'OK - can book
strSQL = "INSERT INTO Bookings ([Room/Hall/Court], " _
& "[Member/Class], " _
& "[Class No], [Date], [Time]) " _
& "VALUES (" & [Room/Hall/Court] & ", " _
& "No," _
& [Class No] & ", " _
& "#" & myUSADate(dtDate) & "#," _
& "#" & [Class Time] & "#)"
DoCmd.RunSQL strSQL
Else 'already booked
myDisplayWarningMessage "This slot has been booked by another class - cannot book" & vbCrLf _
& [Room/Hall/Court] & " on " & dtDate & " at " & [Class Time]
End If
dtDate = dtDate + 7 'date for next week
Loop

DoCmd.SetWarnings True
IstFSuite.Requery 'requery each room availability
IstSHall1.Requery
IstSHall2.Requery
cmdConfirm.Enabled = False 'to stop user attempting to make the booking again

End Sub

```

Fig 8.3.11 Code to make Class Bookings.

8.3.6 Finally...

Just as for the Member Bookings in Fig 8.2.20, there are a number of things that could be done to improve the ClassBooking form. See Fig 8.3.12. Note that this form is not bound to a table or query, so you cannot use wizards for buttons, but must work out your own code.

Item	Comments
Validate the dates in txtStartDate and txtEndDate (see sections 3.3.1 and 3.3.2).	Similar to CourtBooking form. Enable cmdSee only if valid; see Fig 8.3.5. You could also use txtDayNo to validate that the dates are for the correct day of week (use Weekday function).
Add a <i>Cancel Booking</i> button.	Clear the data fields, reset other properties as appropriate.
Add a <i>New Booking</i> button.	Clear the data fields, reset other properties as appropriate.
Add a <i>Delete Booking</i> button	Delete single bookings as on CourtBooking form. You could also delete all bookings for this class within the two dates.
Ask an 'Are You Sure?' question before confirming and deleting bookings.	Similar to CourtBooking form.
Trap unsaved changes and give the user a chance to confirm or cancel a booking.	You will need to check to see if the user has clicked on any controls or entered data to start a booking; a suggestion is that you check for non-null values in the data fields on the form.
Tidy the form up and improve functionality.	Similar to CourtBooking form. You may also need to enable/disable controls appropriately. Add a 'see all' / 'see free' facility as on Court Booking form. Don't make booking if Centre is closed that day (see Fig 8.2.20)

Fig 8.3.12 Suggestions for improvements to the Class Booking facility. Work out your own test plan.

8.4 Example 3 – Making bookings using a ‘diary page’ grid.

Manual systems often use a diary, with pages that have a column for each room (or doctor, advisor, etc) across the top and times for bookings (or appointments etc) down one (perhaps the left-hand) side. The booking details (perhaps name/number of person booking) are entered in the booking cell where the required column and row intersect. A form that uses the same structure could therefore be very useful for making bookings for a given day. This means that all bookings must be for a single day only; it is not possible to make Class bookings for a range of days (as in section 8.3) with this method.

A Crosstab query seems the obvious choice here to use for the form, as this will provide the data summarised in the way required. This can be problematic, as a room that has no bookings will not give a column in the result, so opening a form based on the query will then fail if the form is expecting that column. The example here shows a method of trapping this error and taking appropriate action.

It is also possible to create a diary page grid using Data Access Objects (DAO) code and arrays. This method is demonstrated in the Further VBA Trainer, and shows the Booking No in each booking cell.

8.4.1 Creating a Crosstab query.

Crosstab queries have the format of rows and columns that correspond to a diary page grid mentioned above. However, they do not allow the user to enter, for example, the Booking No in the required cell, as they are designed to provide aggregate totals (counts, averages, maxima, etc), so the value in each cell is such a numeric total.

See <http://www.cse.dmu.ac.uk/~mcspence/Access.htm> Access FAQ 17 *What is a CROSSTAB query?*

First create a query to select from the Bookings table as shown in Fig 8.4.1. Later on (section 8.4.3) this query will be changed to add a criterion to select just those bookings for the required date. The Bookings table data as used in *McBride* has a booking for each room, which is important here as we want all rooms to have data for the Crosstab query, at least to start with to be able to create the form.

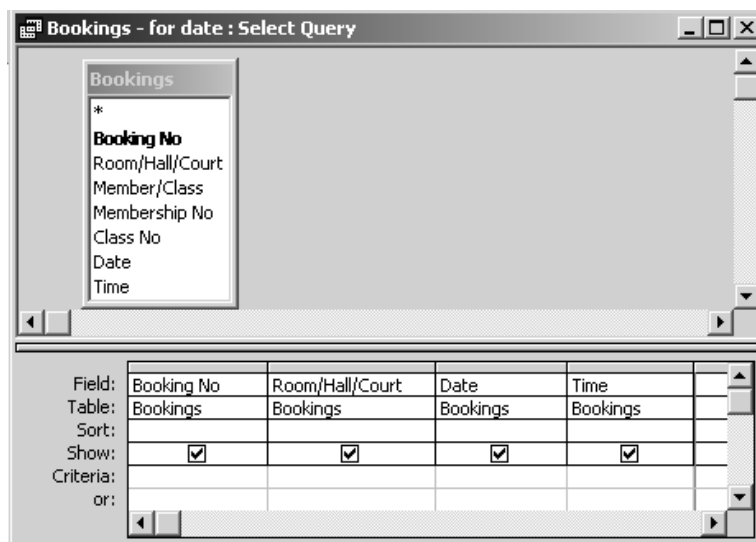


Fig 8.4.1 Selecting columns from the Booking table.

But this query will only select rows for which there is a booking. Create an Outer Join query as shown in Fig 8.4.2 so that there is now at least one row for every possible booking time. Note that the query makes (re)use of the BookingTime table from section 8.2.1.

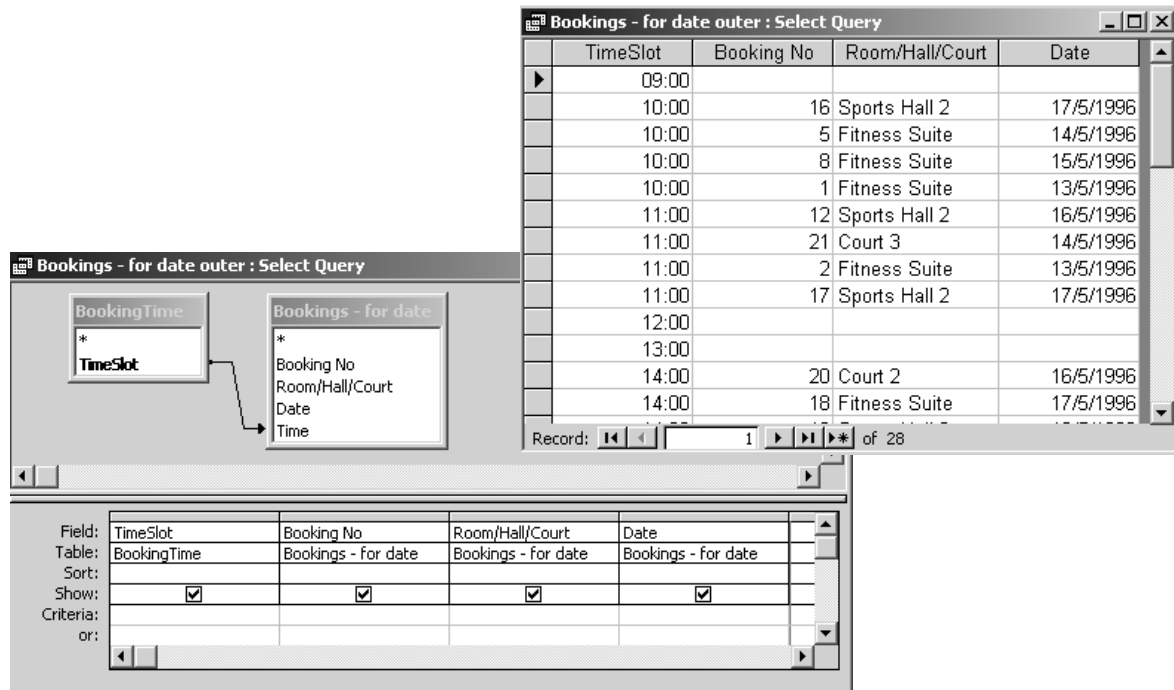


Fig 8.4.2 Showing outer join query of bookings (Data is as original Bookings table in McBride)

Now create a Crosstab query based on the second query:

- Start a new query – choose the Crosstab Query Wizard.
- Select the 'Bookings - for date outer' query.
- Select TimeSlot for the row heading
- Select Room/Hall/Court for the column heading
- Choose to count the Booking No for the calculated column. This will count up the number of bookings for each slot.
- Give the query the name of Bookings_Crosstab.

You should now have a query that gives a result like that shown in Fig 8.4.3. The query counts up all the bookings for each room and shows the count in the cell for the room and time. Some counts are greater than 1 as we have not yet restricted the data to just one day.

The rather odd column headed <> is for those rows where there was no value in Room/Hall/Court.

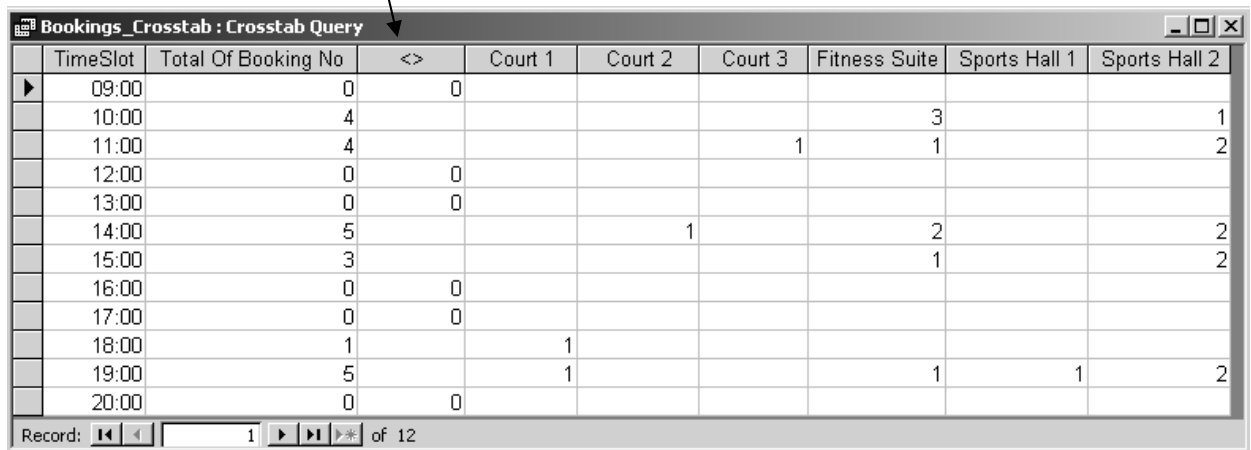


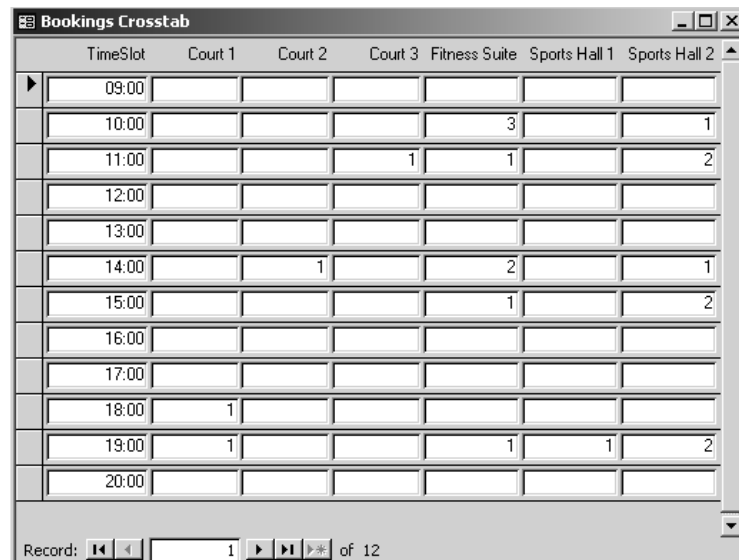
Fig 8.4.3 the result of the Bookings_Crosstab query

8.4.2 Create the diary page form.

Using the form wizard, create a tabular form based on the `Bookings_Crosstab` query. Choose all the fields except for the 'Total of Booking No' and '<>' fields, as these are not required.

Set a format of Short Time for the time, as this looks better than a format that shows seconds as well as hours and minutes.

Your form should now look like Fig 8.4.4.



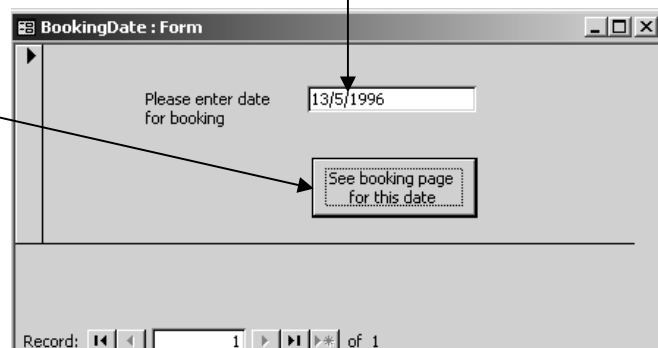
TimeSlot	Court 1	Court 2	Court 3	Fitness Suite	Sports Hall 1	Sports Hall 2
09:00						
10:00				3		1
11:00			1	1		2
12:00						
13:00						
14:00		1		2		1
15:00				1		2
16:00						
17:00						
18:00	1					
19:00	1			1	1	2
20:00						

Fig 8.4.4 The `Bookings Crosstab` form, showing all bookings from the `Bookings` table.

8.4.3 Specify a booking date via a parameter

The user will want to see just the details for a given date, and will normally specify that date via a parameter on a form. So create a simple unbound form (`BookingDate`) with a textbox (`txtDate`) for the date and a wizard button to open the `Bookings Crosstab` form.

See Fig 8.4.5.



Please enter date for booking

13/5/1996

See booking page for this date

Record: 1 of 1

Fig 8.4.5 Simple form to specify booking date and open `Bookings_Crosstab` form

Go back to the `Bookings - for date` query of Fig 8.4.1 and add the following Forms Collection reference to the criterion cell for the date:

```
[forms]![BookingDate]![txtDate]
```

Enter a date of 13/5/1996 (as this date matches the McBride data) and run the `Bookings for date` query. You will see just the bookings for this date.

Now try opening the Bookings_Crosstab query or the Bookings_Crosstab form from the BookingDate form. You will get the error message shown in Fig 8.4.6 in both cases.

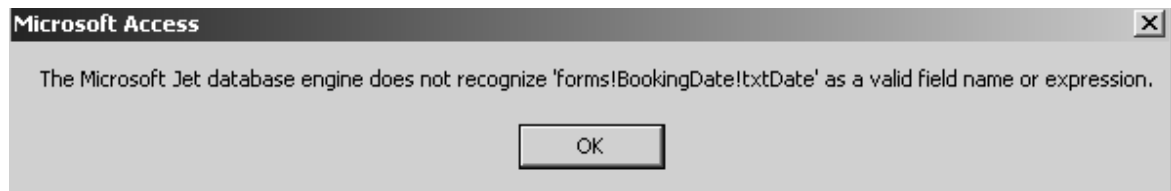


Fig 8.4.6 Error message when referencing a variable parameter value in a Crosstab query

Crosstab queries cannot cope with the usual methods of referencing variable parameters at run time. The parameter reference does not have to be in the Crosstab query itself, it could be in an underlying query (as it is here). Note that it is OK to specify literal values as criteria; for example, if the date was specified as #13/5/1996# in the query then all would be well, but in this case the query would not be much use.

However, a solution is not too difficult. See VBA FAQ 14 *How can I add a criterion to a Crosstab query?* on <http://www.cse.dmu.ac.uk/~mcspence/Access.htm>

Amend the wizard code for the command button on the BookingDate form, by adding the lines in bold in Fig 8.4.7. This code reuses the myCreateDateTable procedure from section 8.3.3. Here the start and end dates are the same, so the table will only have one row in it, with the date on the BookingDate form.

Change your query (see fig 8.4.7) so that the criterion now is:

In (SELECT ClassDate FROM BookingDate)

The query will now look at the row(s) in the BookingDate table and select only those dates that match these row(s). In this example there should only be one row, with the date from the BookingDate form. This is an example of using a sub query; you may find it useful to look at the SQL.

```
Private Sub cmdOpenDiaryPage_Click()
'wizard code to open form
'amended to create date table row for the form query
On Error GoTo Err_cmdOpenDiaryPage_Click

    Dim stDocName As String
    Dim stLinkCriteria As String

    'first put date in a temporary table for query to pick up
myCreateDateTable CDate(txtDate), CDate(txtDate) 'see separate module

    stDocName = "Bookings Crosstab"
    DoCmd.OpenForm stDocName, , , stLinkCriteria

Exit_cmdOpenDiaryPage_Click:
    Exit Sub

Err_cmdOpenDiaryPage_Click:
    MsgBox Err.Description
    Resume Exit_cmdOpenDiaryPage_Click

End Sub
```

Field:	Booking No	Room/Hall/Court	Date	Time
Table:	Bookings	Bookings	Bookings	Bookings
Sort:				
Show:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Criteria:			In (SELECT ClassDate FROM BookingDate)	
or:				

Fig 8.4.7 code to set up the date parameter, and the amended query which now references the row in the BookingDate table

8.4.4 Trapping the form error where Crosstab columns are missing.

If you put the date of 13/5/1996 in the BookingDate form and open the form you may get an error similar to that in Fig 8.4.8, and one or more columns on the form may have #Name? in all cells for that column. There is sometimes an alternative message that references a specific field name.



Fig 8.4.8 error message if a field is missing from the Crosstab query.

But if you run the *Bookings_Crosstab* query it will pick up the data for the required date without a problem. But the result of the query shows that not all of the rooms have columns as this date does not have bookings for all rooms. This causes the error when the form is loaded (as shown in Fig 8.4.8) as the form is expecting columns for all the rooms.

The error number is 2424 (find these numbers out by a `MsgBox Err` statement in your code, as shown in Fig 8.2.17) so it is now straightforward (if tedious) to add code to the `Form_Load` event to trap this error and adjust the cell contents. Add the code shown in Fig 8.4.9 to your *Bookings_Crosstab* form.

```
Private Sub Form_Load()
'trap error where columns are missing and change field properties
On Error GoTo Err_Form_Load

If [Court 1] = "#Name?" Then      'if the field is missing then will get error 2424
    'it doesn't matter what is checked for here – the code will simply fail if the field is not on the form
    [Court 1].ControlSource = ""  'must unbind as the field does not exist
    [Court 1] = Null              'set to Null to remove '#Name?' and show as unbooked
End If

If [Court 2] = "#Name?" Then
    [Court 2].ControlSource = ""
    [Court 2] = Null
End If

If [Court 3] = "#Name?" Then
    [Court 3].ControlSource = ""
    [Court 3] = Null
End If

If [Fitness Suite] = "#Name?" Then
    [Fitness Suite].ControlSource = ""
    [Fitness Suite] = Null
End If

If [Sports Hall 1] = "#Name?" Then
    [Sports Hall 1].ControlSource = ""
    [Sports Hall 1] = Null
End If

If [Sports Hall 2] = "#Name?" Then
    [Sports Hall 2].ControlSource = ""
    [Sports Hall 2] = Null
End If

Exit Sub

Err_Form_Load:
If Err = 2424 Then 'error number for the error where columns are missing
    Resume Next  'resume at statement *after* the error, to change field properties
Else
    MsgBox Err.Description
End If

End Sub
```

See also Fig 8.4.17. The check (and adjustments) for missing fields are made again if all bookings for a room have been deleted.

Part 2 of the Further VBA Trainer has more information about error-trapping.

Fig 8.4.9 Trapping missing fields and correcting cell contents

8.4.5 Using Conditional Formatting for booked/free slots.

Your form should now look like the form in Fig 8.4.4, but each slot should only have a 1 (booked slot) or be Null (free slot). Any other value (such as 2) would indicate a problem, possibly a double-booking.

But the form would look much better if each slot was formatted much more clearly to show which was free and which was booked. I cannot find out how to do this via VBA, and one textbook I have read stated that it is not possible. However, a new feature in Access 2000 was that of Conditional Formatting, and with this feature the required formatting is simple. Click on the fields that you want to format, then go via *Format* → *Conditional Formatting* on the main menu and then set the required formatting as shown in Fig 8.4.10. Choose the back fill and font colours that you want to use. Here I have set them both to the same colour, so that the cell contents do not show up; light grey is used for a booked slot and white for a free slot. (Note that the different choices only show up in the 'preview' box; the choice boxes on the right of the conditional formatting dialog box seem to be the same all the way down).

Font and BackColor both = grey.

Font and BackColor both = white.

By holding down the Shift key, you can select all the room fields at once.

Formatting changed and fields locked. Uses BackStyle, SpecialEffect, BorderStyle and Locked field properties.

Code in Form_Load event to get the date from the BookingDate form (uses Forms Collection reference) and put details in the header. Uses Weekday and WeekdayName functions for the weekday name (txtDayName), and FormatDateTime to display the date (txtDate). See section 4.2.1.

Bookings for: Monday 13 May 1996		TimeSlot	Court 1	Court 2	Court 3	Fitness Suite	Sports Hall 1	Sports Hall 2
	09:00							
	10:00							
	11:00							
	12:00							
	13:00							
	14:00							
	15:00							
	16:00							
	17:00							
	18:00							
	19:00							
	20:00							

Fig 8.4.10 Showing the setting, and the result, of the Conditional Formatting for the Bookings Crosstab form. Grey slots are booked, white slots are free. Some other formatting and changes have also been made to the form.

8.4.6 Making Bookings.

Now we can get down to using the form. So far, not much VBA has been needed, but that will now change.

8.4.6.1 Making a start.

Add the code shown in Fig 8.4.11 to the Click event for the [Court 1] field, and try clicking on a white (free) and grey (booked) slot. You should see the appropriate message displayed. This coding (suitably adjusted – but don't do it yet) will work for the other fields, even for those where there is not a column in the underlying Crosstab query. As we are able to test for, and find, booked and free slots, we can make bookings for the free slots.

```
Private Sub Court_1_Click()
    If IsNull([Court 1]) Then 'is the slot Null?
        MsgBox "This slot is free"
    Else
        MsgBox "This slot is already booked"
    End If
End Sub
```

Fig 8.4.11 Testing for booked and free slots.

When the user clicks on a slot, a code reference to the value in the TimeSlot field will be for the value for the relevant row. We will know from the slot itself which room is wanted and thus whether or not the booking is for a Member or a Class (Courts are for Member bookings, other rooms are for Class Bookings).

There are several rooms in this booking grid, so a useful concept to apply here is that of 'hidden fields' on the form (see section 4.3.8), where data is put in the field(s) in one procedure for other procedures to pick up. Create three new text boxes (delete the labels) in the footer for the Bookings Crosstab form:

- txtRoomHallCourt – to be used to store the room name.
- txtMemberClass – to be used to store True for Member bookings and False for Class bookings.
 - o The Member/Class field on the Bookings table is a Yes/No field, where Yes means Member and No means Class.
- txtTime – the time for the booking.

There is no need for a field for the booking date as this is in the form header (see Fig 8.4.10, txtDate). The fields should be set as Visible = No eventually, but if you leave them visible for now you will see how they work.

```
Private Sub Court_1_Click()
    'put values in hidden fields on form.
    myHiddenFields myconCourt1, myconMember

    'check if slot free or booked and take appropriate action.
    myCheckSlot [Court 1]
End Sub

Private Sub myCheckSlot(prmSlot As TextBox)
    'see if slot is free or booked.

    If IsNull(prmSlot) Then 'is the slot Null?
        MsgBox "This slot is free"
    Else
        MsgBox "This slot is already booked"
    End If
End Sub

Private Sub myHiddenFields(prmRoomHallCourt As String, prmMemberClass As Boolean)
    'called when user clicks on a slot
    'copies the information to hidden fields on the form, to be used by booking and deletion processes.

    txtRoomHallCourt = prmRoomHallCourt
    txtMemberClass = prmMemberClass
    txtTime = TimeSlot
End Sub
```

Uses constants set up in Fig 8.1.1.

Passes the textbox itself as a parameter.

Checks to see if the given slot is free or booked.

Puts slot information in hidden fields for other processes to pick up.

Fig 8.4.12 Setting up the hidden fields and checking if the slot is free or booked

Change the Click event for [Court 1] to now be as shown in Fig 8.4.12, and add the new procedures myCheckSlot (checks for a free or booked slot) and myHiddenFields (puts the booking values in the hidden fields in the form footer). Click on a Court 1 slot and see what happens in these fields. See Fig 8.4.16.

All that will now be needed for a Court 1 booking is the Membership or Class number.

8.4.6.2 Setting up a Bookings form.

Create the simple Bookings form from Unit 16 of McBride; this is just a wizard form using the Bookings table.

Make changes to it as shown in Fig 8.4.13:

- Remove scroll bars, record selector and navigation bar.
- Move the Class No and Membership No fields so that they overlap (you will see why in Figs 8.4.14 and 8.4.16).
- Set the other fields to suitable properties so that they do not look like data entry fields, and set the Locked property to Yes.

Fig 8.4.13 The basic Bookings form

This form can now be opened from the Bookings Crosstab form and the values in the hidden fields on that form can be copied to the relevant fields on the Bookings form when the Bookings form is opened, in the Form_Load event. The user then enters the Membership or Class No and saves the booking (for now, closing the form will do that). When the form is closed, code in the Form_Close event will close and reopen the Bookings Crosstab form, thus refreshing it and showing the new booking on that form.

Put the code shown in Fig 8.4.14 in your Bookings form code module.

```
Private Sub Form_Close()
'close and reopen Bookings Crosstab form to refresh it and show the new booking.

    DoCmd.Close acForm, "Bookings Crosstab"
    DoCmd.OpenForm "Bookings Crosstab"

End Sub

'-----
Private Sub Form_Load()
'copy known information into Bookings form from hidden fields on Bookings Crosstab form

    [Room/Hall/Court] = Forms![Bookings Crosstab]!txtRoomHallCourt
    [Member/Class] = Forms![Bookings Crosstab]!txtMemberClass
    [Date] = Forms![Bookings Crosstab]!txtDate 'this is the form header date
    [Time] = Forms![Bookings Crosstab]!txtTime

'check for member or class booking
If [Member/Class] = myconMember Then 'show Membership No field ready for data entry
    [Membership No].Visible = True
    [Class No].Visible = False
    [Membership No].SetFocus
Else 'show Class No field ready for data entry
    [Membership No].Visible = False
    [Class No].Visible = True
    [Class No].SetFocus
End If

End Sub
```

Fig 8.4.14 code for the basic Bookings form for form Load and Close events.

8.4.6.3 Making Member/Class Bookings.

Change your [Court 1] Click event so that it looks like the code in Fig 8.4.15, and add the new procedure myBooking.

```
Private Sub myCheckSlot(prmSlot As TextBox)
'see if slot is free or booked.

    If IsNull(prmSlot) Then    'is the slot Null?
        myBooking    'call booking procedure
    Else
        MsgBox "This slot is already booked"
    End If

End Sub

'-----
Private Sub myBooking()
'called when a free slot is clicked
'Bookings form uses the information in the hidden fields to make the booking

    DoCmd.OpenForm "Bookings", , , , acFormAdd

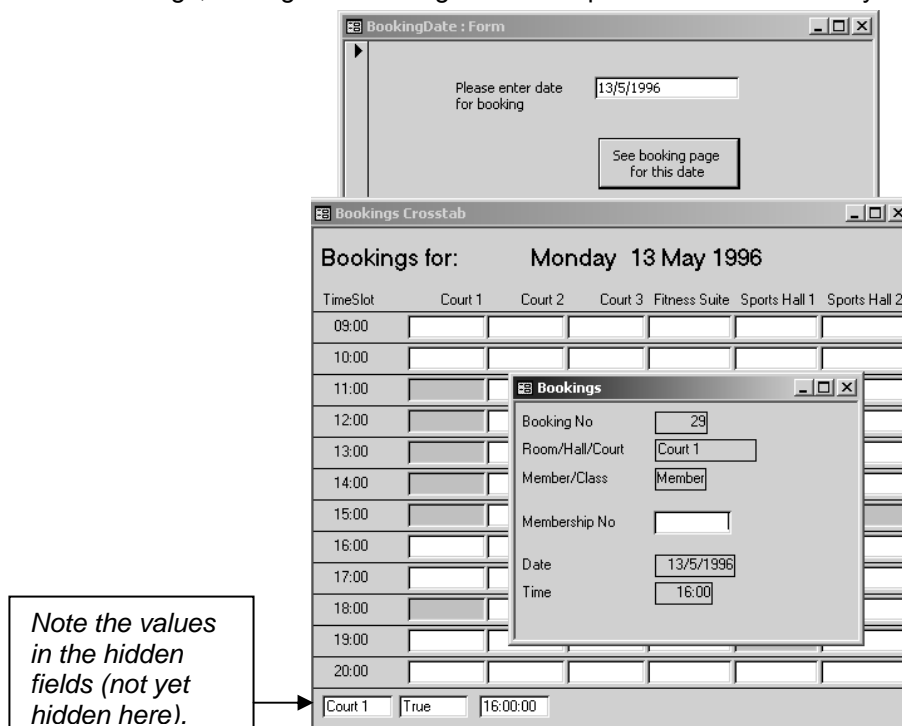
End Sub
```

Create click events for the other two Courts. Then you can make member bookings for all three Courts.

Fig 8.4.15 opening the Bookings form for a member booking (new code in bold)

Now, open the BookingsDate form, enter a date and click on the See booking page for this date button. Click on a free Court slot and see the Bookings form open ready for the Membership No to be entered. Enter a Membership No and close the form; the booking will be saved automatically and the Bookings Crosstab form will be refreshed to show the new booking. See Fig 8.4.16.

The process needs to be improved to ensure that the user does enter a Membership No and to check for double-bookings, amongst other things. These improvements are left for you to do; see Fig 8.4.18.



Note the values in the hidden fields (not yet hidden here).

Fig 8.4.16 making a Member Court booking.

You should now be able to work out how to make Class bookings. Simply create Click events for the Fitness Suite and the two Sports Halls following the example shown in Fig 8.4.12 for Court 1, remembering to use myConClass instead of myConMember. Note how useful reusable procedures are.

8.4.6.4 Using a list box for the Membership/Class No.

It is not good practice to expect the user to type in the Membership/Class No (and certainly not good practice simply to assume that it will be for a valid member/class!). Unless your system is using (or simulating) a swipe card for all entries, a list box as used in sections 3.6, 8.2.6 and 8.3.1 could be a good way to ensure correct data entry.

Extend your booking form and add two list boxes, one each for Member and Class, with associated textboxes to allow the user to filter the contents of the boxes. Add code to make the list boxes work correctly and to put the chosen number in the Membership No Or Class No field as appropriate. To make sure that the user does not type the number directly in the Bookings form set the Locked properties for these two fields to Yes.

The form will look decidedly cluttered with two list boxes, and will also be misleading as only one list box is needed at a time. So move the list boxes and the associated filter text boxes so that they overlap, and add code to the Bookings form Form_Load event to set the Visible properties for the text and list boxes so that only the relevant items are visible for the type (Member or Class) of booking. The code in Fig 8.4.14 already does this for the Membership No and Class No fields, so you merely need to add to this coding. You should then be able to reduce the size of your form.

8.4.7 Deleting Bookings.

The code done so far has put the booking details in hidden fields on the Bookings Crosstab form, but we do not know the Booking No for the booking. However, using the DLookup function we can get this, plus the details of the Member/Class for the booking. We can then display the details of the bookings and ask the user if they want to delete the booking or not, and take action as appropriate.

The code to do this will go in a new procedure in the Bookings Crosstab form module. See Fig 8.4.17. There's a lot of code here, but it is all pretty straightforward:

1. The code starts with a set of variable declarations, into which the booking details and various SQL clauses/statements will be put.
 - o Note that the Booking No and other IDs are AutoNumber fields so are Long Integer (look at the field definition in table design view).
2. Given the room, date and time, there should be just one booking record in the Bookings table, so the DLookup function is used to find that Booking No.
 - o The variable strWhere is used for the WHERE condition for DLookup, rather than coding the condition directly into the function. This makes it easy to check at run-time (or failure-time!) what the condition looks like, using the Debugger.
3. Bookings are for either a Member or a Class, as indicated by the value (True/False) in the hidden field txtMemberClass. So this value is then used to get the Member or Class details as appropriate. Lots more DLookup statements are used (this is a very useful function).
 - o The condition in strWhere is reused, another reason for putting it in a variable.
4. Now that we have the details of the booking we can display them and ask the user if he/she wishes to delete the booking.
 - o For simplicity here I have used a standard format message, but the details could vary for Member/Class bookings. For example, it could be useful to show the telephone number for a member booking in case the user wishes to contact them about the booking.
 - o The reply from the Yes/No question has been put into a variable, partly to simplify the coding a bit (it's a long message to display and there's a fair bit of coding to do if the user replies Yes) and partly so that it can be checked in the Debugger whilst testing.
5. If the user replies Yes, then an embedded SQL statement is used to DELETE the record for this booking; easy to do as we have found the Booking No.
 - o DoCmd.SetWarnings is used to suppress the standard Access messages ("You are about to delete..." etc).
 - o After the deletion has been done, the form is requeried using Requery. This will rerun the query on which the form is based and redraw the form, thus showing the slot as now free. Compare this with Repaint which merely completes any pending updates for a normal bound form.
6. Finally, a check needs to be made to see if the room is now free all day, in which case the column will be missing from the Crosstab query, and the form will show "#Name?" all the way down the column. It looks like Requery does not cause the Form_Load event to be called again. See Fig 8.4.9. And that's it.

```

Private Sub myDeletion(prmslot As TextBox)
'called when a booked slot is clicked
'this code uses the information in the hidden fields
Dim lngBookingNo As Long      'booking number for the selected booking
Dim strWhere As String        'used for complex WHERE conditions for agg functions and SQL
Dim lngMemberNo As Long      'Membership No if a Member booking
Dim lngClassNo As Long       'Class No if a Class booking
Dim strDetail1 As String      'member Lastname, or Class Activity
Dim strDetail2 As String      'member Firstname or Class Tutor
Dim intDelete As Integer      'yes/no reply from MsgBox (check with vbYes/vbNo)
Dim strSQL As String          'SQL for RunSQL

'first get the booking no
strWhere = "[Room/Hall/Court] = " & txtRoomHallCourt & "" _
          & " AND [Time] = #" & txtTime & "#" _
          & " AND [Date] = #" & txtDate & "#"

lngBookingNo = DLookup("[Booking No]", "Bookings", strWhere)

'then get the member/class details for the booking
If txtMemberClass = True Then 'member booking
lngMemberNo = DLookup("[Membership No]", "Bookings", strWhere)
strDetail1 = DLookup("Lastname", "Memberships", "[Membership No] = " & lngMemberNo)
strDetail2 = DLookup("Firstname", "Memberships", "[Membership No] = " & lngMemberNo)
Else 'class booking
lngClassNo = DLookup("[Class No]", "Bookings", strWhere)
strDetail1 = DLookup("[Class Activity]", "Classes", "[Class No] = " & lngClassNo)
strDetail2 = DLookup("[Class Tutor]", "Classes", "[Class No] = " & lngClassNo)
End If

'Now ask the user to confirm the booking
intDelete = myYesNoQuestion("Booking details are:" & vbCrLf & vbCrLf _
    & "Booking No: " & lngBookingNo & vbCrLf _
    & "By: " & strDetail1 & ", " & strDetail2 & vbCrLf _
    & "For: " & txtRoomHallCourt & " on " & txtDate & " at " & FormatDateTime(txtTime, vbShortTime) _
    & vbCrLf & vbCrLf & "DELETE THIS BOOKING?")

If intDelete = vbNo Then
'do nothing
Else
'delete the booking
DoCmd.SetWarnings False
strSQL = "DELETE * FROM Bookings WHERE [Booking No] = " & lngBookingNo
DoCmd.RunSQL strSQL
DoCmd.SetWarnings True
Requery 'requery the form to show the slot now free

'check to see if all bookings for this room and date have been deleted
strWhere = "[Room/Hall/Court] = " & txtRoomHallCourt & "" _
          & " AND [Time] = #" & txtTime & "#"
If DCount("[Booking no]", "Bookings", strWhere) = 0 Then 'yes - room free all day
MsgBox "bookings all gone" 'testing only
prmslot.ControlSource = ""
prmslot = Null 'clear the error "#Name?#" contents of the field
End If

[court 1].SetFocus 'otherwise this goes to the first time slot and looks weird.
End If
End Sub
    
```

Explanation point 1

Explanation point 2

Explanation point 3

Explanation point 4

Explanation point 5

Explanation point 6

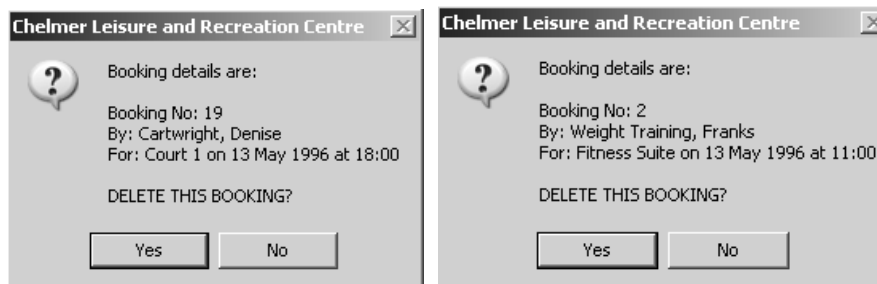


Fig 8.4.17 Procedure to delete a booking

8.4.8 Finally...

You have now seen how to create a Booking Diary Page form, and a way of making and deleting bookings via this form.

Some improvements that should be made to the booking process are listed in Fig 8.4.18. You should be able to work out how to do these by following earlier examples in this document.

Item	Comments
Validate the date parameter in the BookingDate form. Perhaps change the form to use a calendar control.	As in Fig 8.2.20. See section 5.6.
On the Bookings form: <ul style="list-style-type: none"> • Add a <i>Confirm Booking</i> button, check that a Membership or Class number has been entered, and trap double-bookings. • Add a <i>Cancel Booking</i> button. • Trap unsaved changes • Use a list box for the user to select the required member/class. 	<ul style="list-style-type: none"> • Use wizard for <i>Save Record</i> and add code in the error procedure to check for double booking as in as in Fig 8.2.17. Ask an 'Are you sure?' question. • Clear the data fields, close the form. Could use Wizard for <i>Undo</i>. • User may attempt to close the form after entering a Class No or Member No. See section 2.5.2. • See section 8.4.6.4.
Stop the user from making/deleting bookings for dates in the past.	You could check the date in txtdate in the myBookings/myDeletions procedures, or include this as part of the date validation checks. The latter will not allow the user to check details of past bookings, but that may not matter.

Fig 8.4.18 Suggestions for improvements to the diary page bookings facility. Work out your own test plan.

8.5 Exercises

8.5.1 Member Bookings

Implement and test the tasks discussed in section 8.2.10.

8.5.2 Class Bookings

Implement and test the tasks discussed in section 8.3.6.

This example was not illustrated in as great a detail as example 1, as the items missed out have been demonstrated elsewhere in this Trainer and in this Part of the Trainer.

8.5.3 Using a 'diary page' grid.

Implement and test the tasks discussed in section 8.4.8.

8.5.4 Recording attendance.

With some booking systems (hospital, doctor or dentist appointments, for example) it is important to be able to record whether or not a client has attended the appointment. The system can then analyse missed appointments, and write to (or even blacklist!) offenders.

A way of doing this could be to...

- ...add a Yes/No field (*Attended*) to the *Bookings* table, with a default value of *No*.
- ...when a user clicks on a booked slot (sections 8.2.9 and 8.4.6) give them the choice of deleting, recording attendance or cancelling.
 - o You could use `MsgBox` with `vbYesNoCancel` buttons (or create your own `myYesNoCancelQuestion` procedure – see section 1.7.2) and tell the user to click *Yes* for *Delete*, *No* to record attendance or *Cancel* to do nothing. This would be fairly simple to code but could be confusing for the user. As far as I can see, it does not appear to be possible to specify your own button text for a `MsgBox` question.
 - o A better method may be to have a new form which displays the required booking details, and has its own *Confirm Delete*, *Record Attendance* and *Cancel* buttons behind which you put the appropriate code.
- ...if the user chooses to record attendance then use an embedded SQL UPDATE statement to SET the *Attended* field value to *Yes* for the specified booking. See section 6.4.

END OF MAIN PART OF TRAINER

The Appendices are on the following pages.

The Index is at the end of the document.

Appendix A – Events Overview

For the full list of events for an object or control, consult the property box

Event	Occurs on:								When Invoked	Example of Use
	Frm	Rpt	Cmd btn	Txt box	Cbo box	Opt btn	Frm sec	Rpt sec		
On Current	✓								Focus moves from one record to another	Set form defaults for each new record.
Before Insert	✓								When first character is typed in new record	
After Insert	✓								After a new record is inserted	Confirmation message to user
Before Update	✓			✓	✓				Before new field or record is updated	Custom validation(s)
After Update	✓			✓	✓				After new field or record is updated	Field – calculate/copy values automatically; set buttons Form – confirmation message to user
On Delete	✓								When record is deleted	Confirmation message to user
Before Del Confirm	✓								Before confirming a deletion	'Are you sure?' procedure
After Del Confirm	✓								After confirming a deletion	Confirmation message to user
On Open	✓	✓							Before form or report is opened	Change/set defaults
On Load	✓								When form or report is loaded	Set form or report defaults; copy data from another form
On Resize	✓								When a form is resized	
On Unload	✓								Before form or report is unloaded	
On Close	✓	✓							Before form or report is closed	Display statistics/totals; open another form
On Change				✓	✓				When the data is changed	Confirm message/question
On not in list					✓				When value not in limited list is entered	Dynamic updating of list for combo box based on a table
On Enter			✓	✓	✓				When a control first gets the focus	Change font
On Exit			✓	✓	✓				When a control loses the focus on the same form	Reset font
On Got Focus	✓		✓	✓	✓	✓			When control or form gets the focus	Change default properties
On Lost Focus	✓		✓	✓	✓	✓			When control or form loses the focus	Reset default properties
On Click	✓		✓	✓	✓		✓		When control is clicked	Command buttons
On Double Click	✓		✓	✓	✓		✓		When control is double-clicked	Select item from list box
On Mouse Down	✓		✓	✓	✓	✓	✓		When mouse button is pressed	Pop-up menu or help?
On Mouse Move	✓		✓	✓	✓	✓	✓		When mouse moves	
On Mouse Up	✓		✓	✓	✓	✓	✓		When mouse button is released	
On Key Down	✓		✓	✓	✓	✓	✓		When a key is pressed	Check for custom hot keys used
On Key Up	✓		✓	✓	✓	✓	✓		When a key is released	
Key Preview	✓								Yes = invoke keyboard events for forms before keyboard events for controls. No = other way around	
On No Data		✓							When a report has no records	Appropriate information to user; empty report, cancel
On Page		✓							Before a page is printed	Warn user if special stationery is needed
On Error	✓	✓							When a run-time error occurs in a form or report	Suitable message or action
On Filter	✓								When a filter is edited	
On Apply Filter	✓								When a filter is applied or removed	
On Timer	✓								When timer interval reaches zero	Close application if left unused; dynamic date/time
Timer Interval	✓								Specify timer interval in milliseconds	Use with On Timer event
On Format							✓		Before section is formatted	Suppress detail lines
On Print							✓		Before section is previewed, printed or saved	Dynamic changes to controls on report

Appendix B – Coding Standards

Standard	Reason
1 Always put explanatory comments in your code.	Comments make the code easier to understand and debug.
2 Use meaningful names for procedures, variable names, etc.	Meaningful names make the code easier to understand, write and debug.
3 Indent your code.	Indenting makes the code easier to read as it shows the limits of IF, DO etc, and will also make incorrect coding easier to spot (and easier to avoid in the first place).
4 Use different (but concise and explanatory) error messages. A reference to the code module could also be useful.	Some students simply code 'Error' for error messages. A message like this is not at all useful, either to the user, or to the programmer trying to find out which piece of code generated the error.
5 Code one task per procedure.	A procedure with many tasks can get very complicated. If a task is broken down into procedures, each procedure can be tested separately and the main procedure will be easier to code, debug and understand). Some of the sub-procedures may be useful elsewhere as general procedures (code re-use).
6 Write re-usable procedures where possible.	A little time spent on planning (for example, a common message procedure) can save a great deal of time later in testing, debugging, coding and maintenance. Code is also consistent and easier to follow.
7 Do not duplicate code. (if you find you are using 'cut and paste' to copy bits of existing code into a new procedure, then this probably means you need to use a re-usable procedure).	Duplicated code makes more work (with more possibility for errors) for the developer, and can cause problems with maintenance as the maintainer must search for (and may miss) all occurrences of the code. See point 6 above
8 Use 'Private' for forms and reports, unless you know that the procedures, variables, constants etc are also going to be used elsewhere.	Saves memory. Avoids the problem of accidentally coding duplicate names.
9 Put object-specific code in the appropriate module.	For example, put all code specific to a particular form's events in the code module for that form.
10 Use 'Public' for procedures, variables, constants etc that are for general use. These will normally be coded in an Access module. (See also Appendix C, point 8).	This is what 'Public' means, and is what Access modules are for. A maintenance programmer will look in Access modules to see what standard procedures, variables, constants etc are available.
11 Prefix procedures and general variables/constants with a code identifying the module and/or the type.	This can help both developer and maintenance programmer to identify the module in which these are coded and the datatype. For example, names of VBA constants are all prefixed with 'vb'. You could prefix your own constants with 'con' or 'mycon'. More examples are given in Appendix E.
12 Code each new statement on a new line.	This makes code much easier to read and maintain. Do not code IF ... ELSEIF ... all on one line.

Appendix C – Common Coding Errors

Error ¹	Likely Result
1 Using a data name, procedure ² name, etc. that you have not defined or declared.	Compiler will report that the data item or procedure has not been defined. Use Option Explicit – see section 1.2.
2 Typing errors, spelling mistakes in data names, procedure names.	Compiler will report that the data item or procedure has not been defined. See also error 1 above & 10 below.
3 Embedded spaces in names. For example, typing Last Name instead of LastName Or [Last Name] for a field.	This could cause several errors, depending on context. <ul style="list-style-type: none"> • Put square brackets [] around table and field names • Put “ at either end of non-numeric literals • Define names with no spaces (LastName) or underlines (Last_Name).
4 Forgetting to code an End statement.	The compiler will normally point out that an End is missing. Check your logic carefully to work out where it should go. If you indent your code correctly you should remember to code End statements where appropriate.
5 Missing or incorrect punctuation.	<ul style="list-style-type: none"> • Commas should separate arguments in procedure and function headings. • Double quotes should surround non-numeric literal values in expressions. • Code Rem or an apostrophe (') in front of comments. • Code a colon (:) after label names. See error 10 below.
6 Forgetting to repeat the subject name in a compound if.	It may seem logical to code something like If intValue > 5 and < 12 Then... but VBA, like many other programming languages, expects If intValue > 5 and intValue < 12 Then...
7 Using a private procedure outside the module in which it is defined (or in the debugger – see section 1.4.3.3).	You may get one of the following error messages: Run Time Error 2465 – Object-defined Error. OR Sub or Function not defined. If you really want the procedure to be used outside the module, then change it to “Public”. If it is to be a general-purpose procedure, then move it to an Access module. Change it to Public to test via the Debugger, then back to Private if necessary.
8 Forgetting to specify the code module for a form or report public procedure declared elsewhere.	Suppose you had a Public procedure DoThis coded in the module for FormA. In order to reference this procedure from another module, you need to code ... Forms!FormA.DoThis ... This is another use of the forms collection – see Appendix I.
9 Having more than one data item or procedure with the same name.	You should get “Compiler Error: Ambiguous Name detected”. If <u>variables</u> with the same name are coded in different procedures within the same module, then the scope rules will come into play. See Appendix F1.3.
10 Compiler error: ‘Sub or Function not defined’	You are attempting to call a non-existent procedure or function. Some common reasons are: <ul style="list-style-type: none"> • You have misspelled the procedure name • The procedure is defined as Private • You have omitted the colon (:) after a label name.

¹ Note that the compiler will check only for syntax (grammar) errors – i.e. to see that each line (sentence) conforms to the rules of the language. It does not check that the whole code makes logical sense.

² Reference to ‘data items’ includes variables, constants, and objects on forms/reports. Reference to ‘procedures’ includes subs and functions.

Appendix D – Code Documentation Form

CODE MODULE DOCUMENTATION FOR _____ DATABASE

TYPE*: Form / Report / Access NAME _____ Page of
 * delete or circle as appropriate

Object	Type	Event	Name	Public	Brief description

Prepared by _____ date _____

Appendix D – Code Documentation Form

Samples of completed form, showing the procedures and functions developed during Part 1 of this Trainer.

CODE MODULE DOCUMENTATION FOR		<i>Chelmer Leisure Centre</i>		DATABASE	
TYPE*: Form / Report / Access		NAME <i>Membership Category</i>		Page 1 of 1	
* delete or circle as appropriate					
Object	Type	Event	Name	Public	Brief description
<i>Test fee changes button</i>	<i>Command button</i>	<i>Click</i>	<i>cmdTestChanges_Click</i>		<i>Runs query Check Update Fee to show result of applying calculations</i>
<i>Make Fee changes button</i>	<i>Command button</i>	<i>Click</i>	<i>cmdMakeFeeChanges_Click</i>		<i>Updates the Membership Category table</i>

CODE MODULE DOCUMENTATION FOR		<i>Chelmer Leisure Centre</i>		DATABASE	
TYPE*: Form / Report / Access		NAME <i>Calculations</i>		Page 1 of 1	
* delete or circle as appropriate					
Object	Type	Event	Name	Public	Brief description
<i>N/a</i>	<i>N/a</i>	<i>N/a</i>	<i>myUpdateFee</i>	<i>Yes</i>	<i>Calculate new fee for given row of membership category table</i>

CODE MODULE DOCUMENTATION FOR		<i>Chelmer Leisure Centre</i>		DATABASE	
TYPE*: Form / Report / Access		NAME <i>Messages And Questions</i>		Page 1 of 1	
* delete or circle as appropriate					
Object	Type	Event	Name	Public	Brief description
<i>N/a</i>	<i>N/a</i>	<i>N/a</i>	<i>myDisplayInfoMessage</i>	<i>Yes</i>	<i>Takes and displays a given message</i>
<i>N/a</i>	<i>N/a</i>	<i>N/a</i>	<i>TestDisplayInfoMessage</i>		<i>Used to test the above procedure</i>
<i>N/a</i>	<i>N/a</i>	<i>N/a</i>	<i>myYesNoQuestion</i>	<i>Yes</i>	<i>Takes and asks a given question, returning user's Yes or No reply</i>
<i>N/a</i>	<i>N/a</i>	<i>N/a</i>	<i>TestYesNoQuestion</i>		<i>Used to test above function</i>

Appendix E – Some naming conventions for variables, procedures etc.

It is good programming practice to prefix names in code with identifying characters. This does not normally apply to names of items in tables (and the corresponding fields on forms and reports).

A list of some commonly used prefixes is shown below. This Trainer has used some of these in the code. Some of the list below are used in the “Further VBA” Trainer.

The prefix usually indicates the type and the rest indicates the purpose of the item. Prefixes can be combined (but can then lead to rather unwieldy names).

Item / Type	Prefix(es)	Example/comment
Access action constant	ac	acPreview (see wizard code for previewing a report)
ActiveX object	ocx	ocxCalendar in section 5.6
Boolean	bool, b	bNoData in section 5.5.2
Combo Box	cbo	cboFindName in section 3.4.2
Command Button	cmd	cmdClose in section 1.1.3
Constant	const or con	myconChelmerName in section 1.7.1, conBkgStartTime in section 6.6.
Currency	cur	curPrice in F.2.2
Database	db, dbs	dbsTheDatabase in section 3.2.1 of “Further VBA” Trainer V5.0
Date	dt or dte	dtBkgTime in section 6.6
Field	fld	
Form	fm, frm	
Index	idx	idxCounter in Appendix F.3.3.
Integer	l or int	intYear in section 3.3.3.1
List Box	lst	lstNumbers in section 6.5
Page in tab control	pg	PgPersonal in section 7.2.
Parameter	prm	prmFee in section 1.4.1.
Private	priv	Use to specify if own procedures or constants are restricted to the module in which they are coded
Public	pub	Use to specify if own procedures or constants are intended for general use. The function myYesNoQuestion could have been called mypubYesNoQuestion.
RecordSet	rs, rst	rstTest in section 3.2.4 of “Further VBA” Trainer V5.0
String variable	str, st	strName in section 3.5.1, stDocName in section 1.6
TableDef	td, tdef, tbdef	
Text Box (unbound)	txt	txtValue in section 3.2.2.1
Variant datatype	v, var	VCourt1Array in section 4.3.1 of “Further VBA” Trainer V5.0
VB own constants	vb	vbRed in section 3.2.2.2, vbInformation in section 1.7.1
Workspace	ws	
Your own items	my	myYesNoQuestion in section 1.7.2. (or mypubYesNoQuestion) myconChelmerName in section 1.7.1 (or mypubconChelmerName) This prefix can be useful to alert maintenance programmers to the fact that the item being used is not a standard Access item.

Appendix F – Some Basics of Programming

The purpose of this document, as stated in the Preface, is not to teach you how to program. The best way to learn how to program, just like the best way to learn a human language, is to practice it as often as you can. The more you do, the better you should get.

This Appendix discusses briefly several basic concepts which are also applicable to programming in general. An understanding of these will help you with programming, whatever language you use.

There are two main things that you need to be able to do in order to program:

- You need to be familiar with the language that you are using, how it works and the tools that the language provides for you. This Trainer tries to provide you with some of that information, and to equip you with the skills necessary to find out more.
- You need to appreciate just what it is that you trying to do, and the logic that is necessary for you to achieve it. In my experience, it is this aspect of programming that students normally find difficult. Computers run the code that you have written, not what you think you have written. You should plan what you want to do on paper first, and dry-run it, before getting anywhere near a keyboard; only very good and/or very experienced programmers can work things out at a keyboard (and even then, they can get it wrong).

F.1 Declaring and using variables and constants

F.1.1 Datatypes

Whenever you declare a variable or a constant, an area of memory is reserved for you, and you refer to it by the name you have chosen in the definition. The programming language converts this name to an address in order to reference the location in memory.

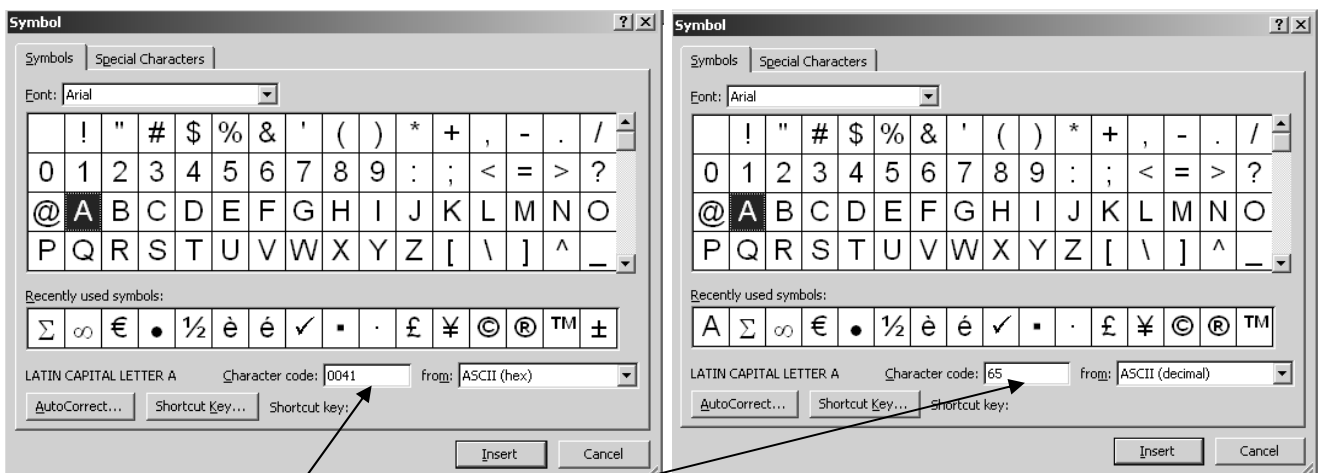
All values in datatypes (and indeed, all content of bytes in memory, including code) are held in binary, a sequence of noughts and ones. Early programmers actually coded in binary, and later programmers needed to know how to interpret stores dumps printed out in hexadecimal (when a program failed, the entire content of memory was 'dumped' out to a printer, in hexadecimal, and programmers had to know how to find their way around this dump and be able to do hexadecimal arithmetic, as well as recognise instruction code). These days, with modern Debuggers, the underlying binary code can be hidden from view and many students forget how data is actually stored.

Consider a byte (8 bits) with the binary content shown here:

0100 0001

What does this represent?

The following dialog boxes are from Word Insert→Symbol:



Character code (hex and decimal)

Appendix F – Some Basics of Programming

Hexadecimal 41 = binary 0100 0001 = decimal 65

Giving an area of memory a datatype is needed to inform the processor what the content is intended to represent.

A byte containing 0100 0001 could therefore represent...

- ...the letter 'A' if it has a String datatype.
- ...the (decimal) number 65 if it has a Byte datatype.

Datatypes also specify the length (in bytes) of the variable or constant. Some are fixed for the datatype; see Access Help for datatypes used for tables, and VBA Help for details about the Dim statement. Note that the **Variant** datatype is the default datatype if no datatype is specified. All textboxes on forms and reports are Variant. A Variant datatype is the only datatype that can take a Null value.

F.1.2 Variables and constants

A **variable** is a named area of memory whose contents can change at run-time. Example:

- stDocName in section 1.6, used by the RunQuery wizard code to store the name of the query.
- intAge in section 3.2.3.1 used to store the result of the calculation of a person's age.

A **constant** is a named area of memory whose contents are fixed, and cannot be changed by code (or anything else) at run-time. If you attempt to assign a value in code, you will get the compile error: "Assignment to constant not permitted". If you want to change the contents of a constant, then you must change the code via a code window; the new value is then fixed for all future uses of the application. Example:

- myconChelmerName in section 1.7.1, used to get a standard heading for forms, reports and messages.

Access VBA also has its own list of constants, some of which (vbRed, vbInformation) you have already seen in this Trainer. For fuller information check VBA help with the keyword *constant*.

F.1.3 Scope

Variables and constants are referred to by name. Scope rules determine, amongst other things, which variable/constant is being referred to if there are two or more with the same name.

If a variable is called, for example, intCounter, in two different procedures, then each procedure has its own **local** variable called intCounter, and the two are recognised as being different variables and thus different areas of memory. They exist only within their own individual procedure.

If a module has a **global** variable called intCounter (declared at the head of the module, outside any procedure) it is available for use by all procedures in that module (and, if declared as public, by other procedures as well).

If a module has a **global** variable called intCounter and a procedure within that module also has a **local** variable called intCounter, then the code will compile OK, but, at run-time, the code inside the procedure will use the local variable in preference to the global variable. This may or may not be what you want.

It is thus good practice to use different names for global and local variables. The compiler will not get confused if the names are the same, but you might! It could perhaps be useful to use a prefix for global variables (for example globintCounter) to avoid confusion (yours, that is).

Similar rules apply when naming procedures (sub or function), as these are also referred to by name. This is one reason why this document suggests that procedures and public constants/variables that you write for yourself are prefixed (see Appendix E).

For further information see VBA Help Answer Wizard with the keyword *scope*.

Appendix F – Some Basics of Programming

F.1.4 Public/Private

Items declared as **Public** are available for use by code in other modules.
Items are **Private** by default, available for use within their own module only.

For further information use the VBA Help Answer Wizard with the keywords, *Public* or *Private*.

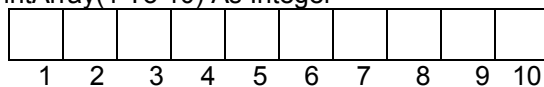
F.1.5 Arrays

In this Trainer you have seen the use of many different datatypes. One other very useful datatype (used in Part 4 of the Further VBA Trainer v5.0) is **array**.

An array is simply a set of elements of the same datatype, but this is a concept that some students find difficult to grasp. You may find it useful to visualise an array as a set of pigeonholes.

For example, the following is a declaration and a visual representation of an array of 10 integer elements:

```
Dim intArray(1 To 10) As Integer
```



Note:

*Dim intArray (1 – 10) As...
will NOT compile.
You must use 'To', not a hyphen.*

Each element in the array is referred to by the array name and the element number (known as the array index or subscript). Examples:

```
intArray (3) = 1 'put the value 1 in element number 3
```

```
Dim idxCounter as Integer
```

```
idxCounter = 3
```

```
intArray(idxCounter) = 1 'put the value 1 in the element indexed by idxCounter (3)
```

The index value for an array must be numeric, but does not need to start from 1. You can use a range that makes more sense in the context. For example, suppose we want to count up the bookings made in the years 1996 to 2000 inclusive:

```
Dim intBkgArray(1996 To 2000) As Integer
```

```
intBkgArray(Year(dtBookingDate)) = intBkgArray(Year(dtBookingDate)) + 1 'add 1 to total
```

This code assumes that:

- The booking date from the record is in a date/time variable called dtBookingDate
- The booking date has been validated to have a year in the range required for the array.
 - o If you attempt to reference an element that does not exist (intBkgArray(2001) for example) then you will get a run-time error as the index (subscript) is out of range.

You can also use constant variables for an array range:

```
Const intStartYear = 1996
```

```
Const intEndYear = 2000
```

```
Dim intBkgArray(intStartYear To intEndYear) as Integer
```

The Debugger is especially useful for looking at values in arrays, as you see each element separately and watch as the code executes. See the screenprint in section 4.3.4 of the Further VBA Trainer v5.0.

The Lucky Numbers example database on <http://www.cse.dmu.ac.uk/~mcspence/Access> shows how to use an array of 49 elements to check that a random lottery number in the range 1 – 49 has not been chosen already.

Appendix F – Some Basics of Programming

F.2 Assignment statements

F.2.1 General format

Assignment statements are statements that put (assign) a value in a variable, constant, property, the return value for a function, field on form/report, etc. The general format of an assignment statement is:

Destination = {Value | Expression | Name}

- The destination is always on the left of the statement.
- { } means choose one from the list
 - o 'Value' means a literal value used in the code
 - o 'Expression' means the result of a formula
 - o 'Name' means the value in another variable or constant

For example:

```
Dim intCounter as Integer
Const conStartValue = 1      'assign value 1 to a constant – example using numeric literal value
intCounter = 0              'set to zero – example using numeric literal value
intCounter = intCounter + 1  'add 1 to value already in intCounter – example of expression
intCounter = conStartValue   'copy value from another variable/constant
```

When assigning values, you must be careful to ensure that the datatypes on each side of the statement are compatible. For example, you cannot assign a non-numeric value to a numeric variable.

For further information look at the VBA Help Answer Wizard with the text *assignment statement*.

F.2.2 Literals

A literal is a value that you specify directly in code.

In the examples in section F.2.1:

```
intCounter = 0      '0 is a numeric literal
intCounter = intCounter + 1  '1 is a numeric literal
```

Numeric literals are coded simply as numbers. They do not have to be integers:

```
Dim curPrice as Currency
curPrice = 5.5      'puts £5.50 in curPrice
```

String literals must be enclosed in quotation marks, to avoid confusion with variable/constants that may have the same name:

```
Dim Test as String
Dim strName as String
strName = "Test"      'puts the string "Test" into the variable strName
strName = Test        'copies contents of the variable called Test into the variable strName
```

Date/Time literals must be enclosed in hash (#) marks:

```
Dim dtDate as Date
Dim dtTime as Date
dtDate = #10/09/2004#  'be careful – this means 9th October 2004!
dtDate = #09 Oct 2004# 'will be changed by VB to #10/09/2004#
dtTime = #14:34#      'will be changed by VB to #2:34:00 PM#
```

VBA Help says: "You must use English (United States) date formats in SQL statements in Visual Basic. However, you can use international date formats in the query design grid". See section 8.3.3. It looks like you must also use USA formats in assignment statements.

Appendix F – Some Basics of Programming

F.3 Control Constructs

There are three basic control constructs (ways of constructing the logic and code) used within code:

- Sequence
 - o A simple statement after which control is passed to the next statement.
- Selection
 - o Where code branches to different statements, depending on some condition
- Iteration
 - o Iterating (repeating, looping) code depending on some condition.

F.3.1 Sequence

There are many examples in this Trainer of **sequences**, so they will not be discussed further here.

F.3.2 Selection

Examples of selection are **IF** and **CASE**.

The syntax of an **IF statement** (taken from VBA help) is:

The items in [] are optional.

If <i>condition</i> Then [<i>statements</i>]	'executed if <i>condition</i> is true
[Elself <i>condition-n</i> Then [<i>elseifstatements</i>] ...	'executed if all earlier conditions are false 'executed if <i>condition-n</i> is true
[Else [<i>elsestatements</i>]]	'executed if all earlier conditions are false – will catch all other conditions
End If	'terminates the statement

IF statements can be nested within each other.

For some examples in this Trainer, see sections 1.7.2, 2.5.2 and 3.2.2.2.
Section 3.2.3.1 has an example of a nested IF statement.

A **CASE statement** is way of coding a selection when you are checking the same variable several times. It can be neater and clearer to follow than using a lot of Elself clauses.

The syntax of a CASE statement (taken from VBA help) is:

The items in [] are optional.

Select Case <i>testexpression</i>	'the item being checked
[Case <i>expressionlist-n</i> [<i>statements-n</i>]] ...	'the check for each expression 'executed if <i>expressionlist-n</i> is true
[Case Else [<i>elsestatements</i>]]	'executed if all earlier conditions are false – will catch all other conditions
End Select	

There is an example in section 1.4.1.

Case statements can be nested within each other.

Appendix F – Some Basics of Programming

F.3.3 Iteration

Iteration is simply another word for “looping”. It is often required that a block of code is repeated until a certain condition is True. There are several types of loops in VBA, some of which are discussed briefly here. They can be nested within each other.

The syntax below is taken from VBA Help.

The items in [] are optional.

Items specified as {A | B | C ...} means choose ‘choose one of A, B, C...’.

Do ... Loop: Looping while or until a condition is True

Do [{While Until} <i>condition</i>] [<i>statements</i>] [Exit Do] ‘use for early exit [<i>statements</i>] Loop ‘end of the loop block	or	Do [<i>statements</i>] [Exit Do] [<i>statements</i>] Loop [{While Until} <i>condition</i>]
--	----	---

There are examples of a Do ... Loop in sections 6.5 and 8.3.5.

While ...Wend: Executes a series of statements as long as a given condition is True.

While <i>condition</i> [<i>statements</i>] Wend

There is no Exit statement allowed for this format. The Do...Loop statement provides a more structured and flexible way to perform looping. There is no example of this format in this Trainer.

For ... Next: Using a counter to run statements a specified number of times. It is particularly useful with arrays.

For <i>counter</i> = <i>start</i> To <i>end</i> [Step <i>step</i>] [<i>statements</i>] [Exit For] ‘use for early exit [<i>statements</i>] Next [<i>counter</i>]	‘If Step is missed off, then 1 is assumed.
---	--

There is an example of a For ... Next loop in section 6.6.

The following is an example of code to initialise all elements of an integer array to zero:

```
Dim intArray(1 To 10) As Integer
Dim idxCounter as Integer
For idxCounter = 1 to 10
    intArray(idxCounter) = 0
Next
```

At the end of the loop, the value in idxCounter will be 11.

Appendix F – Some Basics of Programming

If your loop condition is never met, or you do not Exit as expected, then your loop will never end. You will have what is known as an **infinite loop**.

You will have to terminate execution by one of:

- Esc
- Break
- Cntrl-Alt-Delete, and then cancel the task in the Windows Task Manager (and lose any unsaved changes!).

It is probably best to test all loops within the Debugger, so you can check what is going on, and stop (without losing anything) when you want to.

For further information on loops, look up *loop* in VBA Help.

F.4 Procedures and Parameters

A procedure is a piece of code that you can call to perform a certain task. As you will have seen in this Trainer, all event code is contained within a procedure. See also section 1.1.2.

There are various built-in procedures (mainly functions) within VBA; see Appendix H. You can also create your own, as seen in sections 1.4.1 and 2.4.1.

You may find it helpful to think of calling a procedure as subcontracting the work that you want to be done to that procedure; you do not necessarily need to know how the work is done, only what is done, what you need to provide (as parameters) to the procedure and what the procedure will return to you (as parameters or a returned value). Every procedure has a procedure heading which tells you how to call it:

Private sub mySub1()

- This is a **sub** procedure.
- It is **private** so can only be used from within the module in which it is written.
- Call it by coding: `mySub1`
- There are no parameters

Public sub mySub2(byVal prmData1 as Integer, optional prmData2 as String = "default")

- This is a sub procedure
- It is **public**, so can be called from other modules.
- It requires two **parameters** – the documentation should tell you whether these values are provided by the calling code and/or are returned/changed by the sub.
- The first parameter is passed **byVal**, which means that only a copy of the value is passed; the original location value cannot be altered. By default, all parameters are passed **byRef**.
- The second parameter is **optional**, so can be omitted. The heading specifies a default value.
- Call it by coding: `mySub2 intValue1, strValue2` OR `mySub2 intValue1`
- Note that when calling a sub procedure, the parameters must not be enclosed in round brackets.

Public Function myFunction1(prmData3 as String) as Boolean

- This is a **function** procedure
- It is public, so can be called from other modules.
- It requires one string parameter.
- It **returns** a Boolean value.
 - o The return value can be used in an assignment statement or as part of a condition.
 - `bResult = myFunction1("String Value")` OR `bResult = myFunction1(strValue3)`
 - `if myFunction1("String Value") then...` OR `if myFunction1(strValue3) then ...`
- Note that when calling a function, the parameters must be enclosed within round brackets.

See also section 1.7.1 for a discussion of some basic concepts, with examples.

Use the keywords *Call statement*, *procedure*, *byVal* and *byRef* in VBA Help for further information and examples.

Appendix G – Overview of SQL

SQL is a huge topic, for which there are many textbooks available for those students who wish to explore it further. There are also Trainers available within DMU. This Appendix merely lists the syntax of some SQL statements, for you to reference when using embedded SQL or when creating queries using SQL rather than the Query Design Window.

Tip: The Query Design Window can be useful to create and test SQL to help you with SQL that you want to use in embedded SQL code. You can also simplify the generated SQL by removing unnecessary table qualifications and round brackets.

SQL is not exactly the same for all database software. The general principles may be the same, but each software product may have its own particular dialect. For example:

- The wildcard * used by MS Access is the % character in some other software.
- Crosstab queries may not be available in all database software.
- The TOP clause may not be available in all database software.
- Some other database software may require single quotation marks for strings.
- Access is not fussy about the final semi colon; - other software may insist on it.

Extract from Access 97 Help: **Work with SQL in queries, forms, reports, macros, and modules.**

"You can use SQL, or Structured Query Language, to query, update, and manage relational databases such as Microsoft Access. When you create a query in query Design view, behind the scenes Microsoft Access constructs the equivalent SQL statements. You can view or edit the SQL statement in SQL view. After you make changes to a query in SQL view, the query might not be displayed the way it was previously in query Design view.

Some queries can't be created in the design grid. For pass-through, data-definition, and union queries, you must create SQL statements directly in SQL view."

The rest of this Appendix consists of SQL syntax for you to reference when creating SQL statements. All these extracts are copied from MS Access 97 Help (I can't find them in Access 2000/2002; the VBA Help Answer Wizard with *about SQL queries* will find just a little bit of information.).

- Items in square brackets [] are optional
- Items in curly brackets { } mean that you must choose one from the list
- Three dots ... mean that the item can be repeated (comma-separated).

G.1 CREATE TABLE (to create and define a new, empty, table)

```
CREATE TABLE table (
    field1 type [(size)] [NOT NULL] [index1]
    [, field2 type [(size)] [NOT NULL] [index2]
    [, ...]]
    [, CONSTRAINT multifieldindex [, ...]]
)
```

Access also has a MAKE TABLE query, used to create a table (with data) from the result of running a SELECT...INTO query based on other tables/queries.

Simple example to create a table with a single text column of length 3:

```
CREATE TABLE New (field1 char(3));
```

There are examples of CREATE TABLE SQL statements in section 6.5, and in the Further VBA Trainer.

G.2 ALTER TABLE (to change definition)

```
ALTER TABLE table {
ADD {COLUMN field type[(size)] [NOT NULL] [CONSTRAINT index] | CONSTRAINT multifieldindex} |
DROP {COLUMN field | CONSTRAINT indexname}
}
```

Simple example to add a primary key to the table created in G.1 example above:

```
ALTER TABLE New ADD CONSTRAINT new_pk PRIMARY KEY (field1);
```

Appendix G – Overview of SQL

G.3 CONSTRAINT (to define keys and relationships)

Single-field constraint:

```
CONSTRAINT name {PRIMARY KEY | UNIQUE | NOT NULL |
REFERENCES foreigntable [(foreignfield1, foreignfield2)]}
```

Multiple-field constraint:

```
CONSTRAINT name
{PRIMARY KEY (primary1[, primary2 [, ...]]) |
UNIQUE (unique1[, unique2 [, ...]]) |
NOT NULL (notnull1[, notnull2 [, ...]]) |
FOREIGN KEY (ref1[, ref2 [, ...]]) REFERENCES foreigntable [(foreignfield1 [, foreignfield2 [, ...]])]}
```

Relationships in Access are normally defined via the Relationships Window.

G.4 INSERT (to insert data)

Multiple-record append query:

```
INSERT INTO target [IN externaldatabase] [(field1[, field2[, ...]])]
SELECT [source.]field1[, field2[, ...]]
FROM tableexpression
```

Single-record append query:

```
INSERT INTO target [(field1[, field2[, ...]])]
VALUES (value1[, value2[, ...]])
```

Simple example to add a row to the table New created in G.1:

```
INSERT INTO New ( Field1 ) VALUES ("abc");
```

There are examples of INSERT SQL statements in sections 6.3, 6.5 and 6.6, and in the Further VBA Trainer.

G.5 CREATE VIEW

There does not appear to be a CREATE VIEW statement in MS Access.

Outline syntax is:

```
CREATE VIEW ViewName AS
SELECT....etc – normal SELECT statement
```

When you run an SQL statement in software such as sqlplus, or in an XSQL file, the SQL is run but does not 'exist' anywhere in particular in the database. If you want to use it again somewhere else, you need to copy and paste the code, which is bad practice. So you create an SQL View, which is simply a definition for a virtual table; it does not contain any data, but SQL SELECT statements can be based on a View as though it were a table.

In MS Access, all queries that you create are named and stored within the database. They can then be used from anywhere within the database, and forms and reports are normally based on a query rather than a table. MS Access queries are thus effectively the same as SQL views.

Appendix G – Overview of SQL

G.6 SELECT for inner join

```
SELECT [predicate] { * | table.* | [table.]field1 [AS alias1] [, [table.]field2 [AS alias2] [, ...]] }
FROM tableexpression [, ...] [IN externaldatabase]
[WHERE... ]
[GROUP BY... ]
[HAVING... ]
[ORDER BY... ]
[WITH OWNERACCESS OPTION]
```

An inner join “Combines records from two tables whenever there are matching values in a common field.” This is the default join in a SELECT query, and the one with which you should all be familiar.

The example below shows two methods of coding an inner join SELECT statement, to list membership numbers and category details in the Chelmer Leisure database. Note the different methods of specifying the join between the Membership and Membership Category tables.

Standard SQL (as used by most database software):

```
SELECT [Membership Category].[Category No], [Membership Category].[Category Type],
[Membership Category].[Membership Fee], [Membership].[Membership No]
FROM [Membership Category], [Membership]
WHERE [Membership Category].[Category No]=[Membership].[Category No]
ORDER BY [Membership Category].[Category No];
```

SQL generated by MS Access Query Design Window:

```
SELECT [Membership Category].[Category No], [Membership Category].[Category Type],
[Membership Category].[Membership Fee], [Membership].[Membership No]
FROM [Membership Category] INNER JOIN [Membership] ON [Membership Category].[Category No]
= [Membership].[Category No]
ORDER BY [Membership Category].[Category No];
```

There are examples of SELECT SQL in sections 3.2.4 and 3.4.1.2, and in the Further VBA Trainer.

G.7 UNION

A UNION query is where the results of SELECT queries can be combined. The individual query result datasets must be compatible with each other. You have seen an example of a query of this type in section 1.8.2. See also Appendix G.8.

G.8 SELECT for outer join

The example below shows how MS Access generates an outer join SELECT statement. This is the same as the example in Appendix G.6, but here will select all Membership Categories, even if they do not have any Members. The worked examples in Part 8 use Outer Joins.

```
SELECT [Membership Category].[Category No], [Membership Category].[Category Type],
[Membership Category].[Membership Fee], [Membership].[Membership No]
FROM [Membership Category] LEFT JOIN [Membership] ON [Membership Category].[Category No] =
[Membership].[Category No]
ORDER BY [Membership Category].[Category No];
```

This type of join may not be available in other database software. You will have to code two queries, then join them via UNION using a NOT IN sub query:

- Query 1 – a normal inner join query, as shown in Appendix G.6.
- Query 2 – a query to list all the category details that are not included in Query 1. A NOT IN sub query is used to do this.
- Use UNION to join the result of the two queries.

Appendix G – Overview of SQL

The final result is:

```

SELECT [Membership Category].[Category No], [Membership Category].[Category Type],
[Membership Category].[Membership Fee], [Membership].[Membership No]
FROM [Membership Category], [Membership]
WHERE [Membership Category].[Category No] = [Membership].[Category No]
UNION
SELECT [Membership Category].[Category No], [Membership Category].[Category Type],
[Membership Category].[Membership Fee], "" AS [Membership No]
FROM [Membership Category]
WHERE [Membership Category].[Category No] NOT IN
  (SELECT DISTINCT [Membership Category].[Category No]
   FROM [Membership Category], [Membership]
   WHERE [Membership Category].[Category No] = [Membership].[Category No])
ORDER BY [Membership Category].[Category No];

```

Query 1

Query 2 – note the calculated (empty) field for Membership No

NOT IN sub query

ORDER BY for full set of results

The sub query merely lists (provides the set of) Categories that have members (as a normal inner join will have a result row only where there is a match in both tables). DISTINCT is used (though is not essential here) to get a unique set of rows. A sub query must have only one column in the result. The SQL for the sub query is indented so that it shows clearly, but indenting is not essential.

Query 2 then uses the set of categories who do have members to list those categories that are not in the set by using the NOT IN clause.

G.9 COMMIT

MS Access has a CommitTrans method for use in VBA code when using workspaces. All other changes are automatically committed (saved) with MS Access.

But when using SQL in other software such as sqlplus you must remember to use this command to save all changes made. You will then get the response 'commit complete'.

G.10 GRANT (and REVOKE)

MS Access has its own permissions facility (which I have not explored).

When using sqlplus you must specifically GRANT privileges to other users. You should get the response 'grant succeeded'.

G.11 Aliases for tables

SQL can get rather lengthy when table names are used to qualify table fields, but can be simplified using aliases. The SQL in the OUTER JOIN example in Appendix G.8 can be simplified as follows:

```

SELECT C.[Category No], C.[Category Type], C.[Membership Fee], M.[Membership No]
FROM [Membership Category] AS C,[Membership] AS M
WHERE C.[Category No] = M.[Category No]
UNION SELECT C.[Category No], C.[Category Type], C.[Membership Fee], "" AS [Membership No]
FROM [Membership Category] AS C
WHERE C.[Category No] NOT IN
  (SELECT DISTINCT C.[Category No]
   FROM [Membership Category] AS C, [Membership] AS M
   WHERE C.[Category No] = M.[Category No])
ORDER BY C.[Category No];

```

Appendix H – Built-in Functions

This Appendix merely lists, with brief descriptions and examples, some of the various functions that are built-in to MS Access. References to VBA Help are given for you to find out fuller details, and see just what the examples given here are doing. Try the examples out using the Debugger to see how they work. Some of the functions have been used in this Trainer; check the Index for details.

Note that many (if not all?) of these functions can also be used in MS Access queries and SQL. The type conversion functions are especially useful in query criteria which reference form parameters to ensure that comparisons are done correctly.

H.1 Date and Time functions

VBA Help keywords: *Function; Date, Time, Date and Time*

Function	Description	Example
Date, Now, Time	Returns the system date, date-&-time, time	dtToday = Date
DateAdd	Returns a given date with a specified time-period added to it.	dtDate = DateAdd("d",7,dtDate)
DateDiff	Returns an Integer with the required difference between two dates.	intValue = DateDiff("d", dtDate1, dtDate2)
DatePart	Returns the specified part of the given date as an Integer	intWeekNo = DatePart("ww",Date)
DateValue TimeValue	Returns the given date or time as a string variant	strDate = DateValue(Date)
Day, Hour, Minute, Second, Year, Month, MonthName	Returns an integer that represents the day, hour, etc of the given date. MonthName gives the month name in words (not available in Access 97)	intDay = Day(Date) intYear = Year(Date)
Weekday, WeekdayName	Returns the day of the week, or the day name in words, of the given date (<i>WeekdayName was not available in Access 97</i>).	intWeekDayNo = Weekday(Date) StrWeekDayName = WeekdayName(intWeekDayNo, , vbSunday)

H.2 String functions

VBA Help keywords: *function; string*.

Function	Description	Example
Asc, Chr	Returns ANSI numeric or string value.	intANSI = Asc("A") strANSI = Chr(10)
Format, FormatDateTime, FormatCurrency	Formats an expression in accordance with the format string specified.	strMembNo = Format(intMembKey, "0000") strDate = Format(Date, "dd-mmmm-yy")
InStr	Returns a value indicating the position of the first occurrence of the 2 nd string in the 1 st string.	lngPos = InStr("abcdefgh", "cd")
LCase, UCase	Converts to upper or lower case	strUCase = UCase("aBcD") strLCCase = LCase("aBcD")
Left, Right, Mid	Returns the leftmost, rightmost or middle characters of a string.	strInitial = Left(Forename, 1) strChars = Mid("abcdefghij", 3, 2)
Ltrim, Rtrim, Trim	Removes leading, trailing, or both leading and trailing, spaces from a string	strTrimString = " abc " strTrimString = Trim(strTrimString)
Len	Returns the number of characters in a string.	intInitialsCount = Len(Initials)

Appendix H – Built-in Functions

H.3 Maths functions

VBA Help keyword: *Math*

The full set includes various trigonometric and logarithmic functions not listed here.

Function	Description	Example
Int, Fix	Returns a numeric value with the decimal fraction truncated.	Int(1.5) gives 1 Fix(1.5) gives 1 Int(-1.5) gives -2 Fix(-1.5) gives -1
Rnd	Returns a random number ≥ 0 and < 1 .	Dim MyValue Randomize MyValue = Int((49 * Rnd) + 1)
Sgn	Returns the sign of a numeric value Greater than zero: 1 Equal to zero: 0 Less than zero: -1	intSign = Sgn(Number) <i>(Note – the values returned are different in Access 97!)</i>
Sqr	Returns the square root of a number. <i>There is no function to calculate the square of a number. Can you work out why?</i>	sglRoot = Sqr(Number) <i>(Error: “invalid procedure call or argument” if Number is negative)</i>
Round	Rounds a value to a specified number of decimal places	curInterest = Round(curInterest,2) <i>Not available in Access 97.</i> See also the end of section 1.4.1.

H.4 Financial Functions

VBA Help keyword: *Annuity, or Rate of Return, or Balance, or Interest*

Function	Description (taken from VB and VBA in a Nutshell, by Paul Lomax, Published by O’Reilly)
DDB	Returns double-declining balance depreciation of an asset for a specific period
FV	Returns the future value of an annuity
Ipmt	Returns the interest payment for a given period of an annuity
IRR	Returns the internal rate of return for a given period of an annuity
MIRR	Returns the internal rate of return for a series of periodic cash flows
Nper	Returns the number of periods for an annuity
NPV	Returns the net present value of an investment
Pmt	Returns the payment for an annuity
PPmt	Returns the principal payment for a given period of an annuity
PV	Returns the present value of an annuity
Rate	Returns the interest rate per period for an annuity
SLN	Returns the straight-line depreciation of an asset for a single period
SYD	Returns the sum of years’ digits depreciation on an asset

Appendix H – Built-in Functions

H.5 Miscellaneous functions

Use each name as the keyword in VBA Help

Function	Description	Example
IsDate	Returns true if the expression can be converted to a date.	strTest = "15 Mar 2004" If IsDate(strTest) Then MsgBox "It's a date" End If
IsEmpty	Returns TRUE if a variant datatype has not been initialised	boolCheck = IsEmpty(vVar)
IsNull	Returns TRUE is an expression evaluates to NULL	If IsNull([Date of Birth]) Then... <i>Note that coding If Fieldname = Null then ... will not work! See section 3.2.3.2.</i>
IsNumeric	Returns TRUE if an expression can be evaluated as a number.	boolCheck = IsNumeric(strVar)
Beep	Sounds a note	
Iif	Returns one of two values based on a Boolean expression	See page 134 of McBride and section 3.2.3.2.
MsgBox	Returns the user selection of a choice of buttons	intResponse = MsgBox("Are you sure?", vbYesNoCancel)
InputBox	Returns user input from a simple dialog box	strReply = InputBox("Is anyone there?")
Str	Returns a string version of a number	strValue = Str(459)
Val	Returns the numbers contained in a string	intValue = Val("24 57")
Nz	Use to convert a Null value in a variant to another value	Nz(Me![cboFindName], 0) returns 0 if the value for cboFindName is Null.

H.6 Domain Aggregate functions

VBA Help keywords: *Function; Aggregate*

A domain is a set of records defined by a table, a query or an SQL expression.

Domain Aggregate functions return statistical information about a specific domain or set of records.

They are the VBA equivalent of an SQL SELECT statement that returns a single value, but are for use in code. See Sections 3.2.4 and 3.4.1.2 for examples.

If you misspell a table or field name you may get the error message "You cancelled the previous operation". Not the most helpful of messages!

Function	Description (from Access VBA Help)	Example
DAvg	Calculates the average of a set of values in a domain	DAvg("[Membership Fee]", "Membership Category")
DCount	Determines the number of records in the domain	DCount("[Membership No]", "Membership")
DLookup	Gets the value of a particular field from the domain	DLookup("[Lastname]", "Membership", "[Membership No] = 3")
DFirst, DLast	Returns a random record from a particular field	DFirst("[Lastname]", "Membership", "[Membership No] = 3")
DMin, DMax	Determines the minimum and maximum values in a domain	DMax("[Membership Fee]", "Membership Category")
DStDev, DStDevP	Estimates the standard deviation across a set of values	
DSum	Calculates the sum of a set of values in a domain	DSum("[Stock] * [Unit Price]", "Stock Levels")
DVar, DVarP	Estimates variance across a set of values	

Appendix H – Built-in Functions

H.7 Type Conversion functions

Use phrase in VBA Help Answer Wizard: *type conversion functions*

Each function forces an expression to a specific datatype, and is used in the format:

FunctionName(expression)

Run-time errors will occur if the expression cannot be converted to the required datatype, so appropriate validations may need to be done first.

Function	Description	Example
CBool	Returns a Boolean value for the expression.	CBool(Weekday(Date()) = vbMonday) <i>This will return True if today is a Monday, False otherwise</i>
CByte	Returns a single byte value for a number in the range 0-255.	Dim byNumber As Byte byNumber = CByte(InputBox("Enter number in range 0-255"))
CCur	Returns a value in the range for a currency data type.	Const conValue = "5.6" Dim curValue As Currency curValue = CCur(conValue)
CDate	Converts a date or time expression to a date/time data type.	Const conDate = "15 Jan 2005" Dim dtDate As Date Const conTime = "16:45" Dim dtTime As Date dtDate = Cdate(conDate) dtTime = Cdate(conTime) if CDate(txtDate) > Date() then ...
CDbl	Converts to a Double datatype.	
CDec	Converts to a Decimal datatype.	
CInt	Converts to an Integer datatype.	Const conValue = "5.5" Dim intValue As Integer intValue = CInt(conValue) <i>Fractions are rounded. >= 0.5 is rounded up. < 0.5 is rounded down.</i>
CLng	Converts to a Long datatype.	Fractions are rounded.
CSng	Converts to a Single datatype.	
CStr	Converts to a String Datatype. Result will depend on the value in expression. Some examples of possible results are shown here; for fuller details see Help.	Dim strResult as String strResult = CStr(CBool(Weekday(Date) = vbMonday)) <i>'get True" or "False"</i> strResult = CStr(Date()) <i>'get system date in short date format, e.g. "23/10/2004"</i> CStr(#2/1/2004#) <i>returns "01/02/2004"</i> CStr(#1 Feb 2004#) <i>returns "01/02/2004"</i> <i>See also Format function in Appendix H.2.</i> strResult = CStr (123) <i>' get "123"</i>
CVar	Converts to a Variant datatype.	

Appendix I – The Forms Collection

Look at VBA Help with the keywords *form*; *collection*. Useful items to read are those entitled:

- Forms Collection
- Form Object

Although the information about this Collection does not appear to be in Access Help, it is worth noting that Forms Collection references can be used in queries, as shown in section 5.6.

This Collection is extremely useful and used throughout this Trainer (see sections 3.2.4, 3.5, 4.3.8, 5.6 and 6.4 for some examples) and the “Further VBA” Trainer.

You can reference a **field** or **property** on an open form from another form, a report or a query by:

`forms![form name]![field name]` or `forms![form name].property`

and for sub forms

`forms![form name]![sub form name]![field name]` or `forms![form name]![subform name].property`

also

`forms![form name]![sub form name1]![sub form name2]![field name]` etc

Examples:

- in an assignment statement: `variablename = forms![form name]![field name]`
- changing a field property: `forms![form name]![field name].Visible = False`
- changing a form property: `forms![form name].TimerInterval = 0`

The form name can also be in a variable:

- in an assignment statement: `variablename = forms(strFormName)![field name]`
- changing a field property: `forms(strFormName)![field name].Visible = False`
- changing a form property: `forms(strFormName).TimerInterval = 0`

To reference a **page on a form tab control** (see Section 7.2) you need to use the page name as well as the form name:

`strStreet = Forms![Membership With tabs]!Street`

`Forms![Membership with tabs]!PgPersonal.Visible = False`

As well as referencing fields on an open form, you can also reference **public procedures** within a module on that form.

Suppose you have created a public sub procedure `myPublicSub` in the code module for your Chelmer Leisure Main Menu. This can be called from another code module by:

`Forms![Chelmer Leisure Main Menu].myPublicSub`

The Chelmer Leisure Main Menu form must be open, or this reference will fail at run-time.

If a procedure is to be used as a public procedure, then it would be best to declare it in a separate Access Module.

Appendix J – Some useful DoCmd Methods

The DoCmd Object has a great many useful methods, with associated actions, some of which have been used in this Trainer. For full details of each action, and for the list of all DoCmd actions, see VBA Help with the keyword *DoCmd*.

The table below has a list of some DoCmd actions, taken from VBA Help. These include actions that you will see in generated Wizard code, and have seen in various parts of this Trainer.

Code: DoCmd.ActionName.... (then follow the guidance in the code pop-up box and in VBA Help).

ActionName	Brief Description
AddMenu	Create custom or global menu bars or shortcuts.
ApplyFilter	Applies a filter to records for a form. (See also ShowAllRecords action)
CancelEvent	Cancels an event. See Help for list of applicable events.
Close	Closes a window. A common use is to close a form.
CopyObject	Copies the specified object to the current or a different, database. Could be useful for creating backups.
Beep	Sounds a Beep. (Though seems to work on its own, without having to use DoCmd?)
DeleteObject	Deletes the specified object.
FindNext	Finds the next record that meets the criteria.
FindRecord	Finds the first record that meets the criteria.
GoToControl	Moves the focus to the specified field or control.
Maximise	Maximises the active window. (see also Restore action)
Minimise	Minimises the active window. (see also Restore action)
OpenForm	Opens the specified form in the required mode, applying an optional filter.
OpenReport	Opens the specified report in the required mode, applying an optional filter.
OpenQuery	Runs the specified query.
OutputTo	Outputs the data in the specified format, e.g. to an MS Excel spreadsheet. (See also TransferSpreadsheet).
Quit	Exits MS Access.
Requery	Updates the data in a specified control by requerying the source of the control.
Restore	Restores a maximised or minimised window to its previous size.
RunSQL	Runs SQL for an action or a data definition query.
SendObject	Creates an email message where you can send the selected object, or a message, or both, to specified recipients.
SetWarnings	Turns the display of system messages on/off. See section 6.2.
ShowAllRecords	Removes any applied filter, e.g. for a form.
ShowToolbar	Displays or hides a built-in toolbar or a custom toolbar.
TransferDatabase	Imports, exports or links data between the current database and another database
TransferSpreadsheet	Imports, exports or links data between the current database and a spreadsheet file.
TransferText	Imports, exports or links data between the current database and a text file.

INDEX

<u>Item</u>	<u>Section</u>	<u>Item</u>	<u>Section</u>
#, for date and time	See Date/Time datatype	C	
&, to concatenate items	1.7.1, 2.6.2, 3.5.1, 3.5.2, 3.5.3, 3.5.4, 6.3, 6.4, 6.5, 6.6	Calculations, on a form	3.2
_, to continue code over lines	2.6.2, 4.3.9, 6.3, 7.3.5	Calculations, on a report	5.4
A		Calendar control	5.6
Accelerator keys	4.2.7	Call statement	App F.4
acDataErrAdded constant	6.3	Cancel, event parameter	2.5.2, 3.3.1, 3.3.2, 3.3.3.1, 5.5.1, 5.8
acDataErrContinue constant	6.3	Caption property	2.2.1, 5.2, 8.2.6
acDataErrDisplay constant	6.3	CASE statement	1.4.1, 1.9.2, App F.3.2
acExport property	7.3.1	CDate function	3.3.2, App H.7
acForm constant	5.6	CInt Function	3.3.3.1, App H.7
acFormAdd method	4.3.6	Clear method	3.6.4
acFormEdit constant	4.3.4	Click event	1.6, 2.2.2, 3.5.1, 6.6
acFormReadOnly constant	4.3.5	Close method	4.2.5, 5.6, 6.5, App J
acImport property	7.3.1	CloseCurrentDatabase method	4.2.5
acNewRec constant	2.6.1	Coding standards	App B
acViewPreview constant	5.6	Colour Constants	2.4.1, 2.4.3
Adding records	2.6.1	Colours, defining your own	See RGB function
AddItem method	3.6.4	Column property	3.4.2, 3.6.1, 8.2.7, 8.3.2
AfterUpdate event, calendar control	5.6	ColumnCount property	3.6.1
AfterUpdate event, combo/list box	3.4.2, 3.6.3	ColumnHeads property	3.4.2, 3.6.1
AfterUpdate event, form	2.5.3, 3.4.1.2	ColumnWidths property	3.4.2, 3.6.1
AfterUpdate event, text box field	3.2.2.3, 3.2.3.2, 3.2.4, 3.3.1, 3.3.3.1, 3.3.3.2, 5.6, 6.4	Combo box to find record	3.4.2
Age calculation	See myCalculateAge	Combo box, add to list at run-time	6.3
Aliases in SQL	App G.11	Combo Box, using	3.4.2, 3.5.2, 3.5.3, 3.6, 5.7, 6.4
AllowEdits form property	2.2.1, 2.3, 2.5.2, 2.6.1, 3.4.2, 3.5.1, 4.3.8	Command button	1.6, 2.2.2, 2.3, 2.5.1, 2.6, 2.7.2, 3.2.2.3, 3.4.1.1, 4.2.3
ALTER TABLE SQL	App G.2	Command button, hyperlink	2.4.3
ApplyFilter method	3.5.1, 3.5.2, 4.2.1, App J	Command button, properties	2.4.4
Argument	See parameter	Command button, raised/sunken	2.4.2
Array	App F.1.5	Command button, text colour	2.4.1
Assignment statement	1.4.1, 3.2.2.3, 5.4, App F.2	Comments in code	1.4.1
AutoExpand property, combo box	3.4.2	COMMIT SQL	App G.9
AutoNumber key on SQL	6.6	Compacting	7.4
B		Compiling code	1.4.2
BackColor property	2.4.2, 3.2.2.2, 3.4.2	Concatenation	See &
BackStyle property	2.4.2, 3.2.2.1	Conditional formatting	3.2.2.2, 8.4.5
Backup	7.4	Confirmation messages	2.6.2, 6.2
BeforeUpdate event, field	3.3.1, 3.3.3.1	Const declaration	1.7.1, 2.4.1, 3.3.3.1, 6.5, 6.6, 8.1, 8.2.6
BeforeUpdate event, form	2.5.2, 2.6.1, 3.3.1, 3.3.2	Constant	F.1.2
Blank record when form opened	2.6.1	CONSTRAINT SQL	App G.3
Boolean datatype	2.6.1, 3.3.1, 3.3.3.2, 5.5.2	Continue code over two or more lines	3.3.3.1, 4.3.9, 5.7, 6.3, 6.6, 7.3.5
BorderStyle property	5.6	Control Tip text	2.4.4, 4.2.6
BoundColumn property	3.4.2, 3.6.1	ControlSource property	3.2.2.1, 3.2.2.3, 3.6
Breakpoint, in Debugger	1.4.3.2	Count of records	3.4.1.2
ByRef, parameter	1.7.1, App F.4	CREATE TABLE SQL	6.5, App G.1
Byte datatype	1.4.1	CREATE VIEW SQL	App G.5
ByVal, parameter	1.7.1, App F.4	Currency datatype	1.4.1
		Current event, form	2.2.3, 2.2.4, 2.3, 2.6.1, 3.2.2.1, 3.2.2.3, 3.2.3.2, 3.2.4, 3.4.2, 3.6.2, 4.3.3, 4.3.8, 7.2, App A

INDEX

<u>Item</u>	<u>Section</u>	<u>Item</u>	<u>Section</u>
D			
Datamode, when open form	4.3.4, 4.3.7	Error message, DoMenuItem cancelled	2.5.2
Datatype, creating a table SQL	6.5	Error message, duplicate key	8.2.8.2
Datatypes	App F.1.1	Error message, expected variable or procedure, not module	1.4
Date & Time on form, dynamically	4.2.1	Error message, expression with no value	5.5
Date function	1.1.2, 3.3.1, 4.2.1, 5.6, App H.1	Error message, field Fn doesn't exist in table	7.3.6
Date/Time datatype	3.3.1, 3.5.5, 6.5, 6.6, App F.2.2	Error message, file locked for editing	7.3.6
Date/Time functions	App H.1	Error message, invalid procedure call or argument	App H.3
DateAdd function	3.3.1, 6.6, App H.1	Error message, invalid use of Null	3.2.3.2, 7.3.6
DateDiff function	3.2.3.1, App H.1	Error message, invalid use of Null	3.2.3.2, 7.3.5
Date literal values, using in the	3.2.3.1	Error message, MS Access can't find field name	8.4.3
Debugger and in code		Error message, MS Jet not recognise field name or expression	8.4.3
Date values, calculated, using in SQL	6.6 (at end), 8.3.3	Error message, not a valid path	7.3.6
Day function	3.2.3.1, 8.3.3, App H.1	Error message, object-defined error	App C
DCount function	3.4.1.2, 3.5.1, 3.5.6, 4.3.9, 4.4.3, 4.4.4, 6.4, 6.5, 6.6, 8.2.8.1, 8.4.5, App H.6	Error message, sub or function not defined	1.4.3.3, App C
Debugging code	1.4.3, 3.2.3.1, 6.3	Error message, subscript out of range	App F.1.5
DELETE SQL	6.5, 6.6, 7.3.4, 8.2.9, 8.3.3, 8.4.5	Error message, when saving	2.5
Delete table	See DROP TABLE SQL	Error wizard code, amending	2.5.2
Deleting Records	2.6.2	Errors, in coding	App C
Detail_Format event, report	5.8	Errors, in embedded SQL	6.6
Detail_Print event, report	5.3, 5.8	Error-trapping	See On Error
Dialog	5.6	Event-based programming	1.1.4, App A
Dim declaration	1.4.1, 6.5, 6.6, 8.4.5	Events, order of	1.1.4
Dirty event, form	2.7.4.	Exit Do/For	3.7.8, App F.3.3
DISTINCT in SQL statement	3.5.2	Exit Sub	1.6, 2.5.2, 6.5
DLookup function	3.2.4, 4.4.4, 4.4.6, 8.4.5, App H.6	Explicit, compiler option	1.2, App C
Do...Loop	6.5, App F.3.3	Export to spreadsheet	7.3.5
Documenting code	1.8, App D	F	
Domain Aggregate Functions	3.2.4, App H.6	Filter on DoCmd.OpenForm	4.3.9
Dot operator, the	1.1.3	Filter property	3.5.1, 3.5.7
DoubleClick event	3.6.3.4, 8.2.7, 8.3.2	FilterOn property	3.5.1, 3.5.6, 3.5.7
DROP TABLE SQL	6.5	Filters, applying	3.5
Dropdown combo box property	3.6.2, 8.3.4	Filters, combining	3.5.7, 3.7.5
DSum function	3.7.2, App H.6	Filters, removing	3.5.6, App J
E			
Editing data	2.3	Financial functions	App H.4
Email, sending	App J	Flag, programming technique	2.6.1, 3.3.3.2, 5.5.2
Enabled property	1.1.3, 2.4.4, 3.3.3.1, 8.2.6, 8.2.8.1	FontBold property	3.2.2.2
Err object	2.5.2, 3.4.1.1	FontBold property	5.3
Error message, ambiguous name	App C	FontItalic property	5.3
Error message, can't assign value	3.2.2.1	For...Next loop	3.7.8, 6.6, App F.3.3
Error message, cancelled prev op	6.6, 7.2, App H.6	ForeColor property	2.4.1, 2.4.3, 2.5, 2.6.1, 5.3
Error message, combo box list	6.3	Form Current event	See Current event, form
Error message, could not find file	7.3.6	Form properties	4.2.2
Error message, could not lock table	6.5	Form, change size at run-time	See InsideWidth property
Error message, data validation	3.3.1	Format function	7.3.5, App H.2
		FormatDateTime function	5.6, 8.4.5, App H.2

INDEX

<u>Item</u>	<u>Section</u>	<u>Item</u>	<u>Section</u>
Forms Collection	3.2.4, 3.5.2, 3.5.3, 3.5.5, 4.3.8, 4.3.9, 5.6, 5.7, 6.4, 8.4.6.2, App C, App I	Load event, form	2.2.1, 2.6.1, 4.2.1, 5.6, 6.4, 6.5
Formula	See ControlSource	Local	1.2, 1.4.1, 1.4.3.3, 1.6, App F.1.3
Formula, in code	3.2.2.3	Locals window, in Debugger	1.4.3.2
Function	See Procedure	Locked property	3.2.2.1, 3.4.2
G		LostFocus event	3.4.2
Global	1.7.1, 2.6.1, 6.4, 6.6, App F.1.3	Lottery Numbers	6.5
Glossary	1.3	M	
GotFocus event	3.4.2, 3.5.1	Maths functions	App H.3
GoToRecord method	2.6.1	MDE files	7.6.2
GRANT SQL	App G.10	Membership Category form	1.6, 1.9.3, 6.4
Greying-out	See Enabled	Membership form	2.1, 2.4.2, 2.4.3, 2.4.4, 2.5.3, 2.6.4, 3.2.3.2, 3.4.2, 3.5.1, 4.3.2, 4.4.3, 7.2
H		Message boxes, separate lines in	See vbCrLf
Heading on form/reports	2.2.1	Method	1.1.3
Help, Access and VBA	1.3	Mid function	3.5.1, 3.7.8, App H.2
Hidden field (textbox) on form	4.3.8, 8.4.6.1, 8.4.6.3	Miscellaneous functions	App H.5
Hidden column, in list box	3.6.1, 8.2.4	Modules: Form, Report and Access	1.1.1, 1.2, 1.4.1, 3.2.3.1
Highlighting item on form	3.2.2.2	Month function	3.2.3.1, 8.3.3, App H.1
Highlighting item on report	5.3	MsgBox function	1.7, App H.5
I		MultiRow property, tab control	7.2
IF statement	1.4.1, 1.7, 3.2.2.2, 3.2.3.1, 5.3, App C, App F.3.2	MultiUser access	7.5.4
IIF function	3.2.3.2, 8.2.5, 8.3.1, App H.5	myCalculateAge function	3.2.3
Immediate window, in Debugger	1.4.3.2, 1.7.1, 3.2.3.1	myCalcYears function	3.7.3, 5.2, 5.3, 5.5
Import from spreadsheet	7.3.2, 7.3.3, 7.3.4	myBooking sub	8.4.6.3
Index, for table	8.2.8.2	myCheckReorder level sub	3.2.2.2, 3.2.2.3
Infinite loop	App F.3.3	myCheckSlot	8.4.6.1
InputBox function	1.7.3, 7.3.2, App H.5, H.7	myCompareValues function	3.3.3.2
INSERT SQL	6.3, 6.5, 6.6, 6.7.1, 8.3.3, 8.3.5, App G.4	myCreateDatetable sub	8.3.3, 8.3.4, 8.4.3
InsideWidth form property	4.3.8	myShowMemberCount sub	3.4.1.2, 3.5
Int function	3.3.3.1, App H.3	myDisplayInfoMessage sub	1.7, 2.5.3, 2.6.2, 7.3.2, 8.2.8.1, 8.2.9
IsDate function	3.3.1, App H.5	myDisplayWarningMessage sub	1.9.1, 2.5, 3.3.1, 3.3.2, 3.3.3.1, 5.5.1, 8.2.8.1, 8.2.8.2, 8.3.5
IsMissing function	1.7.1	myHiddenFields sub	8.4.6.1
IsNull function	3.2.3.2, 3.3.1, 3.3.3.1, 8.4.6, App H.5	myImportData sub	7.3.2
IsNumeric function	3.3.3.1, App H.5	myIsAlphabetic function	3.5.1, 3.7.8
L		myResetButtonsToOff sub	2.4.1, 2.5
Label, in code	1.6	mySetToViewMode sub	2.2.4, 2.5
Len function	3.5.1, 3.7.8, App H.2	myUpdateFee function	1.4.1, 1.5
LimitToList property	3.5.2, 3.5.3, 3.6.1, 5.7, 6.3, 6.4	myUSADate function	8.3.3, 8.3.5
Linking tables	7.5.1, 7.5.3	myYesNoQuestion function	1.7.2, 2.5.2, 2.6.2, 6.3, 7.3.5, 8.2.9, 8.4.5
List box, using	3.6, 6.5, 8.2.4, 8.3.1, 8.3.4	N	
ListWidth property	3.4.2, 3.6.1	Naming conventions	App E
ListRows property	3.6.1	NoData event, on report	5.5
Literal	App F.2.2	NotInList event, combo box	6.3
		Now function	4.2.1, App H.1
		Null	3.2.2.3, 3.2.3.2, 3.2.4, 4.4.4, 5.6, see also IsNull

INDEX

<u>Item</u>	<u>Section</u>	<u>Item</u>	<u>Section</u>
Numeric datatype	3.5.3, 6.4, 6.6, App F2.2	SELECT SQL	3.2.4, 6.5, App G.6,
Nz function	3.4.2, App H.5	SetFocus method	App G.8 1.1.3, 1.9.3, 2.2.3, 2.6.1, 3.3.2, 4.2.3, 6.4, 7.2, 8.2.8.1
O			
On Error, code in event	1.6, 2.5.1, 2.5.2, 7.3.6, 8.2.8.2, 8.4.4	Sorting and Grouping, report	5.7
Open event, report	5.2, 5.4, 5.7	SpecialEffect property	2.4.2, 3.2.2.1
OpenForm method	4.3.4, 4.3.7, 4.3.9, App J	Splitting a database	7.5
OpenQuery method	1.6	SQL	6, 7.3.2, App G
OpenReport method	5.6, App J	Startup options	4.2.4, 7.6.1
Option Compare Database	1.2	Step through code, in Debugger	1.4.3.2
Option Explicit	1.2	stLinkCriteria variable, on	4.3.4, 4.3.9
Optional parameter	1.7.1, App F.4	DoCmd.OpenForm	
OrderBy property	5.7	Stock form	2.7.3, 3.2.1
OrderByOn property	5.7	Str function	3.4.2, App H.5
P			
PageIndex property	7.2	String datatype	1.7.1, 3.5.1, 3.5.2, 6.3, 6.5, 6.6, App F2.2 App H.2
Parameter	1.4.1, 1.7.1, 8.2.6, 8.2.7, 8.4.6.1, App F.4	String functions	
Print event, report header	5.4	Sub	See Procedure
Private	1.1.2, 1.2, 1.4.3.2, App F.1.4	Sub Query	8.4.3
Procedure: Sub and Function	1.1.2, App F.4	T	
Prompts when coding	1.4.1	Tab controls on forms	7.2
Property	1.1.3	Temporary table, using	6.5
Public	1.1.2, 1.7.1, 2.4.1, 2.4.2, 3.2.3.1, 8.1 App F.1.4	Test Plans and Notes	1.4.3.2, 2.1, 3.2.3, 3.3.3.2, 5.3
Q			
Query, action	6.2	Text field	See String datatype
Query, Crosstab	8.4.1, 8.4.3, 8.4.4, 8.4.5	Textbox	3.2.2.3
Query, data definition	6.2	Timer event	4.2.1
Query, Outer Join	8.2.3, 8.3.4	TimerInterval property	4.2.1
Quit method	4.2.5	TransferSpreadsheet method	7.3, App J
Quotation marks in string	3.5.1, 3.5.2, 6.3, 6.4, 6.5, 6.6	Transparent property	2.4.3, 2.4.4
R			
Randomize	See Rnd function	Type conversion functions	App H.7
RecordSource property	3.2.4, 5.6	U	
RemoveItem method	3.6.4	UCase function	3.5.2, 3.7.8, App H.2
Repaint method	1.1.3, 1.9.3	Undo	2.5.2, 3.6.3.5
Requery method	3.6.2, 3.6.3.2, 8.2.6, 8.2.7, 8.3.5, 8.4.5 App G.10	UNION SQL	1.8.2, App G.7
REVOKE SQL	App G.10	Unique property	8.2.8.2
RGB function	2.4.1	UPDATE SQL	6.4, 6.7.2, 7.3.2
Rnd function	6.5, App H.3	V	
Round function	1.4.1, App H.3	Validations	3.3
RowSource property	3.4.2, 3.6.1, 6.5	Value property, calendar control	5.6
RowSourceType property	3.6.1, 6.5	Variable	App F.1.2
RunSQL method	6, 8, App J	Variant datatype	3.2.2.3, App F.1.1, App H.5
S			
Saving automatically	3.2.2.3	VBA language	1.3
Saving records	2.5	vbBlack constant	See Colour Constants
Scope	App F.1.3	vbBlue constant	See Colour Constants
SetWarnings method	6.2, 8	vbCrLf constant	2.6.2, 6.3, 6.6, 8.4.5
		vbInformation constant	1.7.1
		vbLongDate constant	5.6
		vbNo constant	6.3, 6.6
		vbQuestion constant	1.7.2, 6.6
		vbRed constant	See Colour Constants
		vbSunday constant	4.2.1

INDEX

<u>Item</u>	<u>Section</u>
vbWhite constant	See Colour Constants
vbYesNo constant	1.7.2, 6.6
Viewing data via a form	2.2
Visible property	2.4.4, 4.3.8, 5.5.2, 7.2
<u>W</u>	
Watch window, in Debugger	1.4.3.2
Weekday function	4.2.1, App H.1
Weekday, sort in day of week order	8.3.1
WeekdayName function	4.2.1, App H.1
While...Wend loop	App F.3.3
Wildcard	3.5.1
With	7.7, 8.2.6, 8.2.7
Wizard code	1.6, 2.5.2
Wizard code, calling	2.6.1, 3.2.2.3
<u>Y</u>	
Year function	3.2.3.1, 3.3.3.1, 8.3.3, App F.1.5, App H.1
Yes/No datatype	3.5.4, 6.6