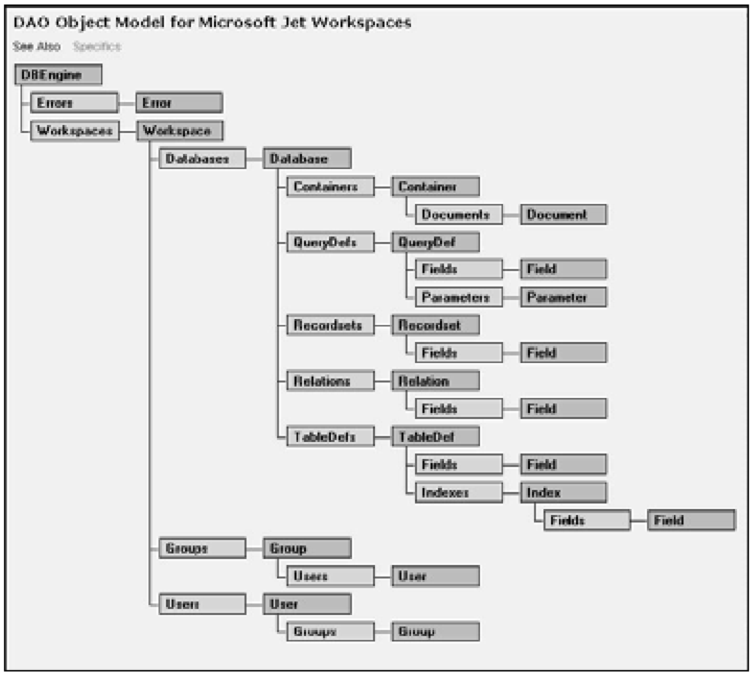
Appendix A

Creating and Manipulating Databases with DAO

Although the default Object Model in Microsoft Office Access 2003 is ADO (ActiveX Data Objects), you will still encounter numerous Microsoft Access applications out there that are programmed using the previous data access object model known as DAO (Data Access Objects). Since these database applications may need to be updated to the newest data access technology, you may want to go over most important features of DAO to better understand programming differences when porting code from DAO to ADO.

Similar to ADO, you can use DAO to manipulate both the structure of your database as well as its data by using Visual Basic code. Most DAO objects represent objects that you work with in your database. For example, a TableDef object represents an Access table, a QueryDef object represents a query, and a Field object corresponds to a field in a table. This appendix introduces you to many DAO objects. You will learn here how to create various objects and set their properties. Next, you will manipulate these objects by applying various methods. Before we start, you may want to take a look at the DAO hierarchy of objects shown in Figure A-1.

Figure A-1 The DAO Object Model



As you can see from Figure A-1, DAO objects are organized in hierarchical relationships. The DBEngine object positioned at the top in the DAO object hierarchy is often referred to as the Jet engine and is used to reference the database engine as a whole. All the other objects and collections in the DAO object hierarchy fall under DBEngine.

The DBEngine contains the following two collections of objects:

- The Errors collection stores a list of errors that have occurred in the DBEngine. These errors are represented by the Error objects and should not be confused with the Err object which stores run-time errors generated in Visual Basic.

- The Workspaces collection (which is the default collection of the DBEngine object) contains the Workspace objects and is used for database security in multi-user applications. The Workspace object is used in conjunction with User and Group objects.

Each open database is represented by the Database object. The Database object is used to reference a Microsoft Access database file (.MDB file) or another external database represented by an ODBC data source. The Databases collection contains all currently open databases (unlike the Microsoft Access user interface, in DAO you can have more than one database open at a time).

The Containers, QueryDefs, Relations, and TableDefs collections contain objects which are used to reference various components of the Database object.
For example, the TableDef object represents a table or a linked table in a Microsoft Jet workspace. The QueryDef object represents a query in DAO. If values are supplied to a query, they are represented in DAO by a Parameter object. The Parameter collection contains all of the Parameter objects defined for a QueryDef object. The Relation object represents a relationship between fields in tables and queries. The Container object is used to access collections of saved objects that represent databases, tables, queries, and relationships.

The Recordsets collection contains all open Recordset objects. Each Recordset object represents a set of records within a database. You will use Recordset objects for retrieving, adding, editing, and deleting records from a database.

The Field object represents a field in a table, query, index, relation, or recordset. The Fields collection is the default collection of a Tabel Def, QueryDef, Index, Relation, and Recordset object.

Some DAO objects have a Properties collection. The Properties collection contains a separate object for each property of the DAO object that is referenced. You can use an object's Properties collection to enumerate its properties or to return their settings. You can also define your own custom properties on DAO objects.

As you work with the following pages of this appendix, refer to the DAO object model to find out how the objects used in example procedures relate to one another.

Before you can begin programming with Data Access Objects (DAO), you must establish a reference to the DAO Object Library. Follow these steps to set up the necessary reference:

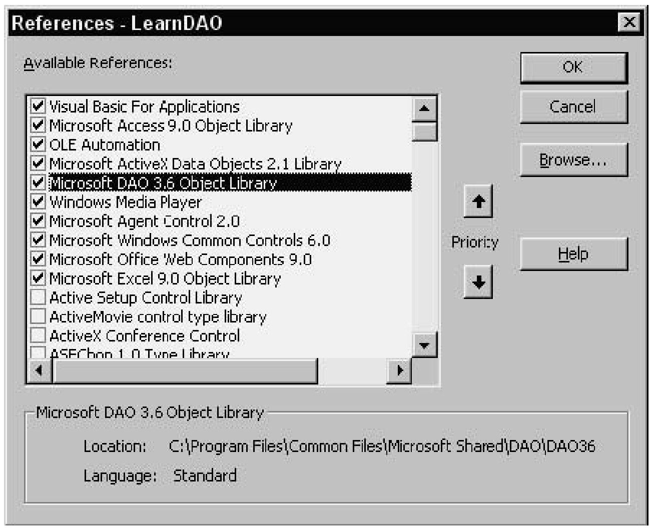
1. In the Visual Basic Editor window, choose Tools | References.

2. In the Available References box (Figure A-2), click Microsoft DAO 3.6 Object Library.

3. Click OK.

   If you cannot find the Microsoft DAO 3.6 Object Library in the Available References box, you must rerun Microsoft Office 2003 Setup to install Data Access Objects for Visual Basic.

Figure A-2 **Referencing DAO Object Library**



Available References:

- ☑ Visual Basic For Applications
- ☑ Microsoft Access 9.0 Object Library
- ☑ OLE Automation
- ☑ Microsoft ActiveX Data Objects 2.1 Library
- ☑ Microsoft DAO 3.6 Object Library
- ☑ Windows Media Player
- ☑ Microsoft Agent Control 2.0
- ☑ Microsoft Windows Common Controls 6.0
- ☑ Microsoft Office Web Components 9.0
- ☑ Microsoft Excel 9.0 Object Library
- ☐ Active Setup Control Library
- ☐ ActiveMovie control type library
- ☐ ActiveX Conference Control
- ☐ ASFChop 1.0 Type Library

Microsoft DAO 3.6 Object Library
Location: C:\Program Files\Common Files\Microsoft Shared\DAO\DAO36
Language: Standard

This appendix explains how to:

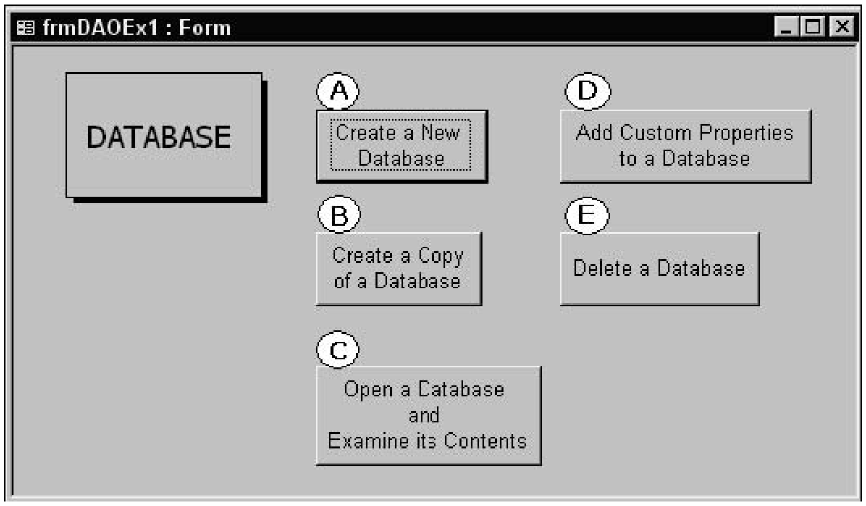
- Work with a database using Data Access Objects (DAO)
- Create tables and define fields
- Add indexes and relations
- Create and execute queries
- Manipulate records and sets of records
- Implement database security and transaction processing

Working with a Database Using DAO

During the course of working with Microsoft Access, you've probably designed a great number of databases using the Access user interface. You already know that creating a new database is as simple as coming up with a database name. However, if you want to perform the same task programmatically, things can get a little more complicated, especially when you're new to database programming.

The easiest way to learn how to work with a Microsoft Access database programmatically is to create a short VBA procedure that performs a specific task. Figure A-3 shows typical things you may want to do with a database object.

Figure A-3 Typical Database Operations


frmDAOEx1 : Form
DATABASE
A
Create a New Database
B
Create a Copy of a Database
C
Open a Database and Examine its Contents
D
Add Custom Properties to a Database
E
Delete a Database

Creating a Database

To get started with a database creation you need to learn about the Workspace object and its CreateDatabase method. Each Access user session has one Workspace object. When you start Microsoft Access, the program automatically creates a default workspace named DBEngine.Workspaces(0). The Workspace object has several useful methods among which the most frequently used ones are CreateDatabase (for creating a new database, see Listing A-3 A) and OpenDatabase (for opening an existing database, see Listing A-3 C).

Creating a new database requires that you specify the name and the path of your database as well as the built-in constant indicating a collating order for creating the database. For English, German, French, Portuguese, Italian, and Modern Spanish use the built-in constant dbLangGeneral. For other languages, and other options that can be used when creating a new database, see the online Help for the CreateDatabase method.

The procedure in Listing A-3 A prompts the user for a database name and then proceeds to create a new database. As soon as the database is created, it is closed with the Close method of a Database object.

Listing A-3 A Creating a new database using DAO

```
Private Sub cmdCreate_Click()
   Dim db As DAO.Database
   Dim dbName As String

      dbName = InputBox("Enter the name of a new database:", _
         "Database Name")
      If dbName = "" Then Exit Sub
      Set db = CreateDatabase(dbName, dbLangGeneral)
      db.Close
      Set db = Nothing
End Sub
```

To create a Database object in code, first declare an object variable of type Database. Because in Access 2003 the default object library is ADO (see Chapter 2 for working with ADO objects), it's a good idea to qualify the object with the name of the DAO object library. By qualifying objects when you use them, you ensure that Visual Basic will always create the correct object. Once the Database object variable is defined set the variable to the object returned by the CreateDatabase method:

```
Set db = CreateDatabase(dbName, dbLangGeneral)
```

The CreateDatabase method creates an empty database. If the database already exists, an error occurs. You can check for the existence of the database by using an If statement in combination with the VBA Dir function and then use the VBA Kill statement to delete the database before calling the CreateDatabase method (see Listing A-3 B).

Copying a Database

At times you may want to duplicate your database programmatically. Creating a copy of your database in code requires that you define two string variables: one for holding the name of the source database and the other one specifying the name for the duplicate version. These variables are then used with the CompactDatabase method of a DBEngine object to create a copy of a database.

Before using the CompactDatabase method, make sure the source database is closed and there is enough disk space to create a duplicate copy. Listing A-3 B shows how to copy a database.

Listing A-3 B Copying a database with DAO

```
Private Sub cmdCopy_Click()
    Dim dbName As String
    Dim dbNewName As String

    dbName = InputBox("Enter the name of the database you want " & _
                "to copy: " & Chr(13) _
                & "(example: C:\TestData.mdb)", "Create a copy of")
        If dbName = "" Then Exit Sub
        If Dir(dbName) = "" Then
            MsgBox dbName & " was not found. " & Chr(13) _
                & "Check the database name or path."
            Exit Sub
        End If
    dbNewName = InputBox("Enter the name of the duplicate " & _
                    "database:" & Chr(13) _
        & "(example: C:\TestData2.mdb)", "Save As")
            If dbNewName = "" Then Exit Sub
            If Dir(dbNewName) <> "" Then
                Kill dbNewName
            End If
            DBEngine.CompactDatabase dbName, dbNewName
End Sub
```

The procedure in Listing A-3 B prompts for the name of the source database and checks whether the user entered a valid database name. If the user clicked Cancel or entered invalid name, the procedure ends. The second InputBox statement prompts the user for the name of the duplicate database. The new database name must be different from the existing one. If the user clicks Cancel, the procedure will end. If the database already exists, it will be automatically deleted. Notice how the VBA Dir function is used to check for the existence of the database with the specified name:

```
If Dir(dbNewName) <> "" Then
       Kill dbNewName
End If
```

Because the database cannot be deleted programatically using DAO, the VBA Kill statement is used to perform the deletion. The last statement in the above procedure uses the CompactDatabase method of the DBEngine object to create a copy of a database using the user-supplied arguments: a source database name (dbName) and a destination database (dbNewName).

Opening and Examining an Existing Database

To open an existing database, use the OpenDatabase method of the Workspace object. This method requires that you provide at least one parameter — the name of an existing database. Before you can list the contents of your database, you should learn few things about containers and documents. Each Database object has a Containers collection that consists of built-in Container objects. The Containers collection is used for storing Microsoft Access's own objects. The Jet engine creates the following container objects: Databases, Tables, and Relations. Other container objects are created by Microsoft Access (Forms, Reports, Macros, and Modules).

The table below shows container objects and the type of information they contain.

Table A.1 Container objects and the type of information they contain

Container Name	Type of information stored
Databases	Saved databases
Tables	Saved tables and queries
Relations	Saved relationships
Forms	Saved forms
Modules	Saved modules
Reports	Saved reports
Scripts	Saved scripts

Each container object contains a Documents collection. Each document in this collection represents an object that can be found in an Access database. For example, the Forms container stores a list of all saved forms in a database and each form is represented by a document object. You cannot create new Container and Document objects. You can only retrieve the information about them.

Listing A-3 C uses For Each…Next loop to retrieve the names of all the container objects in the opened database. If there are any documents in the specified container, the inner For Each… Next loop prints the name of each document object in the Immediate window.

When you open the database with the OpenDatabase method, you should remember to close it. The Close method removes the database from the Database collection.

Listing A-3 C Opening a database with DAO and reading its contents

```
Private Sub cmdOpen_Click()
   Dim db As DAO.Database
   Dim dbName As String
   Dim conType As Container
   Dim doc As Document
```

```
    dbName = InputBox("Enter a name of an existing database:", _
               "Database Name")
      If dbName = "" Then Exit Sub
      If Dir(dbName) = "" Then
         MsgBox dbName & " was not found."
         Exit Sub
      End If
   Set db = OpenDatabase(dbName)
   With db
   ' list the names of the Container objects
         For Each conType In .Containers
               Debug.Print conType.Name & " container: " & _
               conType.Documents.count
            If conType.Documents.count > 0 Then
         ' list the document names in the specified container
               For Each doc In conType.Documents
                  Debug.Print doc.Name
               Next doc
            End If
         Next conType
         .Close
   End With
End Sub
```

The cmdOpen_Click procedure shown above uses the OpenDatabase method of the DBEngine object to open the specified database in the default workspace. The database is opened as shared and read/write. By supplying additional arguments for the OpenDatabase method you could open the database exclusively (a database opened exlusively can be accessed by a single user at a time) or as read-only. See the online help for details.

Adding User-Defined Properties to a Database

The Database object, like other Data Access Objects in Microsoft Access, contains a Properties collection. You can use a For…Each loop in your VBA code to display values of properties associated with the Database object.

To find out which properties apply to the currently open database, open a new module in your Microsoft Access database and type the ListDatabaseProperties procedure, as shown below:

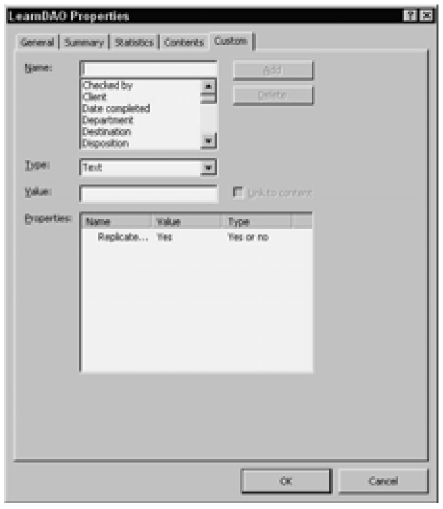
```
Sub ListDatabaseProperties()
   Dim db As DAO.Database
   Dim prp As DAO.Property

   Set db = CurrentDb()

   Debug.Print "Database properties:"
   For Each prp In db.Properties
         Debug.Print prp.Name
   Next prp
End Sub
```

Some of the database properties that will be listed in the Immediate window after running the above procedure are: Name, Connect, Transactions, Updatable, CollatingOrder, QueryTimeout, Version, RecordsAffected, ReplicaID, DesignMasterID, Connection, AccessVersion, Build.

In addition to built-in database properties, you can add custom (user-defined) properties to a database by using the Microsoft Access user interface, or you can write a Visual Basic procedure like the one in Listing A-3 D further in this section.

Figure A-4 shows the Database Properties window where you can add custom properties to a database by using the Microsoft Access user interface. To access this window open the database for which you want to set up custom properties and choose File | Database Properties. Click the Custom tab and type the name of the custom property in the Name box. Click the down arrow next to the Type box and choose the data type for your custom property. Next, type the value of the property in the Value box. Lastly, click the Add button to add the property to the Properties collection.

Figure A-4 **You can use the Database Properties window to add custom properties to your database, or you can write a VBA procedure (see Listing A-3 D further in this section).**



Type: Text

Name	Value	Type
Replicate...	Yes	Yes or no

The cmdDbCustomProp_Click procedure in Listing A-3 D below creates a user-defined property for the current database and appends it to the Properties collection of the database. Next, the procedure enumerates the names and values of all properties in the database.

Notice that instead of using three separate statements to create a user-defined property, you can use the following line of code:

```
Set prp = db.CreateProperty(custPrpName, dbText, “1”)
```

After creating a new property, you must append it to the Properties collection of the Database object:

```
db.Properties.Append prp
```

The ErrorHandler code in this procedure deletes the custom property from the Properties collection if the property with the specified name already exists. This allows you to run the procedure an unlimited number of times. While looping through the Properties collection of your database, other errors may occur.

For example, an attempt to read the value of the Connection property causes Microsoft Access to display error 3251 (“Operation is not supported for this type of object”). If you are working with a Microsoft Access database and not with an ODBC data source, attempting to read the Connection property value will generate an error. You can ignore this error by using the VBA Resume Next statement. This statement will cause Visual Basic to resume the statement immediately following the statement that caused the error, therefore the remaining database property names and values can be read.

After running the procedure in Listing A-3 D the new custom property will be listed as the last property in the Immediate window. This property will not appear in the list of custom properties in the Database Properties window.

Listing A-3 D   Adding a custom property to a Database with DAO

```
Sub cmdDBCustomPrp_Click()
   Dim db As DAO.Database
   Dim prp As DAO.Property
   Dim custPrpName As String

   custPrpName = "dbVersion"
   On Error GoTo ErrorHandler
   Set db = DBEngine.Workspaces(0)(0)
   Set prp = db.CreateProperty()
   prp.Name = custPrpName
   prp.Type = dbText
   prp.Value = "1"
   db.Properties.Append prp
   MsgBox "New property was created."
   For Each prp In db.Properties
      Debug.Print prp.Name & ": " & prp.Value
   Next prp
   Exit Sub
ErrorHandler:
   'property with this name already exists
   If Err.Number = 3367 Then
      db.Properties.Delete custPrpName
      Resume 0
   ' ignore the Connection property
   ElseIf Err.Number = 3251 Then Resume Next
   Else
         MsgBox Err.Number & ": " & Err.Description
   End If
End Sub
```

Although the DBObject has a version property that indicates the version number of the Microsoft Jet database engine (the current version is 4.0), you could use the dbVersion custom property as created in the above procedure to store version number for your database.

To use your custom property (dbVersion), add a text box to any form in a current database. Next, activate the Properties window for the selected text box and type the following in the Default Value property of the text box control:

```
=[CurrentDB].[Properties]("dbVersion").[Value]
```

When you run the form, the text box control will show 1 — the current setting of the dbVersion property.

As mentioned earlier, the DBEngine object represents the Jet database engine. The default collection for this object is Workspaces. When a database is opened it is assigned to DBEngine.Workspaces(0).Databases(0).

In Microsoft Jet you can have multiple databases open at one time. To quickly check the full name of the current database, open the Immediate window and type the bolded statements below:

```
?DBEngine.Workspaces(0)(0).Name
C:\AccessPro\DBSProjectsE\LearnDAO.mdb

?DBEngine(0)(0).Name
C:\AccessPro\DBSProjectsE\LearnDAO.mdb

?CurrentDb.Name
C:\AccessPro\DBSProjectsE\LearnDAO.mdb
```

Deleting a Database

You cannot delete a database programmatically with DAO. To delete a database, use the Visual Basic Kill statement as demonstrated in Listing A-3 E below.

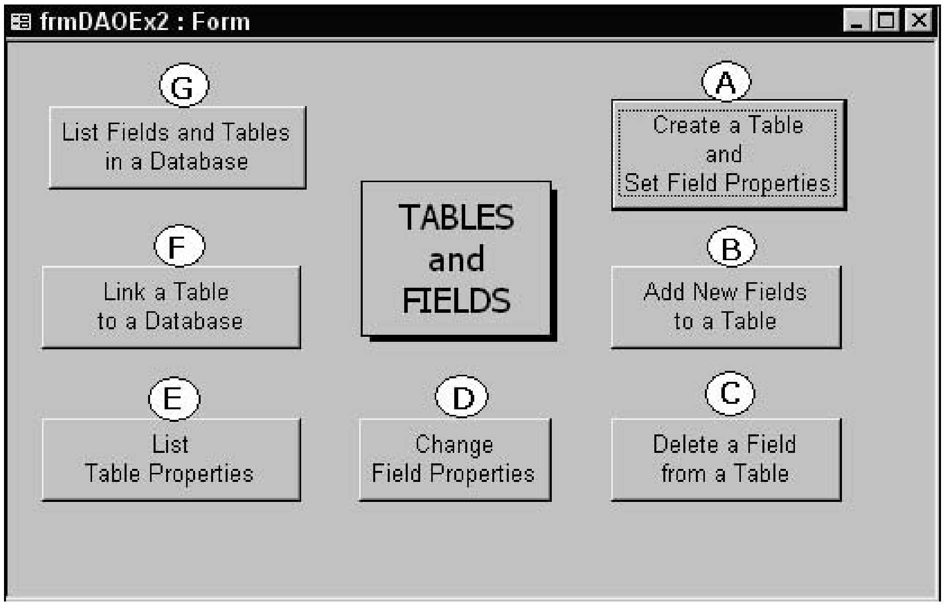
Listing A-3 E   Deleting a database

```
Private Sub cmdDelete_Click()
   Dim dbName As String

   dbName = InputBox("Enter the name of the database " & _
               "you want to delete:" & Chr(13) & _
    "(example: C:\TestData2.mdb)", "Database Name including path")
   If dbName = "" Then
      Exit Sub
   Else
      On Error GoTo ErrorHandler:
      Kill dbName
      MsgBox dbName & " was successfully deleted."
      Exit Sub
   End If
ErrorHandler:
      MsgBox Err.Description
End Sub
```

Creating and Linking Tables with DAO

Now that you know how to create a database programmatically as well as retrieve and create database properties, it's time to fill it in with some useful objects. The first object you will need to create is a table. Figure A-5 shows types of operations you may want to perform on database tables and fields.

Figure A-5 **Frequent operations on database tables and fields**


frmDAOEx2 : Form
G
List Fields and Tables in a Database
A
Create a Table and Set Field Properties
TABLES and FIELDS
F
Link a Table to a Database
B
Add New Fields to a Table
E
List Table Properties
D
Change Field Properties
C
Delete a Field from a Table

Creating a Table and Setting Field Properties

Each saved table in an Access database is an object called a TableDef object.

The TableDef object has a number of properties that characterize it — for example, Name, RecordCount, DateCreated, DateUpdated. The TableDef object also has methods that act on the object. For example, CreateField method creats a new field for the TableDef object and OpenRecordset method creates an object called Recordset that is used to manipulate the data in the table.

The procedure in Listing A-5 A further in this section illustrates how to create a table in a current database. The line,

```
Set db = CurrentDb
```

sets the db variable to point to the currently open database. The instruction CurrentDb is a VBA function. It is not a part of DAO. The CurrentDB function allows you to access the current database from Visual Basic without having to know the database name. The CurrentDB function returns an object variable of type Database that represents the database that is currently open in the Microsoft Access application window. In DAO you refer to the current database as DBEngine.Workspaces(0).Databases(0) or DBEngine(0)(0), or by using the VBA function CurrentDb.

To create a table programmatically, use the CreateTableDef method of a Data-base object. This method requires that you specify a string or string variable to hold the name of the new TableDef object.

For instance, the following line sets the variable tblNew to point to a table named Agents.

```
Set tblNew = db.CreateTableDef(“Agents”)
```

Because a table must have at least one field, the next step in the table creation is using the CreateField method of the TableDef object to create fields. For instance, in the following statement:

```
Set fld = tblNew.CreateField(“AgentId”, dbText, 6)
```

- tblNew is a table definition variable.
- “AgentId” is a string specifying the name for the new field object.
- dbText is an integer constant that determines the data type of the new Field object (See Table A.2 below).
- 6 is an integer indicating the maximum size in bytes for a text field. Text fields can hold from 1 to 255 bytes. This argument is ignored for other types of fields.

Table A.2 Constants for the Type property in VBA

Data Type	**Constant**	**Value**
Boolean	dbBoolean	1
Byte	dbByte	2
Integer	dbInteger	3
Long	dbLong	4
Currency	dbCurrency	5
Single	dbSingle	6
Double	dbDouble	7
Date/Time	dbDate	8
Text	dbText	10
OLE object	dbLongBinary	11
Memo	dbMemo	12
GUID	dbGUID	15

When creating fields for your table you may want to set certain Field properties. Listing A-5 A demonstrates how to set the following built-in properties: Validation Rule, Validation Text, Default Value, and Required.

When you define validation rules for a field you need to set two properties. The Validation Rule property is a text string that describes the rule for validation. In Listing A-5 A we require that each entry in the AgentId field begins with the letter “A”. The validation Text property is a string that is displayed to the user when the validation fails, that is when the user attempts to enter data that does not comply with the validation rule.

The Default value property sets or returns the default value of a Field object. In Listing A-5 A we make the data entry easier for the user by specifying the “USA” as the default value in the Country field. Each new record will automatically have an entry of “USA” in the Country field. Because certain fields should not be left blank, to ensure that the user enters data in a particular field, you should set the Required property of that field to True.

In addition to built-in properties of an object, there are two other types of properties:

- application-defined properties
- user-defined properties

The application-defined property is created only if you assign a value to that property. A classic example of such a property is the Description property of the TableDef object. To set the Description property of a table in the Access user interface, simply right-click on the table name and choose Properties, then type the text you want in the Description field. At this time Access will create a

Description property for the table and will append it automatically to the Properties collection for that TableDef object. If you do not type a description in the Description field, Access will not create a Description property. Therefore, if you use the Description property in your code, Access will display an error. For this reason, it is a good idea to check beforehand whether a referenced property exists.

Users may create their own properties to hold additional information about an object.

Listing A-5 A demonstrates how to use the CreateProperty method of the TableDef object to create application-defined or user-defined properties. To create a property you will need to supply the name for the property, the property type and the property value. For example, in Listing A-5 A the CreateProperty method property is used to create a Caption property for the DateOfBirth field in the newly created table Agents:

```
Set prp = tblNew.Fields("DateOfBirth").CreateProperty("Caption")
```

Next, the data type of the Property object is defined:

```
prp.Type = dbText
```

Table A.2 above lists constants for the Type Property in VBA. Finally, a value is assigned to our new Property:

```
prp.Value = "Date of Birth"
```

Instead of writing three separate lines of code, you can create a new property of an object with the following line:

```
Set prp = tblNew.Fields("DateOfBirth").CreateProperty("Caption", dbText, "Date of Birth")
```

A user-defined property must be appended to Properties collection of the corresponding object.

In Listing A-5 A, the Caption property is appended to the Properties collection of the Field object, and the Description property is appended to the Properties collection of the TableDef object:

```
fld.Properties.Append prp
tblNew.Properites.Append prp
```

After creating a field and setting its built-in, application-defined or user-defined properties, you should use the Append method to add this field to the Fields collection, as in the following example:

```
tblNew.Fields.Append fld
```

Once all the fields have been created and appended to the Fields collection, you must remember to append the new table to the TableDefs collection, as in the following example:

```
dbTableDefs.Append tblNew
```

You can delete user-defined properties from the Properties collection, but you can't delete built-in properties. If you set a property in the user interface, you don't need to create and append the property in code because the property is automatically included in the Properties collection.

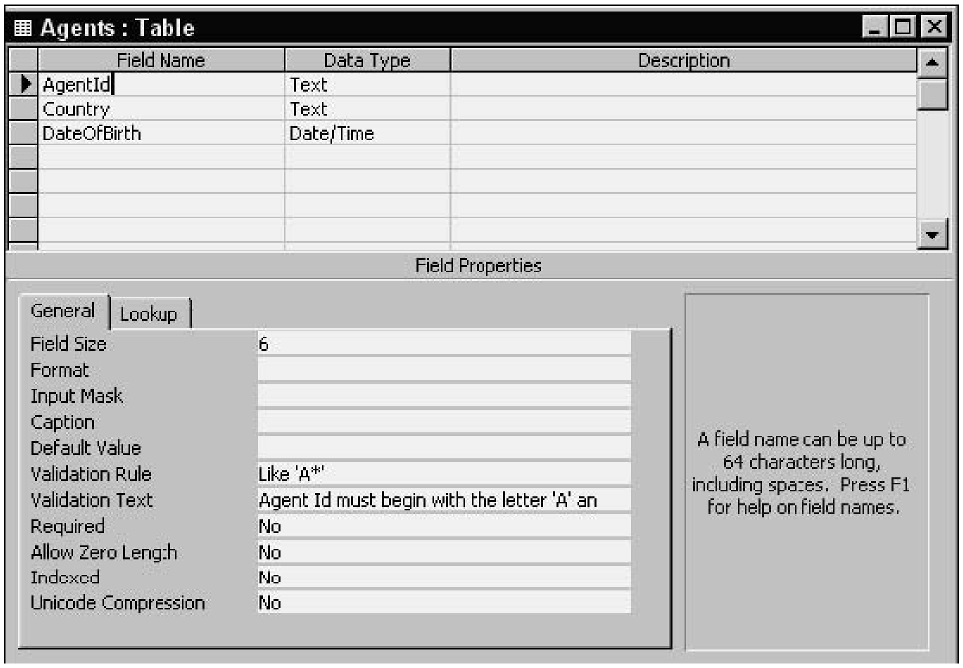
Listing A-5 A Creating a table and setting field properties

```
Private Sub cmdCreateTable_Click()
   Dim db As DAO.Database
   Dim tblNew As DAO.TableDef
   Dim fld As DAO.Field
   Dim prp As DAO.Property

   On Error GoTo ErrorHandler
   Set db = CurrentDb
   Set tblNew = db.CreateTableDef("Agents")

   Set fld = tblNew.CreateField("AgentId", dbText, 6)
   fld.ValidationRule = "Like 'A*'"
   fld.ValidationText = "Agent Id must begin with the letter 'A' " _
      & "and cannot contain more than 6 characters."

   tblNew.Fields.Append fld

   Set fld = tblNew.CreateField("Country", dbText)
   fld.DefaultValue = "USA"
   tblNew.Fields.Append fld

   Set fld = tblNew.CreateField("DateOfBirth", dbDate)
   fld.Required = True
   tblNew.Fields.Append fld

   db.TableDefs.Append tblNew

   'Create Caption property and set its value
   'add it to the collection of field properties
   Set prp = tblNew.Fields("DateOfBirth").CreateProperty("Caption")
   prp.Type = dbText
   prp.Value = "Date of Birth"
   fld.Properties.Append prp
   MsgBox fld.Properties("Caption").Value
   Set prp = tblNew.CreateProperty("Description")
   prp.Type = dbText
   prp.Value = "Sample table created with DAO code"
   tblNew.Properties.Append prp
   Exit Sub
ErrorHandler:
   MsgBox Err.Number & ": " & Err.Description
End Sub
```

After running the code in Listing A-5 A, a new table named "Agents" appears in the Access database window. If you can't see the table, press F5 to refresh the database window.

To check the properties which were set and defined in this procedure, activate the Agents table in the design view, click the Field name for which you set or created a custom property in the code, and examine the corresponding Field Properties. Figure A-6 shows the current settings of the Validation Rule and Validation Text properties for the AgentId field.

Figure A-6 **You can create a database table like this one using VBA code. You can also set appropriate field properties programmatically (see Listing A-5 A).**



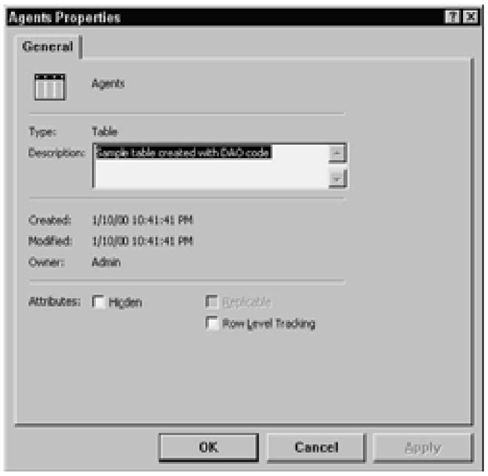
Field Name	Data Type	Description
AgentId	Text	
Country	Text	
DateOfBirth	Date/Time	

Field Size	6
Format	
Input Mask	
Caption	
Default Value	
Validation Rule	Like 'A*'
Validation Text	Agent Id must begin with the letter 'A' an
Required	No
Allow Zero Length	No
Indexed	No
Unicode Compression	No

A field name can be up to 64 characters long, including spaces. Press F1 for help on field names.

To check the value of the Description property for the Agents table that was set as a result of running the procedure in Listing A-5 A, right-click the Agents table in the Database window, and choose Properties from the shortcut menu.

Figure A-7 **A description property for a table object can be set using the Access user interface or via the VBA code (see Listing A-5 A).**



Agents Properties

General

Agents

Type: Table

Description: Sample table created with DAO code

Created: 1/10/00 10:41:41 PM

Modified: 1/10/00 10:41:41 PM

Owner: Admin

Attributes: Hidden Replicable Row Level Tracking

OK Cancel Apply

Adding New Fields to a Table

Using the Access user interface, you can easily add new fields to an existing table by opening a table in Design view and choosing Insert from the menu. After typing the name and data type for the new field in an empty row, the field becomes a part of your table. At times, however, you may need to insert additional fields into an existing table using VBA code.

Listing A-5 B shows how to use the CreateField and Append methods to add two new fields to the tblClientMeetings. See the preceding section on creating tables if you need to review the process of creating a table and defining its fields.

Listing A-5 B Adding new fields to a table

```
Private Sub cmdAddField_Click()
   Dim db As DAO.Database
   Dim tdf As DAO.TableDef

   On Error GoTo ErrorHandler
   Set db = OpenDatabase("C:\DAOEXTE.MDB")
   Set tdf = db.TableDefs("tblClientMeetings")

   MsgBox "Number of fields in the table: " & _
            db.TableDefs("tblClientMeetings").Fields.count
   With tdf
      .Fields.Append .CreateField("NoOfMeetings", dbInteger)
      .Fields.Append .CreateField("Result", dbMemo)
   End With
   MsgBox "Number of fields in the table: " & _
      db.TableDefs("tblClientMeetings").Fields.count
   db.Close
   Exit Sub
ErrorHandler:
   MsgBox Err.Number & ": " & Err.Description
End Sub
```

The cmdAddField_Click procedure shown above uses the following With…End With construct to quickly add two new fields to an existing table:

```
With tdf
   .Fields.Append .CreateField("NoOfMeetings", dbInteger)
   .Fields.Append .CreateField("Result", dbMemo)
End With
```

Each new field is appended to the Fields collection of the specified TableDef object. In the above example, we create a new field on the fly while calling the Append method. Make sure to include a space after the Append method and the dot opearator in front of the CreateField method.

To add two new fields to an existing table without using the With…End With construct, you would use the following two statements:

```
tdf.Fields.Append tdf.CreateField("NoOfMeetings", dbInteger) tdf.Fields.Append
tdf.CreateField("Result", dbMemo)
```

However, using the With…End With construct makes the code both clearer and faster to execute.

Removing a Field from a Table

You may remove any field from an existing table, whether or not this field contains data. However, you can’t delete a Field object from a TableDefs object’s Fields collection after you have created an index that references that field. You must first delete the index.

To remove the field in code, use the Delete method. In Listing A-5 C we remove two fields that were added by the cmdAddField _Click procedure in Listing A-5 B.

Listing A-5 C Removing a field from a table

```
Private Sub cmdDeleteField_Click()
   Dim db As DAO.Database
   Dim tdf As DAO.TableDef

   On Error GoTo ErrorHandler
   Set db = OpenDatabase("C:\DAOEXTE.MDB")
   Set tdf = db.TableDefs("tblClientMeetings")

   MsgBox "Number of fields in the table: " & _
```

```
        db.TableDefs("tblClientMeetings").Fields.count
    With tdf
        .Fields.Delete "NoOfMeetings"
        .Fields.Delete "Result"
    End With

    MsgBox "Number of fields in the table: " & _
        db.TableDefs("tblClientMeetings").Fields.count
    db.Close
    Exit Sub
ErrorHandler:
    MsgBox Err.Number & ": " & Err.Description
End Sub
```

Modifying Table and Field Properties

The cmdChangePrp_Click procedure in Listing A-5 D illustrates how to change the description of a table programmatically:

```
tdf.Properties("Description").Value = "Change properties"
```

This procedure also shows how to set the Format property of a field from a VBA code. Because the Format property of the DateOfBirth field was not yet set via the user interface, Access doesn't know it exists, so the VBA code must use the CreateProperty method to create this property prior to setting its value.

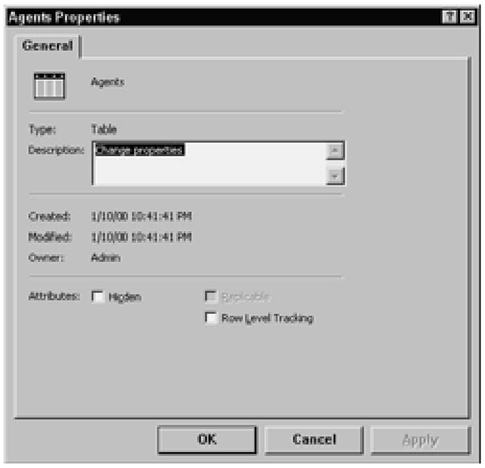
When writing procedures to set properties defined by Microsoft Access, you should write error handling code to verify that the property you are trying to set already exists in the Properties collection.

Listing A-5 D Changing table and field properties

```
Private Sub cmdChangePrp_Click()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim prp As DAO.Property
    Dim fld As DAO.Field

    On Error GoTo ErrorHandler
    Set db = CurrentDb
    Set tdf = db.TableDefs("Agents")
    Set fld = tdf.Fields("DateOfBirth")

    tdf.Properties("Description").Value = "Change properties"
    Set prp = fld.CreateProperty("Format")
    prp.Type = dbText
    prp.Value = "mm/dd/yyyy"
    fld.Properties.Append prp
```

```
   db.Close
   Exit Sub
ErrorHandler:
   MsgBox Err.Number & ": " & Err.Description
End Sub
```

Figure A-8 **The table description can be set manully by right-clicking the table and choosing Properties, or by running a Visual Basic procedure (see Listing A-5 D above).**



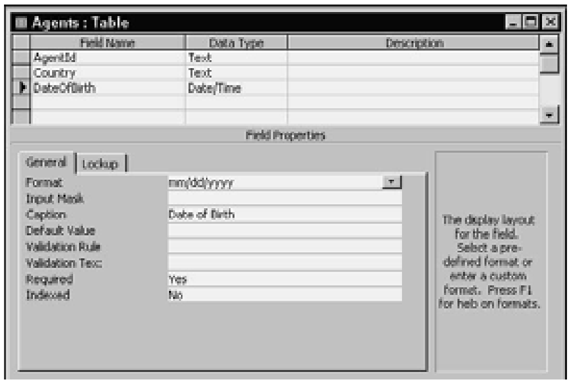
Agents

Type: Table
Description: Change properties

Created: 1/10/00 10:41:41 PM
Modified: 1/10/00 10:41:41 PM
Owner: Admin

Attributes: ☐ Hidden ☐ Replicable ☐ Row Level Tracking

Figure A-9 **The Format property of a field in a table can be set manually in the Design view of a table or programmatically from VBA code (see Listing A-5 D above).**



Field Name	Data Type	Description
AgentId	Text	
Country	Text	
DateOfBirth	Date/Time	

Format	mm/dd/yyyy
Input Mask	
Caption	Date of Birth
Default Value	
Validation Rule	
Validation Text	
Required	Yes
Indexed	No

The display layout for the field. Select a pre-defined format or enter a custom format. Press F1 for help on formats.

Retrieving Table Properties

As mentioned earlier, some DAO objects have a Properties collection. Properties are objects, and just like other objects, they are contained in collections. You can iterate through all of the properties of an object using the For Each…Next programming structure.

Listing A-5 E demonstrates how to produce a list of all properties of a table. When you run the cmdTablePrp_Click procedure, the property names and their current values are written to the Immediate window. The Properties collection allows you to create and store new properties. Refer to the Listing A-5 A to find out how to create new field and table properties.

Listing A-5 E   Retrieving table properties

```
Private Sub cmdTablePrp_Click()
   Dim prp As DAO.Property
   Dim tdf As DAO.TableDef
   Dim db As DAO.Database

   Set db = CurrentDb
   Set tdf = db.TableDefs("Agents")

   For Each prp In tdf.Properties
      On Error Resume Next
         Debug.Print prp.Name & ": " & prp.Value
   Next
   MsgBox "Finished processing."
End Sub
```

Linking a Table to a Database

Linking allows you to use data in another Access database without actually copying the data from the other database. You can also link tables that reside in other programs or file formats such as Microsoft Excel, Microsoft FoxPro, dBase, or Paradox.

To link a table to a database, use the CreateTableDef method to create a new table:

```
Set myTable = db.CreateTableDef("TableDBASE")
```

Next, specify the Connect property of the TableDef object. For example, the following statement specifies the connect string:

```
myTable.Connect = "dBase 5.0;Database=C:\Program Files\Microsoft Office\Office11\1033"
```

Next, specify the SourceTableName property of the TableDef object to indicate the actual name of the table in the Source database:

```
myTable.SourceTableName = "Customer.dbf"
```

Finally, use the Append method to append the TableDef object to the TableDefs collection:

```
db.TableDefs.Append myTable
```

The procedure in Listing A-5 F demonstrates how to link a dBase table to a current database. Prior to running this procedure, locate the Customer.dbf file on your computer and modify the Database path used in the procedure's Connect string.

Listing A-5 F Linking a table to a database

```
Private Sub cmdLinkTblDbase_Click()
   Dim db As DAO.Database
   Dim myTable As DAO.TableDef

   On Error GoTo err_LinkDbaseTable

      Set db = CurrentDb
      Set myTable = db.CreateTableDef("TableDBASE")
      myTable.Connect = "dBase 5.0;Database=C:\Program Files\" & _
            "Microsoft Office\Office11\1033"
      myTable.SourceTableName = "Customer.dbf"
      db.TableDefs.Append myTable

      db.TableDefs.Refresh
      MsgBox "dBase table has been successfully linked."
      Exit Sub
err_LinkDbaseTable:
      MsgBox Err.Number & ": " & Err.Description
End Sub
```

Retrieving the Names of All the Fields in all the Tables

The following procedure prints the name of every table in the current database and lists the names of all the fields in those tables.

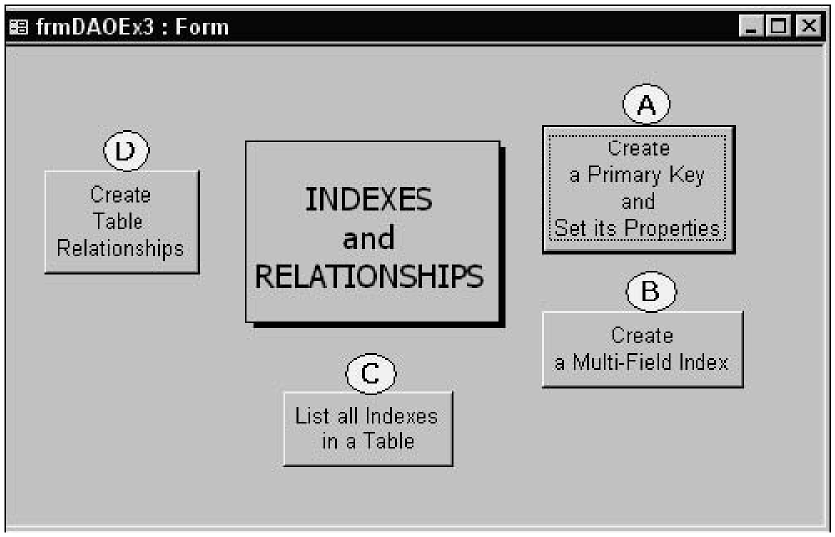
Listing A-5 G Retrieving table and field names

```
Private Sub cmdListFields_Click()
   Dim db As DAO.Database
   Dim x As Integer
   Dim y As Integer

   Set db = CurrentDb
   For x = 0 To db.TableDefs.count - 1
         Debug.Print "Table: " & db.TableDefs(x).Name
      For y = 0 To db.TableDefs(x).Fields.count - 1
         Debug.Print Chr(9) & db.TableDefs(x).Fields(y).Name
      Next y
   Next x
End Sub
```

Creating Indexes and Establishing Table Relationships with DAO

After defining the fields for your tables, take time to identify fields with unique values as well as determine the relationships between tables (Figure A-10).

Figure A-10 Learning about table relationships and indexes.


frmDAOEx3 : Form
D
Create Table Relationships
INDEXES and RELATIONSHIPS
A
Create a Primary Key and Set its Properties
B
Create a Multi-Field Index
C
List all Indexes in a Table

Creating a Primary Key and Setting Its Properties

As you know, each table in your database should include a field (or set of fields) that uniquely identifies each individual record in a table. Such a field or set of fields is called a Primary Key. Listing A-10 A demonstrates how to use DAO programming to create a Primary Key in a table as well as how to set some of the index properties.

Indexes determine the order of records accessed from database tables and whether or not duplicate records are accepted. While indexes can speed up access to specific records in large tables, too many indexes can also slow down updates to the database.
In DAO indexes are created using the CreateIndex method for a TableDef object. The following statement creates an index named PrimaryKey:

```
Set idx = tdf.CreateIndex("PrimaryKey")
```

To ensure that the correct type of index is created, you need to set index properties as needed. For example, the Primary property of an index indicates that the index fields constitute the primary key for the table:

```
idx.Primary = True
```

To specify that the fields in a multi-field index must be filled in, use the Required property as shown below:

```
idx.Required = True
```

In other words, the Required property indicates whether the index can accept null values. When you set this property to True, nulls will not be accepted.

Use the IgnoreNulls property to determine whether a record with a null value in the index fields should be included in the index:

```
idx.IgnoreNulls = False
```

Use the Unique property to specify whether or not the values in an index must be unique:

```
idx.Unique = True
```

To actually index a table, you must use the CreateField method on the Index object to create a Field object for each field to be included in the index:

```
Set fld = idx.CreateField("AgentId", dbText)
```

Once the Field object is created, you need to append it to the Fields collection:

```
idx.Fields.Append fld
```

The last step in index creation is appending the Index object to the Indexes collection:

```
tdf.Indexes.Append idx
```

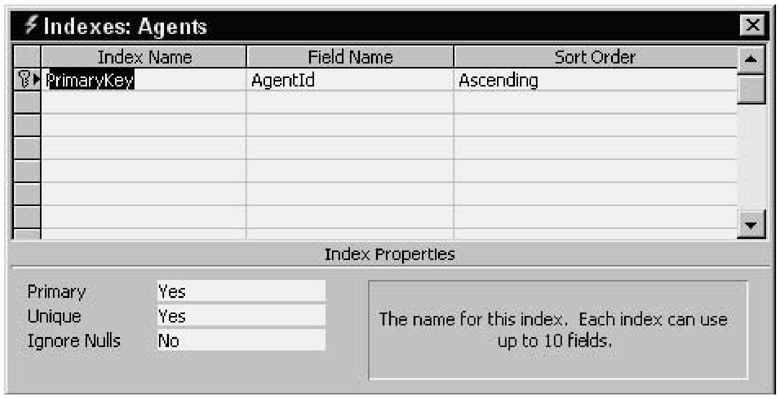
The result of running the cmdCreateIndex_Click procedure is shown in Figure A-11 following Listing A-10 A.

Listing A-10 A Creating a Primary Key and setting its properties

```
Private Sub cmdCreateIndex_Click()
```

```
   Dim db As DAO.Database
   Dim tdf As DAO.TableDef
   Dim fld As DAO.Field
   Dim idx As DAO.Index

   Set db = CurrentDb
   Set tdf = db.TableDefs("Agents")

   'create a Primary Key
   Set idx = tdf.CreateIndex("PrimaryKey")
   idx.Primary = True
   idx.Required = True
   idx.IgnoreNulls = False
   Set fld = idx.CreateField("AgentId", dbText)
   idx.Fields.Append fld

   'add the index to the Indexes collection in the Agents table
   tdf.Indexes.Append idx
End Sub
```

Figure A-11 **Indexes window after running the procedure in Listing A-10 A.**



Index Name	Field Name	Sort Order
PrimaryKey	AgentId	Ascending

Index Properties

Primary	Yes
Unique	Yes
Ignore Nulls	No

The name for this index. Each index can use up to 10 fields.

Creating a Multi-Field Index

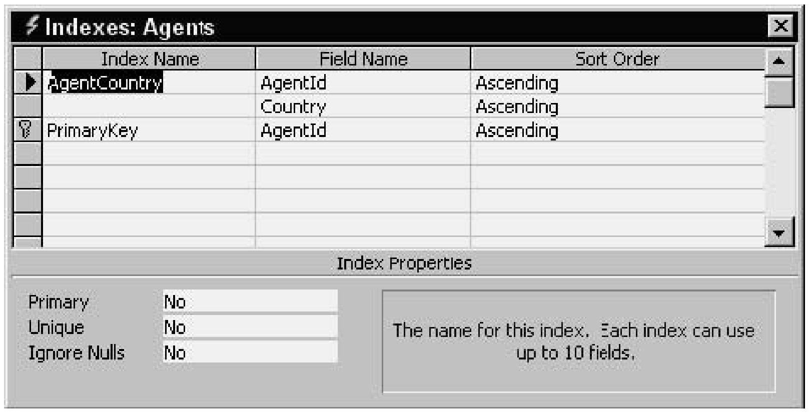
The cmdMultiIndex_Click procedure in Listing A-10 B (shown below) creates a two-field index in the Agents table. Start by using the CreateIndex method on a TableDef object. Next, use the CreateField method on the Index object to create the first field to be included in the index, and then append this field to the Fields collection. Repeat the same two steps for the second field you want to include in the index.

It is important to remember that the order in which the fields are appended has an effect on the index order. If you open the Indexes window after running the procedures in Listing A-10 A and A-10 B, you should see two indexes named AgentCountry and PrimaryKey (see Figure A-12 below Listing A-10 B).

Listing A-10 B Creating a multi-field index

```
Private Sub cmdMultiIndex_Click()
    Dim db As DAO.Database
    Dim tdf As DAO.TableDef
    Dim fld As DAO.Field
    Dim idx As DAO.Index

    Set db = CurrentDb
    Set tdf = db.TableDefs("Agents")

    Set idx = tdf.CreateIndex("AgentCountry")

    Set fld = idx.CreateField("AgentId", dbText)
    idx.Fields.Append fld

    Set fld = idx.CreateField("Country", dbText)
    idx.Fields.Append fld

    tdf.Indexes.Append idx
End Sub
```

Figure 2-12 Indexes displayed in this window were created by running VBA procedures in Listing A-10 A and A-10 B.



Index Name	Field Name	Sort Order
AgentCountry	AgentId	Ascending
	Country	Ascending
PrimaryKey	AgentId	Ascending

Primary	No
Unique	No
Ignore Nulls	No

The name for this index. Each index can use up to 10 fields.

Listing All Indexes in a Table

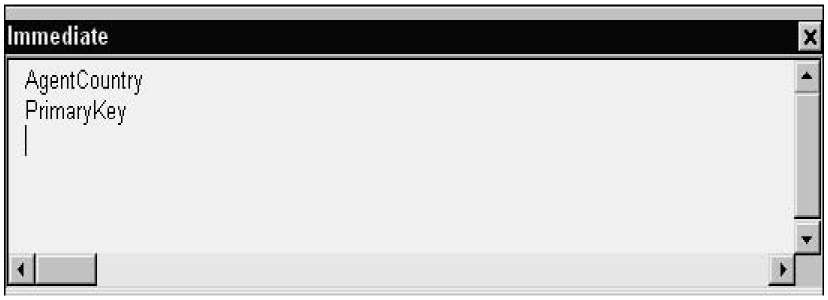
Instead of opening the Indexes window in the Microsoft Access user interface to check the names of available indexes, you may use the For Each…Next VBA looping structure to retrieve the names of indexes from the Indexes collection of the TableDef object. To get the names of indexes in the Agents table, run the cmdIndexNames_Click procedure in Listing A-10 C below. The last statement in this procedure activates the Immediate window, so that you can quickly see the results of running this procedure. The statement,

```
SendKeys "^g"
```

is the same as pressing Ctrl+g, which opens the Immediate window.

Listing A-10 C Retrieving the names of indexes defined in a specific table

```
Private Sub cmdIdxNames_Click()
   Dim db As DAO.Database
   Dim tdf As DAO.TableDef
   Dim idx As DAO.Index

   Set db = CurrentDb
   Set tdf = db.TableDefs("Agents")

   For Each idx In tdf.Indexes
      Debug.Print idx.Name
   Next
   'show immediate window
   SendKeys "^g"
End Sub
```



```
AgentCountry
PrimaryKey
```

Establishing Relationships Between Tables

The right relationship is everything. In order to combine data from two or more tables you need to link the tables through established related fields. A relationship simply matches data in key fields — usually these are fields with the same name in both tables. To create a relationship manually you open the Relationship window, add the tables you want to relate, drag the key field from one table, and drop it on the key field in another table.

A Relation object represents a relationship between certain fields in tables and queries. The Relation object can be used to view or create relationships.

Listing A-10 D demonstrates how to relate three tables to one another using

DAO. The first step in establishing a relationship between two tables is to use the CreateRelation method, as shown below:

```
Set myRelation = db.CreateRelation("OneToMany1")
```

Notice that this method requires a string or a string variable denoting the name of the new relation. In this example, we named the first relationship “OneToMany1.”

The statement,

```
myRelation.Table = "tblCountries"
```

specifies the name of the referenced table (this table contains the primary key).

The statement,

```
myRelation.ForeignTable = "tblCountriesProvinces"
```

specifies the name of the referencing table in this relationship (this table contains the foreign key).

You may specify the Key Table and Foreign Table using one statement as follows:

```
Set myRelation = db.CreateRelation("OneToMany1", "tblCountries", tblCountriesProvinces")
```

Next, you may want to set the Attributes property of the Relation object to allow cascading updates and cascading deletes, so that Microsoft Jet database engine updates or deletes records in related tables automatically:

```
myRelation.Attributes = dbRelationUpdateCascade + dbRelationDeleteCascade
```

Table A-3 shows the attributes for the Relation object.

Table A-3 Attributes for a Relation object

Description	**Constant**
Set one-to-one relationship	dbRelationUnique
No referential integrity	dbRelationDontEnforce
Relationship exists in non-current database that contains the two linked tables	dbRelationInherited
Cascading updates enabled	dbRelationUpdateCascade
Cascading deletions enabled	dbRelationDeleteCascade

After setting the relation attributes, specify the key field in referenced table and foreign key in referencing table:

```
Set myField = myRelation.CreateField("CountryId")
myField.ForeignName = "CountryId"
```

Next, Append the Field object to the Fields collection of the Relation object:

```
myRelation.Fields.Append myField
```

The last step in establishing the relationship between two tables requires that you append Relation object to the Relations collection:

```
db.Relations.Append myRelation
```

The purpose of the cmdRelations_Click procedure is to establish one-to-many relationship between three tables, therefore the procedure goes on to create “OneToMany2” relationship. Because duplicate or invalid Relation names will cause an error when the Append method is invoked, an error handler is included.

The error handler will delete the Relation objects from the collection, if the relations with the specified names already exist. This ensures that you can run the procedure in Listing A-10 D an unlimited number of times. The Resume 0 statement sends the Visual Basic back to the line that caused the error, and the procedure can continue. If any other error is encountered, the procedure will display the error number and its description.

Listing A-10 D Establishing relationships between tables

```
Private Sub cmdRelations_Click()
   Dim db As DAO.Database
   Dim myRelation As DAO.Relation
   Dim myField As DAO.Field

   On Error GoTo Err_Relations
   Set db = CurrentDb
   Set myRelation = db.CreateRelation("OneToMany1")
   myRelation.Table = "tblCountries"
   myRelation.ForeignTable = "tblCountriesProvinces"

   'create one-to-many relationship with Cascading Updates and Deletes
   myRelation.Attributes = _
      dbRelationUpdateCascade + dbRelationDeleteCascade

   Set myField = myRelation.CreateField("CountryId")
   myField.ForeignName = "CountryId"
   myRelation.Fields.Append myField
   'save the relationship
   db.Relations.Append myRelation

   'create a new relationship
   Set myRelation = db.CreateRelation("OneToMany2")

   myRelation.Table = "tblStateProvince"
   myRelation.ForeignTable = "tblCountriesProvinces"

   'create one-to-many relationship with Cascading Updates and Deletes
   myRelation.Attributes = _
      dbRelationUpdateCascade + dbRelationDeleteCascade

   Set myField = myRelation.CreateField("StateProvinceId")
   myField.ForeignName = "StateProvinceId"
   myRelation.Fields.Append myField

   'save the relationship
   db.Relations.Append myRelation
   Exit Sub
Err_Relations:
      If Err.Number = 3012 Then
         db.Relations.Delete "OneToMany1"
         db.Relations.Delete "OneToMany2"
         Resume 0
      Else
         MsgBox Err.Number & ": " & Err.Description
      End If
End Sub
```

Figure A-14 below shows the Relationship window after running the cmdRelations_Click procedure in Listing A-10 D. To examine this window, make sure the Database window is active and choose Tools | Relationships from the Access menu bar. Select Relationships | Show All.

Figure A-14 **You can relate two tables by dragging the primary key from one table and dropping in on the corresponding foreign key in the other table, or running a VBA procedure in Listing A-10 D.**

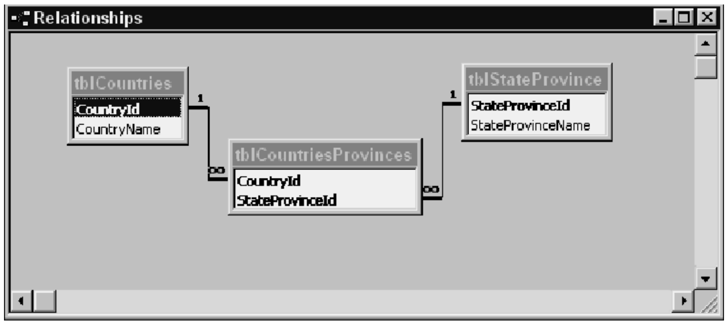
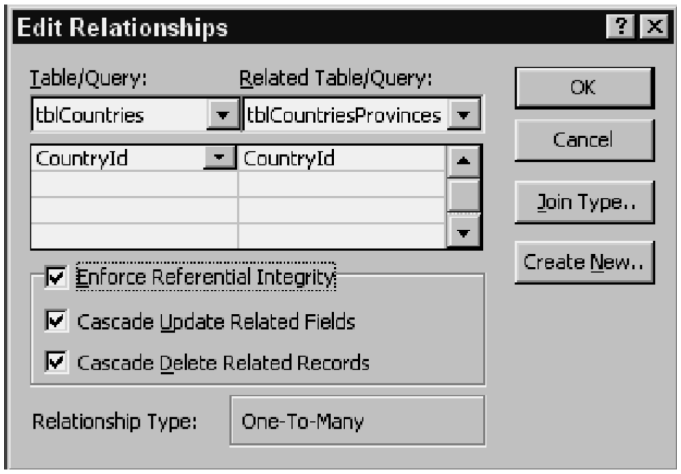


Figure A-15 **The Edit Relationships dialog box displays the Enforced Referential Integrity rules as set by the VBA code in Listing A-10 D.**


Edit Relationships

Table/Query: tblCountries
Related Table/Query: tblCountriesProvinces

Table/Query	Related Table/Query
CountryId	CountryId

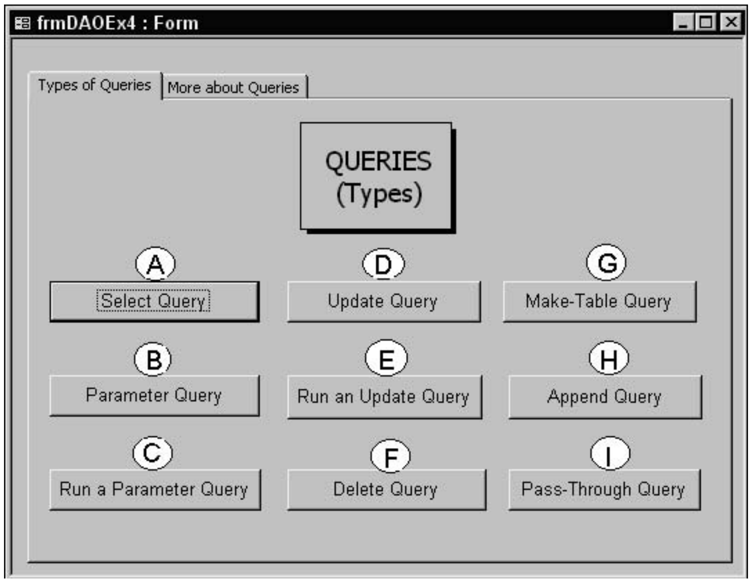
☑ Enforce Referential Integrity
☑ Cascade Update Related Fields
☑ Cascade Delete Related Records

Relationship Type: One-To-Many

Creating and Running Queries with DAO

Microsoft Access supports several types of queries. The simplest queries allow you to select a set of records from a table or a number of tables. Other queries perform specific actions on existing data, such as making a new table, appending rows to a table, updating the values in a table, or deleting rows from a table. Access provides a friendly interface for creating queries manually. This section teaches you how to create the same queries using DAO code.

Figure A-16 **Creating and executing queries programmatically.**



Creating a Select Query with DAO

The cmdSelectQuery_Click procedure in Listing A-16 A illustrates how to create a simple select query that lists all the employees with the entry of 'Ms.' in the Employees table. Select queries retrieve a set of records from a database table. These queries are easily recognized by the SELECT and FROM keywords in their syntax:

SELECT LastName FROM Employees	Selects the LastName field from the Employees table. If there is a space in the field name enclose the field name in the square brackets: [Last Name].
SELECT FirstName, LastName, PhoneNo FROM Employees	Selects the FirstName, LastName, and PhoneNo fields from the Employees table.
SELECT * FROM Employees	Selects all fields for all records from the Employees table. The asterisk (*) is used to represent all fields.

The QueryDef object represents a saved query in a database. All QueryDefs objects are contained in the QueryDefs collection. You can read and set the SQL definition of a Query object using the SQL property.

To create a query in code, use the CreateQueryDef method. For example, to create a Select query named myQuery, the following statement is used:

```
Set qdf = db.CreateQueryDef("myQuery", mySQL)
```

When you specify the name for your query, the new QueryDef object is automatically appended to the QueryDefs collection when it is created. The second argument of the CreateQueryDef method is a string variable that holds a valid Access SQL statement. Prior to using this variable, you must assign to it a string expression:

```
mySQL = "SELECT * FROM Employees WHERE TitleOfCourtesy = 'Ms.'"
```

The WHERE clause is used with the Select queries to specify criteria that determine which records the query will affect. The procedure in Listing A-16 A selects from the Employees table all records that have a value of 'Ms.' in the TitleOfCourtesy field. The equals (=) operator can be substituted by the keyword LIKE, as in the following:

```
mySQL = "SELECT * FROM Employees WHERE TitleOfCourtesy LIKE 'Ms.'"
```

Other examples of using the WHERE clause to restrict records are shown below:

SELECT * FROM Employees
WHERE City IN (‘Redmond’, ‘London’)
	Selects from the Employees table all fields for all the records that have the value Redmond or London in the City field.

SELECT * FROM Employees
WHERE City IN (‘Redmond’, ‘London’)
AND
[ReportsTo] LIKE ‘Buchanan, Steven’
	Selects from the Employees table all fields for all records that have the value Redmond or London in the City field and have a value ‘Buchanan, Steven’ in the [ReportsTo] field.

SELECT*FROM Employees
WHERE ((Year([HireDate])<1993)
OR
(City=’Redmond’))
	Selects from the Employees table all fields for all records that have a value less than 1993 in the HireDate field or have the value ‘Redmond’ in the City field.

SELECT DISTINCT City
FROM Employees
	Selects from the Employees table all the distinct values in the City field. The DISTINCT keyword eliminates duplicate values from the returned set of records.

When creating queries in code, be sure to include an error handler. After all, the query you are trying to create may already exist, or an unexpected error could occur.

Listing A-16 A Creating a Select query

```
Private Sub cmdSelectQuery_Click()
   Dim db As DAO.Database
   Dim qdf As DAO.QueryDef
   Dim mySQL As String
   On Error GoTo Err_SelectQuery

   mySQL = "SELECT * FROM Employees WHERE TitleOfCourtesy = 'Ms.'"
   Set db = OpenDatabase("C:\Program Files\Microsoft Office\" & _
             "Office11\Samples\Northwind.mdb")
   Set qdf = db.CreateQueryDef("myQuery", mySQL)
   Exit Sub
Err_SelectQuery:
   If Err.Number = 3012 Then
      MsgBox "Query with this name already exists."
   Else
     MsgBox Err.Description
   End If
End Sub
```

If you run the cmdSelectQuery_Click procedure in Listing A-16 A, next time you open the Northwind database you should see the query named myQuery in the list of stored queries in the database window.

Instead of queries that have been saved in the database for future use, it is possible to create a temporary query by setting the QueryDefName property to a zero-length string (""), as in the following example:

```
Set qdf = db.CreateQueryDef("", mySQL)
```

The advantage of temporary queries is that they don't clutter the database window.

If you'd like to sort records returned by the Select query, use the ORDER BY clause with the ASC (ascending sort) and DESC (descending sort) keywords, as shown in the following example:

SELECT *
FROM Employees
ORDER BY Country DESC
	Select all records from the Employees table and arrange them in descending order based on the Country field. If no order is specified, the order is ascending (ASC) by default.

Creating and Running Parameter Queries with DAO

A special type of a Select query is known as a Parameter query. Instead of retrieving the same records each time a query is run, a user can enter the search criteria in a special dialog box at run time. In DAO, the parameters of a Parameter query are represented by Parameter objects. The QueryDef object contains a Parameters collection. Parameter objects represent existing parameters.

To create a Parameter query, create a query string that includes the PARAMETERS keyword:

```
mySQL = "PARAMETERS [Enter Country] Text;"&_
"SELECT * FROM CUSTOMERS WHERE Country = [Enter Country];"
```

Listing A-16 B demonstrates how to create a Parameter query that prompts the user for the name of the country to retrieve appropriate records from the Customers table in the Northwind sample database. Before you run this procedure, change the path of the Northwind.mdb to point to the correct folder on your computer.

Listing A-16 B Creating a Parameter query

```
Private Sub cmdParametrQuery_Click()
   Dim db As DAO.Database
   Dim qdf As DAO.QueryDef
   Dim mySQL As String

   On Error GoTo Err_Handler

   mySQL = "PARAMETERS [Enter Country] Text; " & _
```

```
    "SELECT * FROM Customers WHERE Country = [Enter Country];"

   Set db = OpenDatabase("C:\Program Files\Microsoft Office\Office11\" & _
        "Samples\Northwind.mdb")
   Set qdf = db.CreateQueryDef("myParamQuery", mySQL)
ExitHere:
   Set db = Nothing
   Exit Sub
Err_Handler:
   If Err.Number = 3012 Then
      MsgBox "Query with this name already exists."
   Else
      MsgBox Err.Description
   End If
   Resume ExitHere
End Sub
```

Before executing an existing Parameter query, assign a value to the parameter, as shown in Listing A-16 C below. Once the parameter value is specified, you need to open a recordset based on the query. Read the section “Finding and Reading Records” later in this appendix for introduction to recordsets.

Listing A-16 C Running a Parameter query

```
Private Sub cmdRunParamQry_Click()
   Dim db As DAO.Database
   Dim qdf As DAO.QueryDef
   Dim rst As DAO.Recordset
   Dim fld As DAO.Field

   Set db = CurrentDb
   Set qdf = db.QueryDefs("qryPrmClientsByLastName")

  'specify the parameter
  qdf.Parameters("Enter Client's Name") = "K"

   'open a recordset based on the specified query
   Set rst = qdf.OpenRecordset(dbOpenDynaset)
   MsgBox "Number of records: " & rst.RecordCount

   'write the contents of the second field to the Immediate window
   Do Until rst.EOF
      Debug.Print rst(1)
      rst.MoveNext
   Loop

   'close the recordset
   rst.Close
   Set db = Nothing
```

```
End Sub
```

Creating and Running an Update Query with DAO

An Update query is a type of action query. Update queries are very convenient to use when you want to change fields for a single record or for multiple records in a table. The procedure in Listing A-16 D creates and runs an Update query. As a result, the prices for all products supplied by Tokyo Traders are increased by two dollars. The Execute method of a QueryDef object is used to run any type of action query.

The UPDATE statement consists of the following three parts:

UPDATE	tableName or QueryName
SET	expression / operation to perform
WHERE	criteria / limit operation to desired rows

The criteria in the WHERE clause is used to determine which rows will be updated. The update query does not produce a result table. To avoid updating wrong records, always determine which rows you want to be updated by creating and running a Select query first.

Listing A-16 D   Creating an Update query

```
Private Sub cmdUpdateQuery_Click()
   Dim db As DAO.Database
   Dim qdf As DAO.QueryDef
   Dim mySQL As String

   On Error GoTo Err_Handler

   mySQL = "UPDATE Suppliers INNER JOIN Products ON " & _
               "Suppliers.SupplierID = Products.SupplierID " & _
               "SET Products.UnitPrice = [UnitPrice]+2 " & _
               "WHERE (((Suppliers.CompanyName)='Tokyo Traders'));"

   Set db = OpenDatabase("C:\Program Files\" & _
      "Microsoft Office\Office11\Samples\Northwind.mdb")

   Set qdf = db.CreateQueryDef("PriceIncrease", mySQL)
   qdf.Execute

ExitHere:
   Set db = Nothing
   Exit Sub
Err_Handler:
      If Err.Number = 3012 Then
         MsgBox "Query with this name already exists."
      Else
         MsgBox Err.Description
```

```
        End If
        Resume ExitHere
End Sub
```

The cmdUpdateRun_Click procedure in Listing A-16 E demonstrates how to use the Execute method to run an existing (previously saved) Update query.

Listing A-16 E   Running an Update query

```
Private Sub cmdUpdateRun_Click()
  Dim db As DAO.Database


  Set db = OpenDatabase("C:\Program Files\Microsoft Office\" & _
       "Office11\Samples\Northwind.mdb")
  db.Execute "PriceIncrease"
  Set db = Nothing
End Sub
```

Running a Delete Query with DAO

With a Delete query you can delete a single record or multiple records from a database. The Delete statement used to delete rows from a table consists of the following three parts:

DELETE	
FROM	Table Name
WHERE	criteria/limit operation to desired rows

You cannot reverse the operation performed by the Delete statement. Always make a backup copy of your table prior to running a Delete query. It is a good idea to create and run a Select query before Delete to see which rows will be affected by the Delete operation.

The Execute method is used to run action queries or execute an SQL state-ment. This method can take optional arguments. For example, in the statement:

```
qdf.Execute dbFailOnError
```

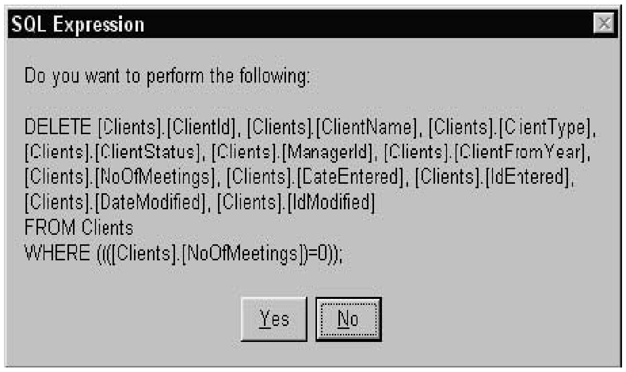
the constant dbFailOnError will generate a run-time error and will roll back updates or deletes if an error occurs.

Use the RecordsAffected property of the QueryDef object to determine the number of records affected by the most recent Execute method. For example, the following statement displays the number of records that were deleted:

```
MsgBox qdf.RecordsAffected & " records were deleted."
```

The procedure in Listing A-16 F runs an existing Delete query if the user responds positively to the message shown in Figure A-17.

Figure A-17 **You can display a SQL statement underlying a query in a message box.**



SQL Expression

Do you want to perform the following:

```
DELETE [Clients].[ClientId], [Clients].[ClientName], [Clients].[CientType],
[Clients].[ClientStatus], [Clients].[ManagerId], [Clients].[ClientFromYear],
[Clients].[NoOfMeetings], [Clients].[DateEntered], [Clients].[IdEntered],
[Clients].[DateModified], [Clients].[IdModified]
FROM Clients
WHERE ((([Clients].[NoOfMeetings])=0));
```

Listing A-16 F **Running a Delete query**

```
Private Sub CmdDeleteQuery_Click()
   Dim db As DAO.Database
   Dim qdf As DAO.QueryDef

   On Error GoTo ExitProc

   Set db = CurrentDb
   Set qdf = db.QueryDefs("qryClientsWithoutMeeting")
    ' Chr(13) & Chr(13) is a double carriage return
   If (MsgBox("Do you want to perform the following: " & _
            Chr(13) & Chr(13) _
            & qdf.SQL, vbYesNo + vbDefaultButton2, _
            "SQL Expression")) = vbYes Then
       qdf.Execute dbFailOnError
       MsgBox qdf.RecordsAffected & " records were deleted."
   End If
ExitHere:
   Set db = Nothing
   Exit Sub
ExitProc:
      MsgBox "Unexpected error."
      Resume ExitHere
End Sub
```

Creating and Running a Make-Table Query with DAO

The Make-Table query creates a new table out of records from one or more tables or queries. Make-Table queries are often used to preserve data as it existed at a particular time or to create a backup copy of a table without backing up the entire database.

Use the SELECT INTO statement to create a make-table query. This statement can consist of the following parts:

SELECT fieldName	Field Name (use * for all fields)
INSERT newTableName	Name of the new table
FROM table/query Name	Name of a table or query from which data is taken
WHERE condition	Criteria / limit operation to desired rows
ORDER BY fieldName	Order the records in the new table

The SELECT INTO statement in cmdMakeATableQuery_Click (Listing A-16 G) is used to make a new table named SouthAmericanClients to store the names of all Brazilian customers from the Customers table in the Northwind database. Notice that by not assigning a name to the query, we create a temporary make-table query:

```
Set qdf = db.CreateQueryDef("", mySQL)
```

Listing A-16 G   Creating and running a Make-Table query

```
Private Sub cmdMakeATableQuery_Click()
   Dim db As DAO.Database
   Dim qdf As DAO.QueryDef
   Dim mySQL As String

   On Error GoTo Err_Handler

   mySQL = "SELECT * INTO SouthAmericanClients FROM Customers " & _
               "WHERE Country='Brazil';"
   Set db = OpenDatabase("C:\Program Files\Microsoft Office\" & _
         "Office11\Samples\Northwind.mdb")
   Set qdf = db.CreateQueryDef("", mySQL)
   qdf.Execute
ExitHere:
   Set db = Nothing
   Exit Sub
Err_Handler:
   MsgBox Err.Description
   Resume ExitHere
End Sub
```

Creating and Running an Append Query with DAO

Append queries are used for adding records from one or more tables to other tables. You can append records to a table in a current database or another Access or non-Access database.

An Append query is an action query that adds new records to the end of an existing table or query. Append queries don't return records. They are useful for archiving records. Before you can archive the records you need to create a new table structure to hold the records. To add a record or multiple records to a table, use the INSERT INTO statement. This statement has the following parts:

INSERT INTO target [(Field1, Field2)]	The name of the table or query to append records to. You may indicate the names of the fields to append data to.
SELECT fieldname	The names of fields to obtain data from.
FROM tableExpression	The name of the table or tables from which records are inserted, or the name of a saved query, or a SELECT statement.
WHERE condition	Criteria / limit operation to desired rows.

The procedure in Listing A-16 H demonstrates how to create and execute an append query using DAO.

Listing A-16 H Creating and running an Append query

```
Private Sub cmdAppend_Click()
   Dim db As DAO.Database
   Set db = OpenDatabase("C:\Program Files\Microsoft Office\" & _
         "Office11\Samples\Northwind.mdb")
   db.Execute "INSERT INTO SouthAmericanClients " _
               & "SELECT  * FROM Customers " _
               & "WHERE Country = 'Argentina'"
   db.Close
   MsgBox "Argentina clients have been appended."
   Set db = Nothing
End Sub
```

Running the above procedure more than once will produce duplicate records in the SouthAmericanClients table. You can clean up this table by creating a VBA procedure that eliminates duplicate records.

Creating and Running a Pass-Through Query with DAO

A Pass-through query works directly with an external ODBC (Open Database Connectivity) data source. Instead of linking to a table that resides on a server, you can send commands directly to the server to retrieve data.

To execute a SQL pass-through query, use the Connect property. If you do not specify a connection string in the Connect property, Access will ask you for the connection information every time you run the Pass-through query (and this can be very annoying).

The procedure in Listing A-16 I uses the MaxRecords property to return 15 records from the dbo.entity table located on a SQL server. Notice that the ReturnsRecords property is set to True. If your query does not need to return records, set the ReturnsRecords property to False.

Listing A-16 I   Creating a Pass-through query (Example 1)

```
Private Sub cmdPassThru_Click()
   Dim db As DAO.Database
   Dim qdfPass As DAO.QueryDef

   On Error GoTo err_PassThru
   Set db = CurrentDb
   Set qdfPass = db.CreateQueryDef("GetRecords")
   'enter your own connect string
   'suppy the server database name you want to connect to,
   'your User Id, password and the Data Source name
   qdfPass.Connect = "ODBC;Database=myDbName; " _
      & "UID=MILL;PWD=year00;DSN=myDataS"
   qdfPass.SQL = "Select * From dbo.entity"
   qdfPass.ReturnsRecords = True
   qdfPass.MaxRecords = 15
   DoCmd.OpenQuery "GetRecords"
   Exit Sub
err_PassThru:
   If Err.Number = 3151 Then
      MsgBox Err.Description
      Exit Sub
   End If
   db.QueryDefs.Delete "GetRecords"
   Resume 0
   Exit Sub
End Sub
```

Instead of displaying a datasheet with the records retrieved from the SQL database, the procedure below reads the records to a temporary query and proceeds to open a recordset based on that query. Next, the contents of two fields are printed to the Immediate window. Note that this procedure uses the OpenRecordset method which is covered later in this chapter.

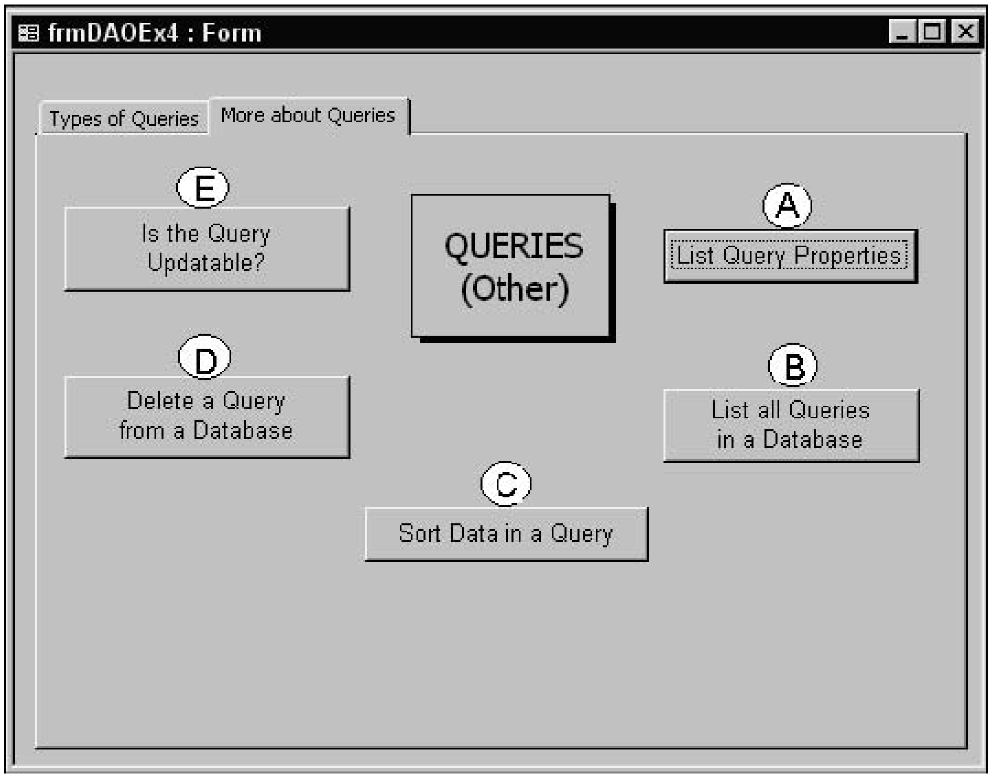
Listing A-16 J Creating and running a Pass-through query (Example 2)

```
Sub PassThru2()
   Dim db As DAO.Database
   Dim qdfPass As DAO.QueryDef
   Dim rstTemp As DAO.Recordset

   On Error GoTo err_PassThru

   Set db = CurrentDb
   Set qdfPass = db.CreateQueryDef("")

   qdfPass.Connect = "ODBC;Database=myDbName;UID=MILL;" _
      & "PWD=year00;DSN=myDataS"
   qdfPass.SQL = "Select * From dbo.entity"
   qdfPass.ReturnsRecords = True
   qdfPass.MaxRecords = 15

   Set rstTemp = qdfPass.OpenRecordset()
   'dump data from two fields to the Immediate window
   With rstTemp
      Do While Not .EOF
         Debug.Print , .Fields("entity_id"), .Fields("entity_name")
         .MoveNext
      Loop
      .Close
   End With
   SendKeys "^g"
ExitHere:
   Set db = Nothing
   Exit Sub
err_PassThru:
   MsgBox Err.Number & ":" & Err.Description
   Resume ExitHere
End Sub
```

Performing Other Operations with Queries

Now that you know how to programmatically create and run various queries, you may be interested to find out how to use Visual Basic to perform other operations related to queries, such as retrieving a list of queries and their properties, sorting data in queries, and so on.

Figure A-18 Programming other tasks related to queries.



Retrieving Query Properties

Just like tables and other database objects, queries have properties. To generate a list of properties for a specific query, use the For Each…Next looping structure to iterate through the Properties collection of a QueryDef object. The procedure in Listing A-18 A demonstrates this.

Listing A-18 A Listing query properties

```
Private Sub cmdListQueryProperties_Click()
   Dim db As DAO.Database
   Dim prp As DAO.Property

   On Error Resume Next
   Set db = OpenDatabase("C:\Program Files\" & _
            "Microsoft Office\Office11\Samples\Northwind.mdb")
   For Each prp In db.QueryDefs("Invoices").Properties
      Debug.Print prp.Name & "= " & prp.Value
   Next prp
   Set db = Nothing
End Sub
```

Below are some of the properties printed to the Immediate window by running the code in the cmdListQueryProperties_Click procedure above.

```
Name= Invoices
DateCreated= 9/13/1995 10:51:44 AM
LastUpdated= 3/12/2003 5:09:58 AM
Type= 0
SQL= SELECT Orders.ShipName, Orders.ShipAddress, Orders.ShipCity, Orders.ShipRegion,
Orders.ShipPostalCode, Orders.ShipCountry, Orders.CustomerID, Customers.CompanyName,
Customers.Address, Customers.City, Customers.Region, Customers.PostalCode, Customers.Country,
[FirstName] & " " & [LastName] AS Salesperson, Orders.OrderID, Orders.OrderDate, Orders.RequiredDate,
Orders.ShippedDate, Shippers.CompanyName, [Order Details].ProductID, Products.ProductName, [Order
Details].UnitPrice, [Order Details].Quantity, [Order Details].Discount, CCur([Order
Details].UnitPrice*[Quantity]*(1-[Discount])/100)*100 AS ExtendedPrice, Orders.Freight
FROM Shippers INNER JOIN (Products INNER JOIN ((Employees INNER JOIN (Customers INNER JOIN
Orders ON Customers.CustomerID=Orders.CustomerID) ON Employees.EmployeeID=Orders.EmployeeID)
INNER JOIN [Order Details] ON Orders.OrderID=[Order Details].OrderID) ON Products.ProductID=[Order
Details].ProductID) ON Shippers.ShipperID=Orders.ShipVia;

Updatable= True
Connect=
ReturnsRecords= True
ODBCTimeout= 60
RecordsAffected= 0
MaxRecords= 0
RecordLocks= 0
FilterOn= False
Description= (Criteria) Record source for Invoice report. Based on six tables. Includes expressions that
concatenate first and last employee name and that use the CCur function to calculate extended price.
OrderOn= True
DatasheetGridlinesBehavior= 3
OrderByOn= False
RecordsetType= 0
Orientation= 0
```

Listing All Queries in a Database

You can obtain the listing of all queries in a database by using the For…Each loop to enumerate the QueryDefs collection of the QueryDef. The procedure in Listing A-18 B writes to the Immediate window the names of all queries in the Northwind database. Before running this procedure, modify the path so it points to the valid location of the Northwind.mdb file on your computer.

Listing A-18 B Listing queries in a database

```
Private Sub cmdListAllQueries_Click()
   Dim db As DAO.Database
   Dim qdf As DAO.QueryDef

   Set db = OpenDatabase("C:\Program Files\" & _
            "Microsoft Office\Office11\Samples\Northwind.mdb")
   For Each qdf In db.QueryDefs
      Debug.Print qdf.Name
   Next qdf
   Set db = Nothing
End Sub
```

Open the Immediate window to view the names of queries retrieved by the cmdListAllQueries_Click procedure above.

Sorting Data in a Query

In the query design view you can specify the desired record order in a query results via the Sort cell in the column containing the field you want to sort by. The procedure in Listing A-18 C demonstrates how to create a query that returns records from the tblClients table sorted by the client's name.

The CreateQueryDef method creates a QueryDef object and assigns it the name that it will have when it is saved:

```
Set qdf = db.CreateQueryDef("qryClientsSorted")
```

Next, the SQL property of the QueryDef is set:

```
qdf.SQL = "SELECT * FROM tblClients ORDER BY ClientName"
```

Notice that the sort order is determined by the ORDER BY clause followed by the name of field (or fields) you want to sort by. Finally, the query is closed with the following statement:

```
qdf.Close
```

Run the cmdSortData_Click() procedure in Listing A-18 C. Switch to the database window and open the query to check the sort order. If the query does not appear in the list of queries, press F5 to refresh the Queries view in the Database window.

Listing A-18 C   Sorting data in a query

```
Private Sub cmdSortData_Click()
   Dim qdf As DAO.QueryDef

   On Error GoTo Err_SortData

      Set qdf = CurrentDb.CreateQueryDef("qryClientsSorted")
```

```
        qdf.SQL = "SELECT * FROM tblClients ORDER BY ClientName"
        qdf.Close
ExitHere:
    Exit Sub
Err_SortData:
        If Err.Number = 3012 Then
            CurrentDb.QueryDefs.Delete "qryClientsSorted"
            Resume 0
        Else
            MsgBox Err.Number & ": " & Err.Description
        End If
        Resume ExitHere
End Sub
```

Deleting a Query from a Database

To remove a QueryDef object from a QueryDef collection, use the Delete method as shown in Listing A-18 D. The cmdDeleteAQuery_Click deletes the query that was created by running the cmdSelectQuery_Click procedure in Listing A-16 A.

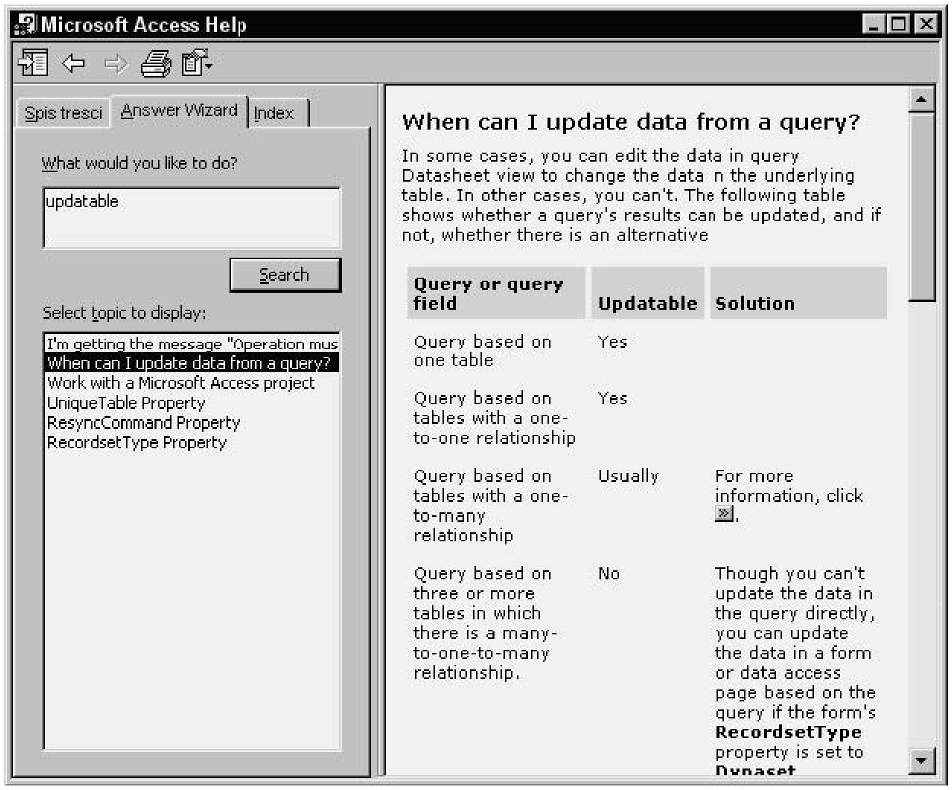
Listing A-18 D   Deleting a query from a database

```
Private Sub cmdDeleteAQuery_Click()
   Dim db As DAO.Database
   Dim qdf As DAO.QueryDef

   On Error GoTo ErrorHandler

   Set db = OpenDatabase("C:\Program Files\" & _
        "Microsoft Office\Office11\Samples\Northwind.mdb")
   db.QueryDefs.Delete "myQuery"
   db.Close

ExitHere:
  Set db = Nothing
  Exit Sub
ErrorHandler:
    MsgBox Err.Number & ": " & Err.Description
    Resume ExitHere
End Sub
```

Determining if a Query is Updatable

When a query is updatable you may edit the values in the result set of records and your changes are automatically reflected in the underlying tables. Microsoft Access online help lists situations when query results can or cannot be updated (Figure A-19). The QueryDef object has the Updatable property that you can use in your VBA code to find out if a query can be updated.

Figure A-19 Records returned by a query may or may not be updatable.



When can I update data from a query?

In some cases, you can edit the data in query Datasheet view to change the data n the underlying table. In other cases, you can't. The following table shows whether a query's results can be updated, and if not, whether there is an alternative

Query or query field	Updatable	Solution
Query based on one table	Yes	
Query based on tables with a one-to-one relationship	Yes	
Query based on tables with a one-to-many relationship	Usually	For more information, click ».
Query based on three or more tables in which there is a many-to-one-to-many relationship.	No	Though you can't update the data in the query directly, you can update the data in a form or data access page based on the query if the form's **RecordsetType** property is set to **Dynaset**

Listing A-18 E checks whether two queries in the Northwind database can be updated. The Updatable property returns True for the Invoices query, and False for the Order Subtotals query. The OpenRecordset method is used to open each of these queries.

Listing A-18 E Determining if a query is updatable

```
Private Sub cmdIsUpdatable_Click()
   Dim db As DAO.Database
   Dim rst As DAO.Recordset

   Set db = OpenDatabase("C:\Program Files\Microsoft Office\" & _
         "Office11\Samples\Northwind.mdb")
   Set rst = db.OpenRecordset("Order Subtotals")
   Debug.Print rst.Updatable
   Set rst = db.OpenRecordset("Invoices")
   Debug.Print rst.Updatable
   Set db = Nothing
   rst.Close
   Set rst = Nothing
End Sub
```

Finding and Reading Records

Working with database records requires that you understand the Recordset object. In DAO there are three types of Recordset objects: Table-type, Dynaset, and Snapshot. Each of these recordsets offers a different functionality (see Table A-4). The numerous procedures in this section demonstrate how to manipulate data using the Recordset object and its methods.

In general, a Recordset object is used to manipulate data from one or more tables or queries. Each column of a Recordset object represents a field, and each row represents a record. The Recordset is a temporary object and is not saved in the database. All Recordset objects cease to exist after the procedure ends. All open recordset objects are contained in the Recordset collection. You create a Recordset object using the OpenRecordset function.

Table A-4 Types of recordsets

Table recordsets	Used to access records in a table stored in Microsoft Access database (.mdb file). Table recordsets allow you to retrieve, add, update, and delete records in a single table. A Table-type recordset object stores references to underlying data in RAM (Random Access Memory).
Dynaset recordsets	Used to access records in a local table stored in Microsoft Access database (.mdb file), as well any table that is linked to an .mdb file. Using Dynasets, you can retrieve, add, update, and delete records from one or more tables. Dynaset recordsets contain indirect references to records in tables.
Snapshot recordsets	Used to access records from a local table stored in Microsoft Access database (.mdb file), as well as any linked table or a query. Snapshot recordsets contain a copy of the records in RAM and provide no direct access to the underlying data. They are used for reading data only — you can't use them to add, update, or delete records. A special type of a snapshot recordset known as Forward-Only, provides the fastest access to the data.

In order to find and read records, you must understand how to navigate through a Recordset object. When you open a Recordset object, the first record is the current record. All Recordsets have a current record.

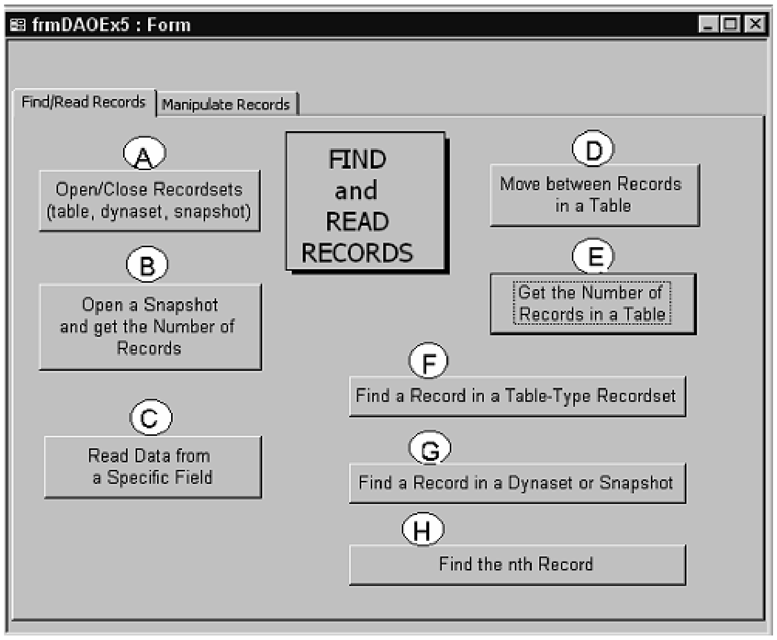
- To move to subsequent records, use the MoveNext method.
- To move to the previous record, use the MovePrevious method.
- The MoveFirst and MoveLast methods will move the cursor to the first and last records.
- If you call the MoveNext method when the cursor is already pointing to the last record, the cursor will move off the last record to the area known as end of file (EOF), and the EOF property will be set to True.

- If you call the MoveNext method when the EOF property is True, an error is generated, because you cannot move past the end of the file. Similarly, by calling the MovePrevious method when the cursor is pointing to the first record, you will move the cursor to the area known as beginning of file (BOF). This will set the BOF property to True. When the BOF property is True and you call the MovePrevious method, an error will be generated.

In the sections that follow, you will find many examples of navigating through recordsets. When navigating through a recordset you may want to mark a specifc record to return to it at a later time. You can use the Bookmark property to obtain a unique identification for a specific record. In a later section you will find out how to use other navigation methods to quickly locate a specific record or a set of records.

The Recordset object has numerous properties and methods. We will discuss only those properties and methods that are required for performing a specific task as demonstrated in the example procedures.

Figure A-20 Finding and reading records using code.


frmDAOEx5 : Form
Find/Read Records
Manipulate Records
A
Open/Close Recordsets (table, dynaset, snapshot)
FIND and READ RECORDS
D
Move between Records in a Table
B
Open a Snapshot and get the Number of Records
E
Get the Number of Records in a Table
F
Find a Record in a Table-Type Recordset
C
Read Data from a Specific Field
G
Find a Record in a Dynaset or Snapshot
H
Find the nth Record

Opening and Closing Various Types of Recordsets

Use the OpenRecordset method to create or open a Recordset. For example, to open a table-type recordset on a table named tblClients, use the following statement:

```
Set rst = CurrentDb.OpenRecordset(“tblClients”, dbOpenTable)
```

Notice that the second argument in the OpenRecordset method specifies the type of a Recordset. In this position, the following constants can be used:

dbOpenTable	Opens a table-type Recordset object
dbOpenDynaset	Opens a dynaset-type Recordset object
dbOpenSnapshot	Opens a snapshot-type Recordset object

If you don’t specify a Recordset type, a table-type Recordset will be created based on the tblClients. A table-type Recordset represents the records in a single table in a database.

The OpenRecordset method opens a new Recordset object for reading, adding, updating, or deleting records from a database. The OpenRecordset method can also be performed on a query. Note that a query can only be opened as a dynaset or snapshot recordset object. For example, to open a recordset based on a query, you could use the following statements:

```
Dim db As DAO.Database
Dim rst As DAO.Recordset

set db = CurrentDb()
Set rst = db.OpenRecordset(“qryMyQuery”, dbOpenSnapshot)
```

The procedure in Listing A-20 A demonstrates how to open various types of recordsets on the tblClients table and return the total number of records. Notice that to get the correct count of records in a Dynaset and Snapshot recordsets, you need to invoke the MoveNext method to access all the records. Counting records in covered in more detail in the next section.

Listing A-20 A Opening Table-, Dynaset-, and Snapshot-type recordsets

```
Private Sub cmd3Rst_Click()
   Dim tblRst As DAO.Recordset
   Dim dynaRst As DAO.Recordset
   Dim snapRst As DAO.Recordset

   Set tblRst = CurrentDb.OpenRecordset("tblClients", dbOpenTable)
   Debug.Print "# of records in a table: " & tblRst.RecordCount

   Set dynaRst = CurrentDb.OpenRecordset("tblClients", dbOpenDynaset)
   Debug.Print "# of records in a Dynaset: " & dynaRst.RecordCount
   dynaRst.MoveLast
   Debug.Print "# of records in a Dynaset:  " & dynaRst.RecordCount

   Set snapRst = CurrentDb.OpenRecordset("tblClients", dbOpenSnapshot)
   Debug.Print "# of records in a Snapshot: " & snapRst.RecordCount
   snapRst.MoveLast
   Debug.Print "# of records in a Snapshot: " & snapRst.RecordCount

   tblRst.Close
   dynaRst.Close
   snapRst.Close
   SendKeys "^g"
End Sub
```

Opening a Snapshot and Counting Records

When you want to search tables or queries, you will get the fastest results by opening a Snapshot-type recordset. A snapshot is simply a non-updateable set of records that contain fields from one or more tables or queries. Snapshot-type Recordset objects can be used only for retrieving data. Use the OpenRecordset method to create or open a recordset. For example, to open a Snapshot-type recordset on a table named tblClients, use the following statement:

```
Set rst = CurrentDb.OpenRecordset("tblClients", dbOpenSnapshot)
```

At times, you may need to know where you are in a recordset. There are two properties that can be used to determine your position in the recordset:

- The AbsolutePosition property allows you to position the current record pointer to a specific record based on its ordinal position in a dynaset or snapshot type recordset object. This property allows determining the current record number. Zero (0) refers to the first record in the Recordset object. If there is no current record, the AbsolutePosition property returns –1. However, because the position of a record changes when preceding records are deleted, you should rely more on bookmarks to position the current record. The AbsolutePosition property can be only used with Dynasets and Snapshots. Because the AbsolutePosition property value is zero-based, one is added to the AbsolutePosition value to display current record information:

  ```
  MsgBox "Current record: " & rst.AbsolutePosition + 1
  ```

- The PercentPosition property shows the current position relative to the number of records that have been accessed. Both the AbsolutePosition and PercentPosition are not accurate until you move to the last record.

The procedure in Listing A-20 B attempts to get the total number of records in the Snapshot-type recordset by using the RecordCount property. Dynaset-and Snapshot-type recordsets, the RecordCount property is the number of records accessed. Therefore, to determine the correct number of records in these recordsets you need to invoke the MoveLast method, to access all records.

Listing A-20 B Opening a Snapshot and retrieving the number of records

```
Private Sub cmdOpenSnapshot_Click()
   Dim rst As DAO.Recordset
   Set rst = CurrentDb.OpenRecordset("tblClients", dbOpenSnapshot)
   MsgBox "Current record: " & rst.AbsolutePosition + 1
   MsgBox "Number of records: " & rst.RecordCount
   rst.MoveLast
   MsgBox "Current record: " & rst.AbsolutePosition + 1
   MsgBox "Number of records: " & rst.RecordCount
   rst.Close
   Set rst = Nothing
End Sub
```

Retrieving the Contents of a Specific Field in a Table

To retrieve the contents of any field, start by creating a recordset based on the desired table or query, then loop through the recordset, printing the field’s contents for each record to the Immediate window.

The procedure in Listing A-20 C generates a listing of all clients in the tblClients table. Client names are retrieved starting from the last record (see the MoveLast method). The BOF property of the Recordset object determines when the beginning of your recordset was reached. The last statement in this procedure (SendKeys "^g") activates the Immediate window so that you can see the results for yourself.

Listing A-20 C Retrieving field values

```
Private Sub cmdReadFromEnd_Click()
   Dim db As DAO.Database
   Dim rst As DAO.Recordset

   Set db = CurrentDb
   Set rst = db.OpenRecordset("tblClients", dbOpenTable)
   rst.MoveLast
   Do Until rst.BOF
      Debug.Print rst!ClientName
      rst.MovePrevious
   Loop
   SendKeys "^g"
   rst.Close
   Set rst = Nothing
End Sub
```

Moving Between Records in a Table

All recordsets have a current position and a current record. A current record is usually the record at the current position. However, the current position can be before the first record and after the last record. You can use one of the following Move methods to change the current position:

MoveFirst	Moves to the first record.
MoveLast	Moves to the last record.
MoveNext	Moves to the next record.
MovePrevious	Moves to the previous record.
Move n	Move forward or backward n positions.

Listing A-20 D Moving between records in a table

```
Private Sub cmdNavigateRecords_Click()
   Dim db As DAO.Database
   Dim tblRst As DAO.Recordset
   Dim dynaRst As DAO.Recordset

   Set db = CurrentDb
   Set tblRst = db.OpenRecordset("tblClients")
   tblRst.Index = "ClientName"
   tblRst.MoveFirst

   Do While Not tblRst.EOF
      Debug.Print "Client: " & tblRst!ClientName
      tblRst.MoveNext
   Loop

   Set dynaRst = db.OpenRecordset("tblClients", dbOpenDynaset)
   dynaRst.MoveFirst
   Do While Not tblRst.EOF
      Debug.Print "Hello" & tblRst!ClientName
      tblRst.MoveNext
   Loop

   tblRst.Close
   dynaRst.Close
   Set tblRst = Nothing
   Set dynaRst = Nothing
   Set db = Nothing
   SendKeys "^g"
End Sub
```

Counting Records in a Recordset

The RecordCount property of a Recordset object returns the number of records that have been accessed. It is equal to zero (0), if there are no records in the recordset, and it is equal to 1, if there are records in a recordset. If you open a table-type recordset (see procedure in Listing A-20-E) and check the RecordCount property, it will return the total number of records in a table.

However, if you open a dynaset or snapshot type recordset, the RecordCount property will return 1, indicating that the recordset contains records. To find out the total number of records in a dynaset or snapshot, call the MoveLast method prior to retrieving the RecordCount property value. The record count becomes accurate after you've visited all the records in the recordset. Refer to the procedure cmd3Rst_Click in Listing A-20-A earlier in this appendix for an example.

Listing A-20 E   Retrieving the number of records in a Table-type recordset

```
Private Sub cmdRecCount_Click()
   Dim db As DAO.Database
   Dim rst As DAO.Recordset

   Set db = CurrentDb
   Set rst = db.OpenRecordset("tblClients", dbOpenTable)
   MsgBox "Number of Records: " & rst.RecordCount
   Set rst = Nothing
   Set db = Nothing
End Sub
```

Finding Records in a Table-Type Recordset

While the Move methods are convenient to loop through records in a Recordset object, you should use Seek or Find methods to look for specific records. When you know exactly which record you want to find in a Table-type recordset, and the field you are searching is indexed, the quickest way to find that record is by using the Seek method. One thing to remember with the Seek method is that the table must contain an index. The Index property must be set before the Seek method can be used. If you try to use the Seek method on a Table-type recordset without first setting the current index, a run-time error will occur. The Seek method searches through the recordset and locates the first matching record. Once the record is found it is made the current record and the NoMatch property is set to False. If the record is not found, the NoMatch property is set to True and the current record is undefined.

You can use the following comparison strings with the Seek method:

"="	Finds the first record whose indexed field is equal to the specified value.
">="	Finds the first record whose index field is greater than or equal to the

	specified value.
“>”	Finds the first record whose index field is greater than the specified value.
“<=”	Finds the first record whose index field is less than or equal to the specified value.
“<”	Finds the first record whose index field is less than the specified value.

The comparison operator used with the Seek method must be enclosed in quotes. If there are several records that match your criteria, the Seek method returns the first record it finds. The Seek method cannot be used to search for records in a linked table. You must use the Find methods (see the next section) for locating specific records in linked tables, as well as Dynaset-type and Snapshot-type recordsets.

Listing A-20 F Finding records in a Table-type recordset

```
Private Sub cmdFindRecords_Click()
   Dim db As DAO.Database
   Dim tblRst As DAO.Recordset

   Set db = CurrentDb
   Set tblRst = db.OpenRecordset("tblClients", dbOpenTable)
      'find the first client in the table whose name
      'begins with the letters "Ka"
      tblRst.Index = "ClientName"
      tblRst.Seek ">=", "Ka"
      If Not tblRst.NoMatch Then
         MsgBox "Found the following client: " & tblRst!ClientName
      Else
         MsgBox "There is no client with such a name."
      End If
   tblRst.Close
   db.Close
   Set tblRst = Nothing
   Set db = Nothing
End Sub
```

Finding Records in Dynasets or Snapshots

Use Find methods to search for a record in a Dynaset-type and Snapshot-type recordset. The following Find records are available:

FindFirst	Finds the first matching record in the recordset.
FindNext	Finds the next matching record, starting at the current record.
FindPrevious	Finds the previous matching record, starting at the current record.
FindLast	Finds the last matching record in the recordset.

If a record is not found for the given criteria, the NoMatch property is set to True.

Before searching for records, set a bookmark at the current record. If the search fails, you will be able to use the bookmark to return to the current record, otherwise you will get the error “No current record.” Each record in a Recordset object has a unique bookmark. You can use the bookmark to locate that record. To get the current record’s bookmark, you need to move the cursor to that record, and assign the value of the Bookmark property of the Recordset object to a variant variable:

```
Dim mySpot As Variant
mySpot = dynaRst.Bookmark
```

In the example procedure in Listing A-20 G, the bookmark is set on the first record of a Dynaset-type recordset. The procecure then searches for clients whose name contains the string "Sp." The names of all clients that match the search criteria are printed to the Immediate window. Next, we return to the bookmarked record by setting the Bookmark property to the value held by the variant variable:

```
dynaRst.Bookmark = mySpot
```

While recordsets based on local Microsoft Access tables support bookmarks, non-Access databases may not support them. To determine whether a Recordset object supports bookmarks, you can check the Bookmarkable property. Bookmarks are supported if this property is True.

```
If dynaRst.Bookmarkable Then
   mySpot = dynaRst.Bookmark
End If
```

If the Recordset object does not support bookmarks, an error occurs. You can set as many bookmarks as you wish. Bookmarks can be created for a record other than the current record by moving to the desired record and assigning the value of the Bookmark property to a String variable that identifies that record.

Listing A-20 G   Finding a record in a Dynaset

```
Private Sub cmdFindRecInDynaset_Click()
    Dim db As DAO.Database
    Dim dynaRst As DAO.Recordset
    Dim mySpot As Variant

    Set db = CurrentDb
    Set dynaRst = db.OpenRecordset("tblClients", dbOpenDynaset)
    MsgBox "Current client: " & dynaRst!ClientName
    mySpot = dynaRst.Bookmark
    'find clients whose name contains the following string "Sp."
    dynaRst.FindFirst "ClientName Like '*Sp.*'"

    Do While Not dynaRst.NoMatch
        Debug.Print dynaRst!ClientName
        dynaRst.FindNext "ClientName Like '*Sp.*'"
    Loop

    dynaRst.Bookmark = mySpot

    MsgBox "Back to record: " & dynaRst!ClientName
    dynaRst.Close
    db.Close
    Set dynaRst = Nothing
    Set db = Nothing
    SendKeys "^g"
End Sub
```

Finding the **n***th Record in a Dynaset or Snapshot*

The procedure in Listing A-20 H demonstrates how to locate the nth record in a Snapshot-type recordset. Notice that immediately after opening the recordset, the MoveLast method is used to ensure that all records have been visited. The total number of records is then stored in the totalRec variable. Next, the MoveFirst method returns to the first record and the InputBox method is used to prompt the user for the number of positions to move forward in the recordset. If the user-supplied value is less than the total number of records, the cursor moves to the specified record and the For…Each loop is used to print this record's field names and values to the Immediate window. An attempt to move beyond the end of the Recordset will cause an error. Therefore, the procedure displays a message if the user-supplied position to move to is greater than the total number of records.

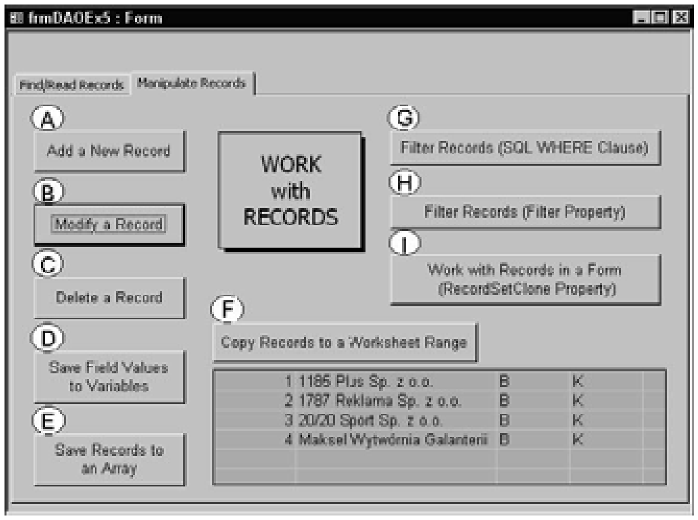
Listing A-20 H Finding the *n*th record in a Dynaset or Snapshot

```
Private Sub cmdFindRecSnap_Click()
    Dim rst As DAO.Recordset
    Dim fld As DAO.Field
    Dim totalRec As Integer
    Dim nth As String

    Set rst = CurrentDb.OpenRecordset("tblClients", dbOpenSnapshot)
    rst.MoveLast
    totalRec = rst.RecordCount
    rst.MoveFirst
    nth = InputBox("Enter the number of positions to move forward:")
        If totalRec > nth Then
            rst.Move nth
            For Each fld In rst.Fields
                Debug.Print fld.Name & ": " & fld.Value
            Next fld
        Else
            MsgBox "You must enter a value that is less than " _
                & totalRec & "."
        End If
    rst.Close
    Set rst = Nothing
End Sub
```

Working with Records

After learning the methods of moving through the recordset, finding a record and retrieving a record’s contents, it’s time to get to know methods that are used for adding, editing, and modifying data in a Table-type or Dynaset-type recordsets. Because Snapshot-type recordsets are static, you can’t change, delete, or add records to them.

Figure A-21 **Manipulating records with DAO programming.**


frmDAOEx5 : Form
Find/Read Records
Manipulate Records
A
Add a New Record
B
Modify a Record
C
Delete a Record
D
Save Field Values to Variables
E
Save Records to an Array
WORK with RECORDS
F
Copy Records to a Worksheet Range
G
Filter Records (SQL WHERE Clause)
H
Filter Records (Filter Property)
I
Work with Records in a Form (RecordSetClone Property)

```
1 1185 Plus Sp. z o.o.          B    K
2 1787 Reklama Sp. z o.o.       B    K
3 20/20 Sport Sp. z o.o.        B    K
4 Maksel Wytwórnia Galanterii   B    K
```

Adding New Records

In the Microsoft Access user interface, before you can add a new record to a table you must first open the appropriate table. In code, you simply open the Recordset object by calling the OpenRecordset method. For example, the following statements declare and open the Recordset object based on the table named tblClients2:

```
Dim tblRst As DAO. Recordset Set tblRst = db.OpenRecordset("tblClients2")
```

Once the Recordset object is open, use the AddNew method to create a blank record. For example:

```
tblRst.AddNew
```

Next, you may set values for all or some of the fields in the new record. You must set the Field's value if the Required property of a field is set to True. In the Microsoft Access user interface in Table Design view, there will be a Yes entry next to the Required property if the entry in the selected field is required. Here are some examples of setting field values in code:

```
tblRst.Fields("ClientName").Value = "Gosh, Regina"
tblRst.Fields("ClientType").Value = "I"
```

Note that because Value is the default property of a Field object, the use of this keyword is optional and it was omitted in the code of the example procedure in Listing A-21 A.

After filling in field values, you need to use the Update method on the Recordset object to ensure that the newly added record is saved:

```
tblRst.Update
```

Listing A-21 A demonstrates how to add a new record to the tblClients2 table and populate some of its fields with values.

Listing A-21 A   Adding a new record to an existing table

```
Private Sub cmdAddNewRec_Click()
   Dim db As DAO.Database
   Dim tblRst As DAO.Recordset

   Set db = CurrentDb
   Set tblRst = db.OpenRecordset("tblClients2")
   With tblRst
      .AddNew
         .Fields("ClientName") = "Gosh, Regina"
         .Fields("ClientType") = "I"
         .Fields("ClientFromYear") = "1999"
         .Fields("DateEntered") = Now()
         .Fields("IDEntered") = "JO"
         .Fields("DateModified") = Now()
         .Fields("IdModified") = "JO"
      .Update
   End With
   MsgBox tblRst.Fields(1).Value
```

```
    tblRst.Bookmark = tblRst.LastModified
    MsgBox tblRst.Fields(1).Value
    tblRst.Close
    db.Close
    Set tblRst = Nothing
    Set db = Nothing
End Sub
```

In a Table-type recordset, the new record is placed in the order identified by the table's index. In a Dynaset-type recordset, the new record is added at the end of the recordset. When you add a new record to a table, the new record does not become a current record. A record that was current prior to adding the new record remains current. In other words, while a new record is being added to the end of the table, the cursor remains in the record that was selected prior to adding a new record. You can, however, make the newly added record current by using a Bookmark and the LastModified property:

```
tblRst.Bookmark = tblRst.LastModified
```

To better understand the truth of the above statements, perform the following:

1. Open the tblClients2 table in the Microsoft Access user interface and delete the record for Regina Gosh that was added to this table when you ran the procedure in Listing A-21 A.

2. Close the tblClients2 table.

3. Modify the procedure cmdAddNewRec_Click in Listing A-21 A by typing the following statements below the End With keywords:

```
MsgBox tblRst.Fields(1).Value
tblRst.Bookmark = tblRst.LastModified
MsgBox tblRst.Fields(1).Value
```

4. Run the modified cmdAddNewRec_Click procedure.

Modifying Records

To edit an existing record, use the OpenRecordset method to open the Recordset object. Next, locate the record you want to modify. In a Table-type recordset, you can use the Seek method and a table index to find a record that meets your criteria. In a Dynaset-type and Snapshot-type recordsets, you can use any of the Find methods (FindFirst, FindNext, FindPrevious, FindLast) to locate the appropriate record. However, recall that you can edit data only in the Table-type or Dynaset-type recordsets (snapshots are used for retrieving data only). Once you've located the record, use the Edit method on the Recordset object, and proceed to change fields' values.When you are done with the record modification, invoke the Update method for the Recordset object.

The procedure cmdModifyRecord_Click in Listing A-21 B opens a Table-type recordset based on the tblClients2 table and changes the status of all clients from K to A. Next, the procedure locates a specific client record. Note that the Index property must be set before using the Seek method for searching the Table-type recordset. If you set the Index property to an index that doesn't exist, a run-time error will occur. Once the desired record is located, the procedure displays the client name in a message box as follows:

```
MsgBox rst!ClientName
```

The above statement is the same as:

```
MsgBox rst.Fields(“ClientName”).Value
```

The bang operator (!) is used to separate an object's name from the name of the collection of which it is a member. Because the default collection of the Recordset object is the Fields collection, you can omit the default collection name. Next, the procedure places the client record into an Edit mode, and changes the value of the ClientFromYear field:

```
rst!ClientFromYear = "1998"
```

This statement can be also written as:

```
rst.Fields(“ClientFromYear”).Value
```

The procedure then calls the Update method to make the field modification permanent. Finally, the Recordset is closed and the table is automatically opened in the Microsoft user interface as read-only.

Listing A-21 B   Modifying a record in a table

```
Private Sub cmdModifyRecord_Click()
   Dim db As DAO.Database
   Dim rst As DAO.Recordset
   Dim crit As String

   Set db = CurrentDb
   Set rst = db.OpenRecordset("tblClients2", dbOpenTable)
   rst.MoveFirst
   ' change the status of all clients from K to A (Active)
   Do While Not rst.EOF
        With rst
          .Edit
```

```
            .Fields("ClientStatus") = "A"
            .Update
            .MoveNext
        End With
    Loop
    ' find the record for Anna Korc - enter data in ClientFromYear field
    crit = "Korc, Anna"
    rst.MoveFirst
    rst.Index = "ClientName"
    rst.Seek "=", crit
    MsgBox rst!ClientName
    rst.Edit
    rst!ClientFromYear = "1998"
    rst.Update
    rst.Close
    db.Close
    Set rst = Nothing
    Set db = Nothing
    'open the modified table as Read-Only
    DoCmd.OpenTable "tblClients2", acViewNormal, acReadOnly
End Sub
```

Deleting Records

To delete an existing record, open the Recordset object by calling the OpenRecordset method. Locate the record you want to delete. In a Table-type recordset, you can use the Seek method and a table index to find a record that meets your criteria. In a Dynaset-type recordset, you can use any of the Find methods (FindFirst, FindNext, FindPrevious, FindLast) to locate the appropriate record. Next, use the Delete method on the Recordset object to perform the deletion. Before using the Delete method, it is a good idea to write code to ask the user to confirm or cancel the deletion. Immediately after a record is deleted, there is no current record. Use the MoveNext method to move the record pointer to an existing record.

The example procedure in Listing A-21 C deletes those clients who show the entry of zero (0) in the NoOfMeetings field. The statement, Do While Not tblRst.EOF tells Visual Basic to execute the statements inside the loop until the end of file is reached (EOF). The If statement inside the loop checks the value of the NoOfMeetings field, and deletes the current record only if the specified condition is True. Every time a record is deleted, the counter variable's value is increased by one. The counter variable stores the total number of deleted records. After the record is deleted, the MoveNext method is callled to move the record pointer to the next existing record as long as the end of file has not yet been reached. Even though you can use the Delete method and the While loop to remove the required records as shown in Listing A-21 C, it is more efficient to delete records with a delete query.

Listing A-21 C Deleting records

```
Private Sub cmdDeleteRecord_Click()
    'create a copy of tblClients as tblClients3

    Dim db As DAO.Database
    Dim tblRst As DAO.Recordset
    Dim counter As Integer
    'delete all the clients with zero (0) meetings

    Set db = CurrentDb
    Set tblRst = db.OpenRecordset("tblClients3")

    tblRst.MoveFirst
    Do While Not tblRst.EOF
        If tblRst!NoOfMeetings = 0 Then
            tblRst.Delete
            counter = counter + 1
        End If
        tblRst.MoveNext
    Loop

    MsgBox "Number of deleted records: " & counter
    tblRst.Close
    db.Close
    Set tblRst = Nothing
    Set db = Nothing
End Sub
```

Saving Field Values to Variables

Once you locate a particular record in a recordset, you can write individual field values to variables, and pass these values to other procedures. The variable must be of the appropriate data type. The example procedure in Listing A-21 D moves to the first record in a Table-type recordset and stores the values of the ClientName and NoOfMeetings fields in the variables named strClientName and intMeetings. Next, these variables are passed as parameters to the ShowData procedure that follows the Listing A-21 D. The ShowData procedure’s main purpose is to display the retrieved values to the user in a message box. Notice that the vbCr is a Visual Basic constant denoting the carriage return character. It is equivalent to Chr(13). Therefore, the statement,

```
MsgBox "Client: " & strClient & vbCr _
& "Meetings: " & intMeetings
```

can also be written as:

```
MsgBox "Client: " & strClient & Chr(13) _
& "Meetings: " & intMeetings
```

Listing A-21 D Saving data in variables

```
Private Sub cmdDataInVariables_Click()
   Dim db As DAO.Database
   Dim tblRst As DAO.Recordset
   Dim strClientName As String
   Dim intMeetings As Integer

   Set db = CurrentDb
   Set tblRst = db.OpenRecordset("tblClients")
   tblRst.MoveFirst
   strClientName = tblRst!ClientName
   intMeetings = tblRst!NoOfMeetings
   ShowData strClientName, intMeetings

   tblRst.Close
   db.Close
   Set tblRst = Nothing
   Set db = Nothing
End Sub

Sub ShowData(strClient As String, intMeetings As Integer)
   MsgBox "Client: " & strClient & vbCr _
   & "Meetings: " & intMeetings
End Sub
```

Saving Records to an Array

Sometimes you may need to retrieve the contents of one or more records. Using the GetRows method of the Recordset object you can copy a block of records from a Recordset object into a VBA array. The GetRows method accepts one argument which specifies the number or rows to be copied to the array. The procedure in Listing A-21 E opens a Dynaset-type recordset based on the tblClients table and calculates the total number of available records. After the record pointer is returned to the first record, the procedure prompts the user for the number of rows to copy. The user-supplied value is then used as the argument of the GetRows method. The records are returned to a Variant variable (myRecords). The GetRows method returns a two-dimensional array. If you run this procedure in the Debug mode, you can find out the values stored in the array by typing the following statements in the Immediate window:

```
?myRecords(0, 0)
```
1 (contents of the first field in the first record)

```
?myRecords(1, 1)
```
1787 Reklama Sp. z o.o. (contents of the second field in the second record)

```
?myRecords(2, 1)
```
B (contents of the third field in the second record)

In the statements shown above, the first subscript identifies the column (field) number, and the second identifies the row (record) number. Because by default arrays are zero-based, zero's in the column and row position indicate the first field in the first record.

Next, the procedure in Listing A-21 E uses the UBound function to calculate the upper bound indices of an array:

a) the total number of returned rows (records):

```
r = UBound(myRecords, 2) + 1
```

b) the total number of fields (columns) in a returned record:

```
c = UBound(myRecords, 1) + 1
```

Because by default arrays are zero-based, a value of 1 is added. Next, to make it more interesting and easier to understand, the procedure dumps the contents of the Variant variable (myRecords) to an Excel worksheet object (OLEExcel) that is embedded in the form (see Figure A-21 earlier in this chapter). To access the embedded Excel worksheet, the GetObject function is called:

```
Dim objEx As Object
Set objEx = GetObject(, "Excel.Application")
```

The next statement,

```
Me!OLEExcel.Action = acOLEActivate
```

uses the Action property in Visual Basic to specify the operation to perform on an OLE object. The constant, acOLEActivate, opens an OLE object for an operation, such as editing.

Next, Excel VBA statements are used to place the contents of the array in an appropriate worksheet range. The dumped data is then reselected and transposed. When the procedure ends, you are left with the Excel Worksheet in edit mode. Click outside the embedded worksheet object to exit the edit mode. Or, you can have the procedure exit the edit mode upon finishing by using the following statement:

```
Me!OLEExcel.Action = acOLEClose
```

The above statement closes an OLE object and ends the connection with the application that supplied the object. Using this setting is equivalent to clicking Close on the object's Control menu.

Listing A-21 E Saving records to an array

```
Private Sub cmdSaveToArray_Click()
   Dim db As DAO.Database
   Dim rst As DAO.Recordset
   Dim myRecords As Variant
   Dim count As Long
   Dim rowsToReturn As Integer
   Dim r As Integer
   Dim c As Integer
   Dim objEx As Object

   On Error GoTo ErrorHandler

   Set db = CurrentDb
   Set rst = db.OpenRecordset("tblClients", dbOpenDynaset)
   Set objEx = GetObject(, "Excel.Application")
   With rst
      .MoveLast
      count = .RecordCount
      .MoveFirst
   End With
   rowsToReturn = CInt(InputBox("How many records to retrieve?"))
   If rowsToReturn <= count Then
      myRecords = rst.GetRows(rowsToReturn)
      r = UBound(myRecords, 2) + 1
      c = UBound(myRecords, 1) + 1
      Me!OLEExcel.Action = acOLEActivate
      With objEx
         .Worksheets(1).Cells.Clear
         .Worksheets(1).Range(Cells(1, 1), _
               Cells(c, r)).Value = myRecords
         Selection.Copy
         .Worksheets(1).Range(Cells(1, 1), _
               Cells(c, r)).Copy
         .Worksheets(1).Range("A" & c + 2).Select
         Selection.PasteSpecial Paste:=xlAll, _
            Operation:=xlNone, _
            SkipBlanks:=False, Transpose:=True
         .Application.CutCopyMode = False
         .Worksheets(1).Range("A" & c + 2).Select
      End With
      ' Me!OLEExcel.Action = acOLEClose
   Else
      MsgBox "Too many rows to retrieve."
   End If
ExitHere:
   Set objEx = Nothing
   rst.Close
   db.Close
   Set rst = Nothing
   Set db = Nothing
   Exit Sub
ErrorHandler:
   Resume ExitHere
End Sub
```

Copying Records to a Worksheet Range

You can copy records directly to a worksheet range by using the CopyFromRecordset method.

- To copy all the records in the Recordset object to a worksheet range starting at cell A1, use the following statement:

  ```
  Worksheets(1).Range("A1").CopyFromRecordset rst
  ```

  The rst following the name of the method is an object variable representing a Recordset object.

- To copy five records to a worksheet range, use the following statement:

  ```
  Worksheets(1).Range("A1").CopyFromRecordset rst, 5
  ```

- To copy five records and four fields to a worksheet range, use the following statement:

  ```
  Worksheets(1).Range("A1").CopyFromRecordset rst, 5, 4
  ```

- You can also specify the number of records (rows) and fields to be copied using variables:

  ```
  Worksheets(1).Range("A1").CopyFromRecordset rst, myRows, myColumns
  ```

The procedure in Listing A-21 F uses the CopyFromRecordset method to copy data from a Recordset object created from the tblClients table. After prompting the user for the number of records to copy, the procedure dumps the contents of the first four (4) fields in those records to an Excel worksheet embedded in the Access form. Figure A-21 shows the result of running this procedure.

Listing A-21 F   Copying records to a worksheet range

```
Private Sub cmdCopyToExcelRange_Click()
   Dim db As DAO.Database
   Dim rst As DAO.Recordset
   Dim rowsToCopy As Integer
   Dim colNum As Integer
   Dim count As Integer
```

```
   Dim objEx As Object

   On Error GoTo ErrorHandler

   Set db = CurrentDb
   Set rst = db.OpenRecordset("tblClients", dbOpenDynaset)
   Set objEx = Me!OLEExcel.Object
   With rst
      .MoveLast
      count = .RecordCount
      .MoveFirst
   End With
   rowsToReturn = CInt(InputBox("How many records to copy?"))
      If rowsToReturn <= count Then
         Me!OLEExcel.Verb = acOLEVerbShow
         Me!OLEExcel.Action = acOLEActivate
         With objEx.Worksheets(1)
            .Cells.Clear
            .Range("A1").CopyFromRecordset rst, rowsToReturn, 4
            .Range("A1").Select
         End With
       ' keep the Excel object open so that you can check the results
       ' Me!OLEExcel.Action = acOLEClose
      End If
ExitHere:
   Set objEx = Nothing
   rst.Close
   db.Close
   Exit Sub
ErrorHandler:
   Resume ExitHere
End Sub
```

Filtering Records

When you want to work only with a certain subset of records you can filter out those records you don’t want to see. You can filter records using the SQL WHERE clause or you can use the Filter property. You can apply a filter to a Dynaset-type or Snapshot-type Recordset object. The fastest way to filter records is to open a new Recordset object by using an SQL statement that includes a WHERE clause. For example, Listing A-21 G provides an example of using the SQL WHERE clause to retrieve data for clients whose last name begins with the letter “B” and who had at least one meeting.

Listing A-21 G Filtering records using SQL WHERE clause

```
Private Sub cmdFilterSQLWhere_Click()
   Dim db As DAO.Database
   Dim rst As DAO.Recordset
   Dim qdf As DAO.QueryDef
   Dim mySQL As String

   Set db = CurrentDb
   mySQL = "SELECT * FROM " _
    & "tblClients WHERE ClientName LIKE 'B*' AND NoOfMeetings >0;"
   Set qdf = db.CreateQueryDef("BClients")
   qdf.SQL = mySQL
   Set rst = db.OpenRecordset("BClients")
   DoCmd.OpenQuery "BClients"
End Sub
```

Listing A-21 H uses the Filter propery to restrict the subset of records to those in which the client name begins with the letter “B” and the value in the NoOfMeetings field is greater than zero (0). The procedure begins with opening a Dynaset-type Recordset object based on the tblClients table and setting the Filter property on this recordset:

```
rst.Filter = "ClientName like 'B*' and NoOfMeetings >0"
```

For the filter to take effect after you set it, you must open a new recordset based on the Recordset object to which the filter was applied:

```
Set FilterRst = rst.OpenRecordset()
```

Next, the procedure writes to the Immediate window the value of the ClientName field for all of the records in the filtered recordset.

Listing A-21 H Filtering records using filter property

```
Private Sub cmdFilterData_Click()
   Dim db As DAO.Database
   Dim rst As DAO.Recordset
   Dim FilterRst As DAO.Recordset

   Set db = CurrentDb
   Set rst = db.OpenRecordset("tblClients", dbOpenDynaset)
   rst.Filter = "ClientName like 'B*' and NoOfMeetings >0"

   Set FilterRst = rst.OpenRecordset()
      Do Until FilterRst.EOF
         Debug.Print FilterRst.Fields("ClientName").Value
         FilterRst.MoveNext
      Loop
   rst.Close
   FilterRst.Close
   db.Close
   Set rst = Nothing
   Set FilterRst = Nothing
   Set db = Nothing
End Sub
```

Creating a Recordset Object from a Form

So far you have worked with Recordset objects that were created from tables or queries. You can also create a Recordset object based on a form. Recall that the source of the data for a form is specified by setting the form's RecordSource property to a table name, a query name, or an SQL statement. Use the RecordsetClone property to refer to the recordset underlying the form.

The RecordsetClone property is used for:

- Navigating and manipulating records without affecting the records displayed in the form.
- Synchronizing the form to the record found in the recordset clone.
- Accessing the RecordCount property of the Recordset object to determine the number of records displayed in the form.
- Determining the value in a particular field of the current record. Listing A-21 I shows an example of creating and using the RecordsetClone property.

Listing A-21 I Working with a recordset underlying a form

```
Private Sub cmdFindClientRecord_Click()
   Dim frmRst As DAO.Recordset
   Dim r As String

   DoCmd.OpenForm "frmClients", acNormal
   MsgBox "Form Name: " & Screen.ActiveForm.Name
   ' get the form and its recordset
   Set frmRst = Screen.ActiveForm.RecordsetClone
   ' find out how many records are in the Clients form
   ' and ask user which record to display
   r = InputBox("There are " & frmRst.RecordCount & " clients." & vbCr _
      & "Which client record would you like to display?", _
         "Enter Client Id", 1)
   If r = "" Then Exit Sub
   frmRst.FindFirst "[ClientId] = " & r
   If frmRst.NoMatch Then
      MsgBox "ClientId " & r & " does not exist in this table."
   Else
     ' show the value of the ClientName field
     ' of the current record in the clone
```

```
        MsgBox "Client Name: " & Forms!frmClients.RecordsetClone!ClientName
        ' display the found record in the form by setting the form's
        ' bookmark to the bookmark of the Recordset clone
        Forms!frmClients.Bookmark = frmRst.Bookmark
    End If
    frmRst.Close
    Set frmRst = Nothing
End Sub
```

Database Security and Transaction Processing

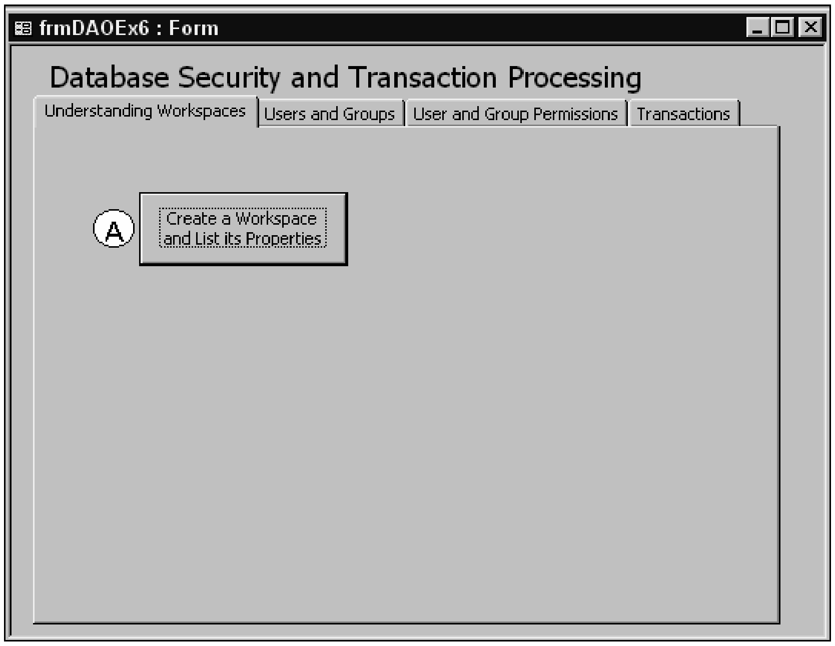
If your database contains confidential information, or you don't want others to change the code or modify the design of database objects, you should consider some method of protecting your application from unauthorized access. In addition to managing database security through Microsoft Access user interface via the options available from the Tools | Security menu, you can write VBA procedures to secure your application. The procedures that follow demonstrate how to use DAO to program security. This section also discusses how you can use transactions when making changes to large amounts of data.

Understanding Workspaces

To programmatically manage security and execute transactions, you need to learn about the Workspace object. As you know, in the Microsoft Access user interface you can only open a single database and log on as a single user at one time. When you start Access, the Jet database engine automatically logs you onto a default workspace (DBEngine.Workspaces(0)) as the Admin user and does not require that you specify a password. The following statements entered in the Immediate window return information about the current Microsoft Access user session:

?DBEngine.Workspaces(0).Count 1	How many workspaces are currently open?
?DBEngine.Workspaces(0).Name #Default Workspace#	What is the name of the currently open workspace object?
?DBEngine.Workspaces(0).UserName admin	Who is the owner of the Workspace object?

Figure A-22 Using the Workspace Object.


frmDAOEx6 : Form
Database Security and Transaction Processing
Understanding Workspaces
Users and Groups
User and Group Permissions
Transactions
A
Create a Workspace and List its Properties

Using the VBA code, you can:

- Create as many workspace objects as you need
- Log on as a different user
- Require to log on with a specific user ID
- Open a second database (When you open multiple workspaces you can open a different database in each workspace and manipulate data in those databases.)

The procedure in Listing A-22 A demonstrates how to create a workspace object using DAO and list its properties. The procedure begins by displaying the number of currently open workspaces. Next the procedure uses a For…Each Next loop to iterate through the collection of properties in the default workspace. The properties of the Workspace object are then written to the Immediate window as shown below:

```
Name = #Default Workspace#
UserName = admin
IsolateODBCTrans = 0
LoginTimeout =
DefaultCursorDriver =
Type = 2
```

Notice that some of the properties of the Workspace object are not set. To avoid an error when attempting to retrieve the value of these properties, the procedure uses the error handler QuickFix.

Next, the procedure uses the CreateWorkspace method to create a new workspace named “2ndWksp” while logged on as the Admin user.

```
Set myWorkspace = DBEngine.CreateWorkspace(Name:="2ndWksp",
UserName:="Admin", Password:="")
```

In the statement above, myWorkspace is an object variable that represents the Workspace object you want to create.

The required arguments of the CreateWorkspace method are as follows:

- Name is a string that uniquely names the new Workspace object.
- UserName is a string that identifies the owner of the new Workspace object.
- Password is a string containing the password for the new Workspace object. The password can be up to 14 characters long and can include any characters except ASCII character 0 (null).

The CreateWorkspace method can be used to log on to another workspace as a different user using different security permissions. Once the new workspace object is created with the CreateWorkspace method, you must use the Append method to add this object to the Workspaces collection:

```
DBEngine.Workspaces.Append myWorkspace
```

Use the Count method of the Workspaces collection to return the number of workspace objects in this collection. Once the new workspace object has been added to the Workspaces collection, the procedure lists its properties to the Immediate window by using the For…Each Next looping structure. The result is shown below:

```
Name = 2ndWksp
UserName = Admin
IsolateODBCTrans = 0
LoginTimeout =
DefaultCursorDriver =
Type = 2
```

Finally, the Close method is used to remove the Workspace object from the Workspaces collection:

```
myWorkspace.Close
```

Before removing a Workspace object from the Workspaces collection, make sure to close all open databases and connections.

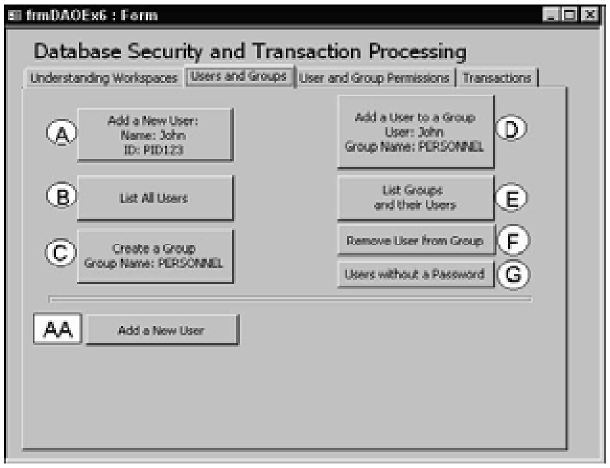
Listing A-22 A Creating a workspace and listing its properties

```
Private Sub cmdWorkspace_Click()
   Dim myWorkspace As DAO.Workspace
   Dim prp As DAO.Property

   On Error GoTo QuickFix
   MsgBox DBEngine.Workspaces.count

   For Each prp In DBEngine.Workspaces(0).Properties
      Debug.Print prp.Name & " = " & prp
   Next prp

   Set myWorkspace = DBEngine.CreateWorkspace(Name:="2ndWksp", _
      UserName:="admin", _
      Password:="")

      DBEngine.Workspaces.Append myWorkspace
      MsgBox DBEngine.Workspaces.count
      Debug.Print "-----------------------------"
      For Each prp In myWorkspace.Properties
         Debug.Print prp.Name & " = " & prp.Value
      Next prp
      myWorkspace.Close
      SendKeys "^g"
      Exit Sub
QuickFix:
      Debug.Print prp.Name & "="
      Resume Next
End Sub
```

Creating a New User Account

The user information is stored in the workgroup information file called System.mdw. Use the User object to establish individual user accounts in a VBA procedure. Use the CreateUser method of the Workspace object to create a new user. The CreateUser method accepts three arguments: userName, userPID, and userPassword. The last argument is optional.

Figure A-23 Setting up Users and Groups with DAO.


frmDAOEx6 : Form
Database Security and Transaction Processing
Understanding Workspaces
Users and Groups
User and Group Permissions
Transactions
A
Add a New User: Name: John ID: PID123
B
List All Users
C
Create a Group Group Name: PERSONNEL
D
Add a User to a Group User: John Group Name: PERSONNEL
E
List Groups and their Users
F
Remove User from Group
G
Users without a Password
AA
Add a New User

The procedure in Listing A-23 A that follows shows how to programmatically create a new user named John. When creating a new user, you must supply a name and a personal identifier (PID). Note that PID is not a password. In the example procedure, the following statement creates a new user account:

```
Set newUser = wsp.CreateUser(strUser)
```

The user name is defined by the contents of the strUser variable. The procedure then uses the PID property of the User object to specify a unique personal identifier for the new user:

```
newUser.PID = "SPC100000"
```

Next, the Password property is used to specify a password for the user. Instead of three lines of code, you can use the following statement:

```
Set newUser = wsp.CreateUser(strUser, "SPC100000", "gramofon")
```

Each new user must be added to the Users collection, using the Append method:

```
wsp.Users.Append newUser
```

If the specified user account already exists, an error will occur. To trap this error, the procedure in Listing A-23 A uses the error handler, which includes the following two statements:

```
wsp.Users.Delete strUser
Resume 0
```

The first statement above deletes the user name specified by the strUser from the Users collection. The second statement returns to the statement that caused the error.

Listing A-23 A Creating a new user Account

```
Private Sub cmdAddNewUser_Click()
   Dim wsp As DAO.Workspace
   Dim newUser As DAO.User
   Dim strUser As String

   On Error GoTo ErrorHandler
   Set wsp = DBEngine.Workspaces(0)
   strUser = "John"
   ' if the user account already exists, an error occurs
   Set newUser = wsp.CreateUser(strUser)
   newUser.PID = "SPC100000"
   newUser.Password = "gramofon"
   wsp.Users.Append newUser
   MsgBox "A new user was added."
   Exit Sub
ErrorHandler:
   wsp.Users.Delete strUser
   Resume 0
End Sub
```

Figure A-24 **The User and Group Accounts section in the Tools | Security submenu lists the new user account (John) that was created by running the VBA procedure in Listing A-23 A.**

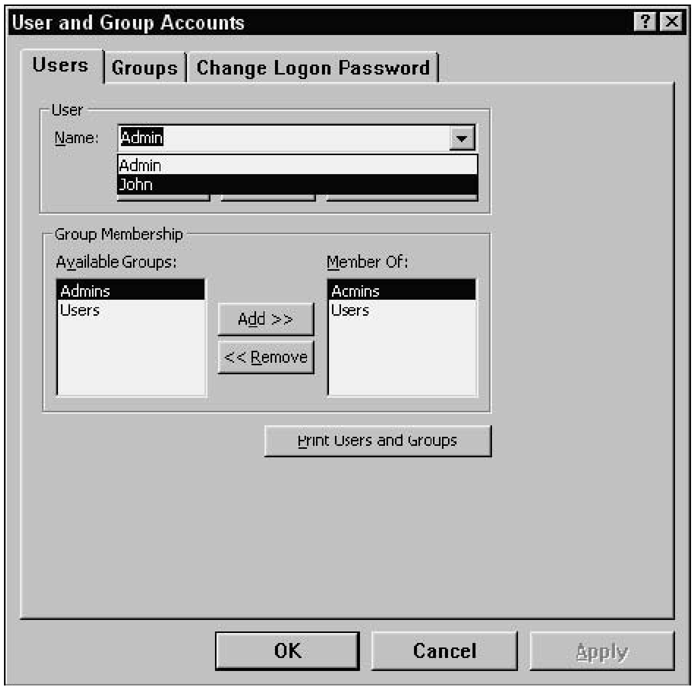
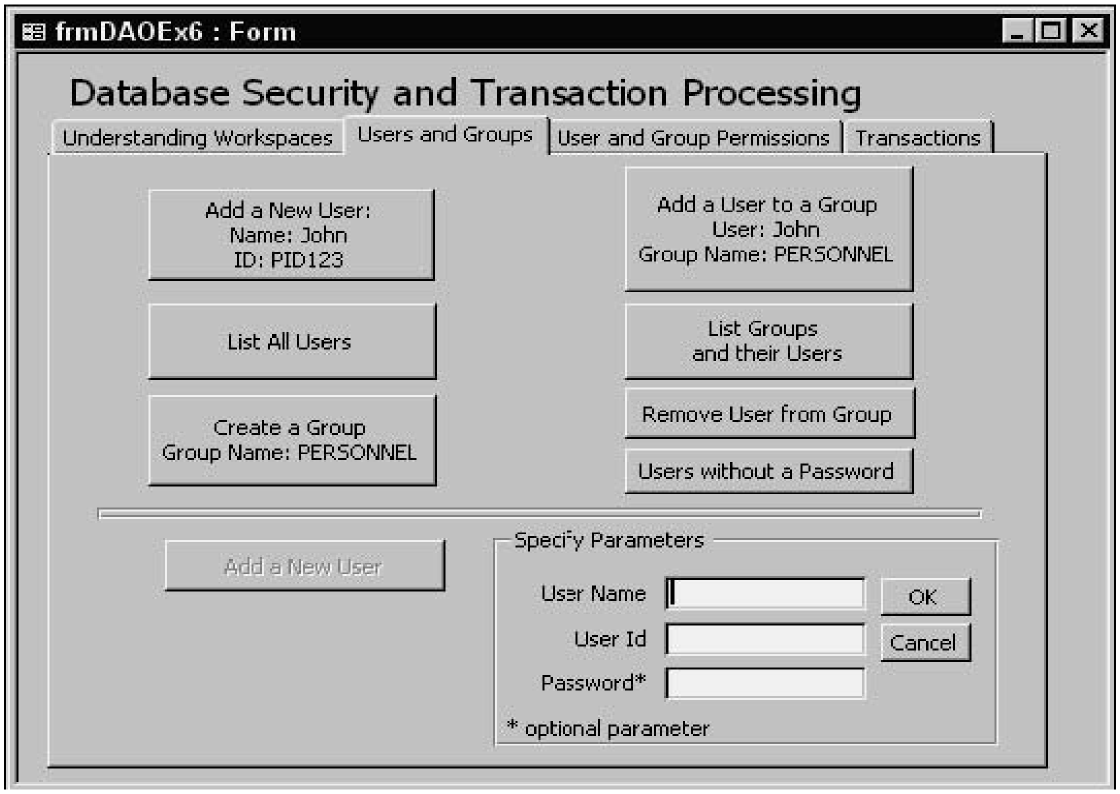


Figure A-25 **Using a custom form to create a New User Account.**


frmDAOEx6 : Form
Database Security and Transaction Processing
Understanding Workspaces
Users and Groups
User and Group Permissions
Transactions
Add a New User: Name: John ID: PID123
List All Users
Create a Group Group Name: PERSONNEL
Add a User to a Group User: John Group Name: PERSONNEL
List Groups and their Users
Remove User from Group
Users without a Password
Add a New User
Specify Parameters
User Name
User Id
Password*
OK
Cancel
* optional parameter

To speed up the process of creating new users, Figure A-23 earlier in this section includes a button named “Add a New User.” Upon clicking this button, the procedure makes visible several hidden controls that allow you to specify the arguments for the CreateUser method. Here’s the code for the button that is disabled in Figure A-25 above.

```
Private Sub cmdAddUser_Click()
   Dim ctrl As Control

   For Each ctrl In Me.Controls
      If ctrl.Properties("Tag").Value = "tgUserAccount" Then
         ctrl.Visible = True
      End If
   Next
   Me.txtName.SetFocus
   cmdAddUser.Enabled = False
End Sub
```

When you click the OK button (Figure A-25), the procedure in Listing A-23 A3 is called. This procedure first checks whether the required user data was supplied, and whether it is valid. You may want to add to it a code block validating the entry in the Password field (passwords can be up to 14 alphanumeric characters long). If everything is looks good, the procedure calls the CreateUserAccount function (see Listing A-23 A4) and passes to it the entries from the corresponding text boxes (txtName, txtPID, and txtPassword).

Listing A-23 A3 Adding a user account

```
' enter the following declaration at the top of the module

Option Compare Database
Dim Flag As Boolean

Private Sub cmdAdd_Click()

   'check if parameters were supplied
   If Me.txtName = "" Or Me.txtPID = "" Then
      MsgBox "You must enter User Name and User Id. ", _
         , "Required Parameters"
      Me.txtName.SetFocus
      Exit Sub
   End If
   If Len(Me.txtPID) < 4 Then
      MsgBox "You must enter at least 4 characters."
      Me.txtPID.SetFocus
      Exit Sub
   End If
      CreateUserAccount Me.txtName, Me.txtPID, Me.txtPassword
      If Not Flag Then
         MsgBox Me.txtName & " is a new user in Users group."
         Me.cmdAddUser.Enabled = True
      Else
         MsgBox "A user account with such a name already exists."
         Me.cmdAddUser.Enabled = False
         Me.txtName.SetFocus
      End If
         With Me
            .txtName.Value = ""
            .txtPID.Value = ""
            .txtPassword.Value = ""
         End With
End Sub
```

The procedure in Listing A-23 A3 above calls the CreateUserAccount function in Listing A-23 A4. Notice that this function can take three parameters. Recall that when a parameter is optional its name is preceded by the Optional keyword.

Listing A-23 A4 Creating a user account

```
Function CreateUserAccount(strUserName, strUserId, _
            Optional strUserPassword)
   Dim wsp As DAO.Workspace
```

```
    Dim newUser As DAO.User

    On Error GoTo ErrorHandler
    Set wsp = DBEngine.Workspaces(0)
    Set newUser = wsp.CreateUser(strUserName, _
        strUserId, strUserPassword)
    wsp.Users.Append newUser
    ' add a new user to the Users group
    newUser.Groups.Append wsp.CreateGroup("Users")
    Exit Function
ErrorHandler:
    MsgBox "A problem occurred."
    MsgBox Err.Number & ": " & Err.Description
    Flag = True
    Exit Function
End Function
```

You can cancel adding a new user, by clicking the Cancel button (see Figure A-25 earlier). The procedure shown below clears the entries in the text boxes, and hides controls that are used for supplying the new user's information.

Listing A-23 A5 Canceling the add new user operation

```
Private Sub cmdCancel_Click()
    Dim ctrl As Control
    On Error GoTo ErrorHandler
        With Me
            .txtName.Value = ""
            .txtPID.Value = ""
            .txtPassword.Value = ""
            .cmdAddUser.Enabled = True
            .cmdAdd.SetFocus
        End With
        For Each ctrl In Me.Controls
            If ctrl.Properties("Tag").Value = "tgUserAccount" Then
                ctrl.Visible = False
            End If
        Next
    Exit Sub
ErrorHandler:
    Me.cmdAddUser.SetFocus
    Resume 0
    Exit Sub
End Sub
```

Listing All Users

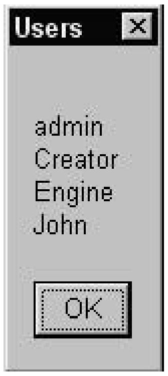
You can check the names of users you added by running the procedures in the previous section by opening the User and Group Accounts dialog box in the user interface (Figure A-24 in the previous section), or you can create a list of users programmatically.

The procedure in Listing A-23 B lists the names of all users in a message box (Figure A-26 below).

Listing A-23 B Listing all users

```
Private Sub cmdListUsers_Click()
   Dim wsp As DAO.Workspace
   Dim myUser As DAO.User
   Dim strUser As String

   Set wsp = DBEngine.Workspaces(0)
   For Each myUser In wsp.Users
      strUser = strUser & Chr(13) & myUser.Name
   Next
   MsgBox strUser, , "Users"
End Sub
```

Figure A-26 You can read the user names in code by retrieving the data from the Users collection.



```
admin
Creator
Engine
John
```

Creating a New Group Account

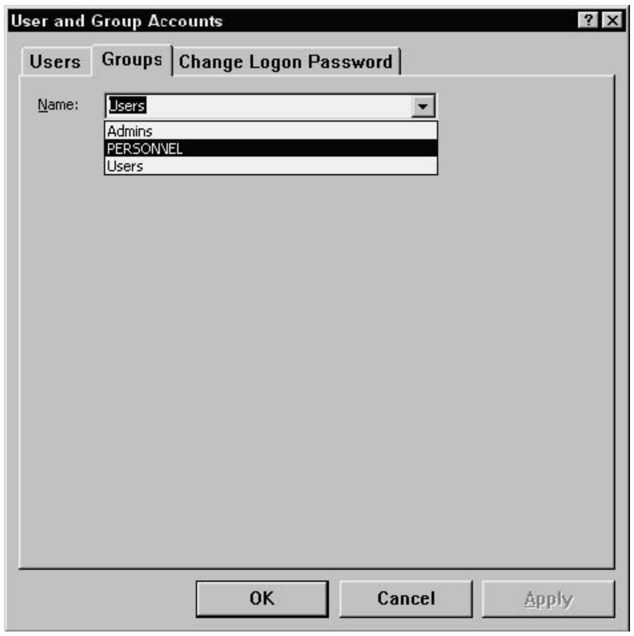
A Group object represents a group of user accounts that have common access permissions within a specific Workgroup. Each group can contain one or more users. Using the Group object, you can establish certain permissions for the group, and each user added to the group will assume permissions granted to the whole group. Microsoft Access has two default groups. The Admins group has permissions to everything. The Users group has full permissions on all newly-created objects. Similar to user accounts, the information about groups is stored in a Workgroup information file called System.mdw.

Listing A-23 C demonstrates how to create a User Group named PERSONNEL. Figure A-2 shows the result of running this procedure.

Listing A-23 C Creating a new group account

```
Private Sub cmdAddNewGroup_Click()
   Dim wsp As DAO.Workspace
   Dim newGroup As DAO.Group

   On Error GoTo ErrorHandler
   Set wsp = DBEngine.Workspaces(0)

   'create a group
   Set newGroup = wsp.CreateGroup("PERSONNEL", "PER50000")
   wsp.Groups.Append newGroup
   MsgBox "A new group was added."
   Exit Sub
ErrorHandler:
   wsp.Groups.Delete "PERSONNEL"
   Resume 0
End Sub
```

Figure A-27 The Personnel Group was added by the cmdAddNewGro up_Click() procedure in Listing A-23 C.



Adding a User to a Group

All new users should be added to an appropriate User group. Use the Append method to add the specified user object to the Users collection:

```
Set groupP = wsp.Groups("PERSONNEL")
```

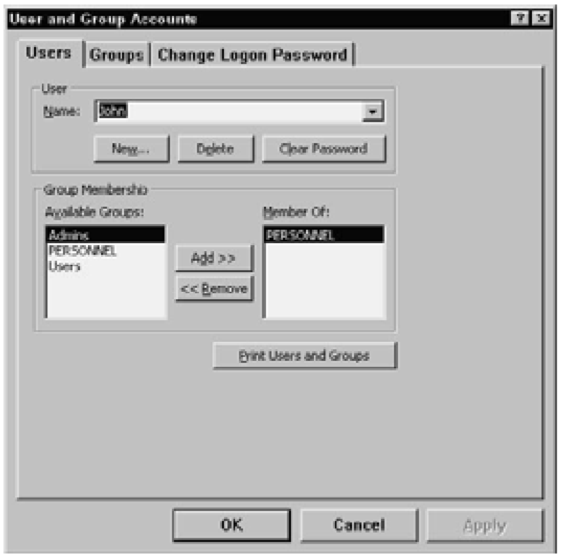
```
groupP.Users.Append
groupP.CreateUser("John")
```

The procedure in Listing A-23 D demonstrates how to add the user named John to the group named PERSONNEL. Prior to running this procedure, make sure that both the specified user and the group already exist (see Listing A-23 A for creating the User account, and Listing A-23 C for creating the Group account).

Listing A-23 D Adding a user to a group account

```
Private Sub cmdAddToGroup_Click()
   Dim wsp As DAO.Workspace
   Dim groupP As DAO.Group
   On Error GoTo ErrorHandler
   Set wsp = DBEngine.Workspaces(0)
   Set groupP = wsp.Groups("PERSONNEL")
   groupP.Users.Append groupP.CreateUser("John")
   Exit Sub
ErrorHandler:
   MsgBox Err.Number & ": " & Err.Description
End Sub
```

Figure A-28 A new user can be added to an existing group account via the User Interface or by running the VBA procedure in Listing A-23 D.



Listing User Accounts in Groups

A group account can contain one or more users. To create a list of users in each group account, run the procedure in Listing A-23 E.

Listing A-23 E Listing users in groups

```
Private Sub cmdListGroupsAndUsers_Click()
   Dim wsp As DAO.Workspace
   Dim myUser As DAO.User
   Dim myGroup As DAO.Group
   Dim thisGroup As DAO.Group

   Set wsp = DBEngine.Workspaces(0)
   For Each myGroup In wsp.Groups
      Debug.Print "Group: " & myGroup.Name
         Set thisGroup = wsp.Groups(myGroup.Name)
         If thisGroup.Users.count < 1 Then
            Debug.Print Chr(9) & "No users found"
         End If
         For Each myUser In thisGroup.Users
            Debug.Print Chr(9) & myUser.Name
         Next
   Next
End Sub
```

Removing Users from Groups

Deleting a user from an existing group is simple. All you need to do is specify the name of user you want to delete, and the name of group this user currently belongs to. Run the procedure in Listing A-23 F to remove any users you created by running previous procedures in this chapter. Notice that this procedure calls the DeleteFromGroup function to perform the deletion.

Listing A-23 F   Removing a user from a group

```
Private Sub cmdRemoveFromGroup_Click()
   Dim strUser As String
   Dim strGroup As String

   On Error GoTo ErrorHandler
   strUser = InputBox("Enter user name:", "Remove User")
   strGroup = InputBox("Enter group name", "Remove from Group")

   DeleteFromGroup strUser, strGroup
   Exit Sub
ErrorHandler:
      MsgBox Err.Number & ": " & Err.Description
   Exit Sub
End Sub

Function DeleteFromGroup(UserAccount, GroupAccount)
   Dim myGroup As DAO.Group
   Dim myUser As DAO.User
   Dim wsp As DAO.Workspace

   On Error GoTo ErrorHandler

   Set wsp = DBEngine(0)
   Set myUser = wsp.Users(UserAccount)

   myUser.Groups.Delete GroupAccount
   MsgBox UserAccount & " was removed from the " & _
      GroupAccount & " group."
   Exit Function
ErrorHandler:
      If Err.Number = 3265 Then
         MsgBox "There is no such user in the " & GroupAccount & " group."
      Else
         MsgBox Err.Number & ": " & Err.Description
      End If
   Exit Function
End Function
```

User Passwords

Recall that when you create a new user account you are not required to supply a password. If you created several users without a password, you can write a VBA procedure to find out which of these users don't have passwords. The procedure in Listing A-23 G uses the For…Each loop to obtain the names of users in the Users collection. To find out whether a user has a password, the procedure creates a new workspace and attempts to log onto it using the user's name (returned by the Name property of the User object) and a zero-length ("") password.

A successful login indicates that the user does not have a password, therefore the hasPassword variable is set to False. The user name is then printed to the Immediate window.

Listing A-23 G Obtaining a list of users without passwords

```
Private Sub cmdNoPassword_Click()
   Dim wksDef As DAO.Workspace        ' default Workspace
   Dim wks As DAO.Workspace           ' another Workspace
   Dim myUser As DAO.User
   Dim strUserName As String
   Dim hasPassword As Boolean

   On Error GoTo ErrorHandler
   Set wksDef = DBEngine.Workspaces(0)
   ' get the name of each user account in the Users collection
   For Each myUser In wksDef.Users
```

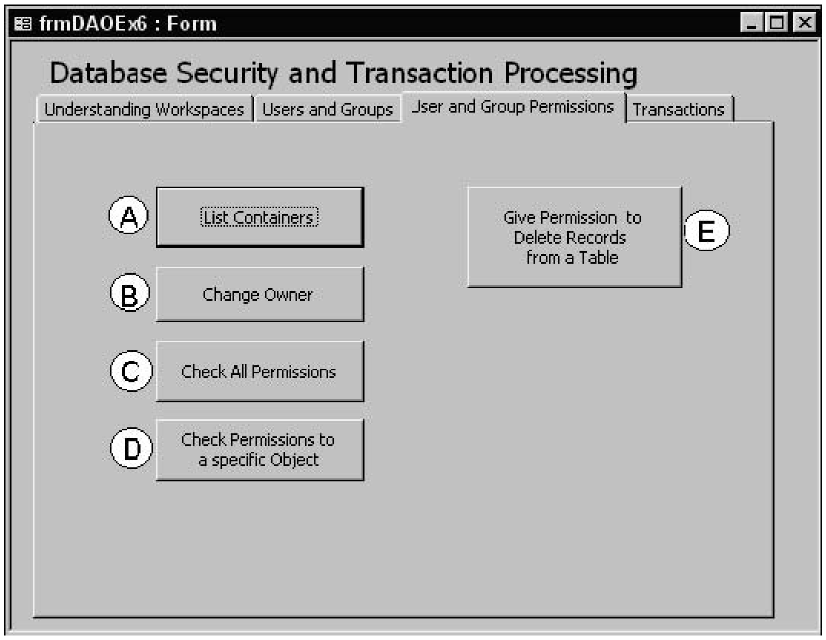
```
      ' store the user account name to the strUserName variable
      strUserName = myUser.Name
      If strUserName <> "Creator" And strUserName <> "Engine" Then
         hasPassword = False
         ' Try to log on to a new workspace as this user, using using
         ' a zero-length string password
         ' if the password is invalid, then goto ErrorHandler
         Set wks = DBEngine.CreateWorkspace("NoPassword", _
             strUserName, "")
         ' if you are able to log on, then the user does not
         ' have password
         If hasPassword = False Then
             Debug.Print "Without Password: " & strUserName
         End If
      End If
   Next
   Exit Sub
ErrorHandler:
   If Err.Number = 3029 Then   'user has password
       hasPassword = True
       Resume Next
   End If
   MsgBox Err.Number & ": " & Err.Description
End Sub
```

User and Group Permissions

In Microsoft Access, there are two types of permissions. Implicit permissions are those that you have as a member of a group. Explicit permissions are those that are assigned to you directly as a user. To simplify maintenance, you should assign permissions to groups, and then assign users membership in those groups. This will allow you to change permissions at the group level without having to do so for each user.

Before you can set user permissions and ownership of objects, you need to review few things you already know about Container objects and Document objects. Recall that in Microsoft Access, there is a Containers collection that stores the following container objects: Databases container, Tables container, Relations container, Forms container, Reports container, Module container, and Scripts container. A container object contains information about the database and about each of its saved forms, modules, relationships, reports, macros, tables and queries. The Container object has several useful properties that you will use for setting user permissions and ownership (Permissions property, Owner property, UserName property, and Inherit property). Each Container object has a Documents collection containing Document objects. Each Document object in turn includes information about a single document.

Figure A-29 **Managing user and group permissions.**


frmDAOEx6 : Form
Database Security and Transaction Processing
Understanding Workspaces
Users and Groups
User and Group Permissions
Transactions
A
List Containers
B
Change Owner
C
Check All Permissions
D
Check Permissions to a specific Object
Give Permission to Delete Records from a Table
E

The procedure in Listing A-29 A demonstrates how to loop through all the container objects within the current database and list the permissions in that container for the Admin user. The procedure also lists all the documents stored in each of the containers.

Listing A-29 A List containers, documents and admin user permissions

```
Private Sub cmdListContainers_Click()
  Dim myCont As DAO.Container
  Dim myDoc As DAO.Document
  Dim prp As DAO.Property

  For Each myCont In DBEngine(0)(0).Containers
    Debug.Print "Container: " & myCont.Name
    Debug.Print "Properties of " & myCont.Name
    Debug.Print "-----------------------------"
    For Each prp In myCont.Properties
      Debug.Print "    " & prp.Name _
          & " = " & prp
    Next prp
    Debug.Print "-----------------------------"
    Debug.Print "Documents in " & myCont.Name & " container:"
    For Each myDoc In myCont.Documents
      Debug.Print "Document Name: " & myDoc.Name _
      & "(" & myDoc.LastUpdated & ")"
    Next myDoc
    Debug.Print "============================="
  Next myCont
End Sub
```

After running the above procedure, open the Immediate window to view the output. A partial listing is shown below.

```
Container: Forms Properties of Forms
  ____________________________
    Name = Forms
    Owner = admin
    UserName = admin
    Permissions = 1048575
    AllPermissions = 1048575
    Inherit = True
______________________________
Documents in Forms container:
Document Name: frmClients(5/1/00 9:24:33 PM)
Document Name: frmDAOEx1(11/2/99 10:26:01 PM)
Document Name: frmDAOEx2(11/3/99 10:26:59 PM)
Document Name: frmDAOEx3(11/3/99 10:32:25 PM)
Document Name: frmDAOEx4(1/19/00 10:35:30 PM)
Document Name: frmDAOEx5(11/13/99 12:50:29 AM)
Document Name: frmDAOEx6(11/17/99 1:17:54 AM)
============================
```

Note: The Permissions property set to 1048575 indicates that the Admin has full permissions to the specified container object.

Changing the Owner of an Object

You can transfer ownership of an object (table, query, form, report, or script) to another user by choosing Tools | Security | User and Group Permissions, and activating the Change Owner tab, or you can do this programmatically, as shown in Listing A-29 B. This procedure demonstrates how to make John the new owner of the frmDAOEx1 form. This procedure calls the ChangeRights function. In order to check whether the user has permission to change the ownership of an object, you need to compare the document's AllPermissions property with the constant dbSecWriteOwner using the AND operator:

```
If (doc.AllPermissions And dbSecWriteOwner) <> 0 Then
```

If the result is True (<>0), the user is authorized to change ownership of objects.

Listing A-29 B   Changing the owner of a form

```
Private Sub cmdChangeRights_Click()
   Dim newOwner As String
   Dim ObjType As String
   Dim ObjName As String

   On Error GoTo ErrorHandler
   newOwner = "John"
   ObjType = "Forms"
   ObjName = "frmDAOEx1"
   ChangeRights newOwner, ObjType, ObjName
   Exit Sub
ErrorHandler:
   MsgBox Err.Number & ": " & Err.Description
End Sub

Function ChangeRights(newOwner, ObjType, ObjName)
   Dim db As DAO.Database
   Dim cont As DAO.Container
   Dim doc As DAO.Document

   On Error GoTo ErrorHandler
   Set db = CurrentDb
   Set cont = db.Containers(ObjType)
   Set doc = cont.Documents(ObjName)

   'check whether you are authorized to change ownership of objects
   If (doc.AllPermissions And dbSecWriteOwner) <> 0 Then
      doc.Owner = newOwner
      MsgBox "The new owner: " & newOwner, , ObjType & ": " _
         & ObjName
   Else
      MsgBox "You are not authorized to change ownership.", , _
          "Contact the database administrator."
      Exit Function
   End If
   Exit Function
ErrorHandler:
   MsgBox Err.Number & ": " & Err.Description
End Function
```

Checking Permissions to Objects

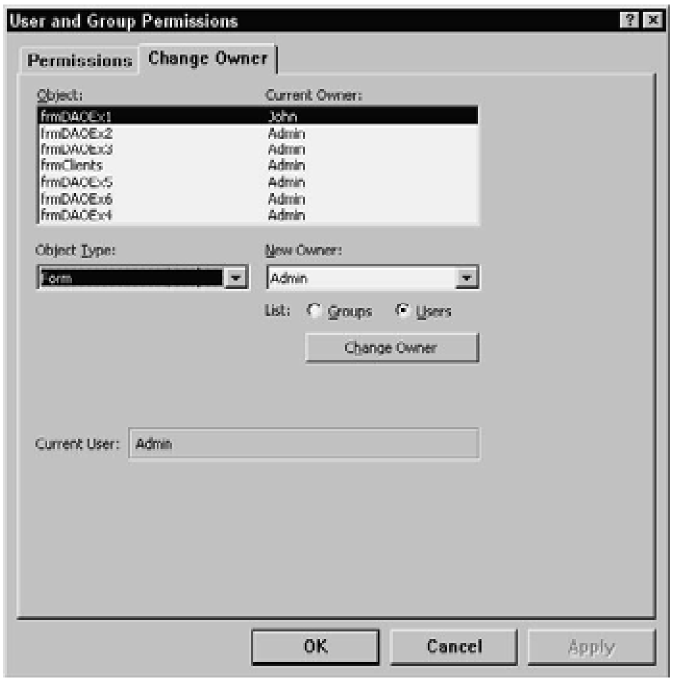
You can check to see whether a user has permissions to a specific object by browsing the User and Group Permissions dialog box (choose Tools | Security, User and Group Permissions), or run the procedure in Listing A-29 C. This procedure asks for the user name and loops through all Container objects and documents to list user’s permissions to each of the documents. Check the procedure output in the Immediate window. Zero (0) indicates that the user cannot access the object at all.

Listing A-29 C Checking user permissions

```
Private Sub cmdCheckPermissions_Click()
   Dim myCont As DAO.Container
```

```
    Dim myDoc As DAO.Document
    Dim strUserName As String

    strUserName = InputBox("Enter user name:", "Check Permissions")
    If strUserName = "" Then Exit Sub
    For Each myCont In DBEngine(0)(0).Containers
        Debug.Print "Container: " & myCont.Name
        For Each myDoc In myCont.Documents
            Debug.Print Chr(9) & "Document name: " & myDoc.Name
            myDoc.UserName = strUserName
            Debug.Print myDoc.Permissions
        Next
    Next
End Sub
```

Figure A-30 **A partial listing of all user permissions to objects as generated by the cmdCheck-Permissions_ Click() procedure in Listing A-29 C.**



Object	Current Owner
frmDAOEx1	John
frmDAOEx2	Admin
frmDAOEx3	Admin
frmClients	Admin
frmDAOEx5	Admin
frmDAOEx6	Admin
frmDAOEx4	Admin

Object Type: Form
New Owner: Admin
List: Groups [✓ Users]
Current User: Admin

Checking Permissions to a Specific Object

You can check programmaticaly whether a user has a certain permission to a specific object. For example, the procedure in Listing A-29 D demonstrates how to check whether the specified user is authorized to delete tblClients in the currently open database. To do this, the document's AllPermissions property is compared with the constant dbSecDeleteData using the AND operator:

```
If (myDoc.Permissions And dbSecDeleteData) > 0 Then
```

If the result of the If statement is True, the user is authorized to delete data from the specified table.

Listing A-29 D Checking permissions to a specific object

```
Private Sub cmdCheckDeletePermission_Click()
   Dim db As DAO.Database
   Dim myCont As DAO.Container
   Dim myDoc As DAO.Document
   Dim strUserName As String

   On Error GoTo ErrorHandler

   Set db = CurrentDb
   Set myCont = db.Containers("Tables")
   Set myDoc = myCont.Documents("tblClients")
   strUserName = InputBox("Enter user name", "Check Permissions")

      myDoc.UserName = strUserName
      If (myDoc.Permissions And dbSecDeleteData) > 0 Then
         Debug.Print Chr(9) & "Authorized"
      Else
         Debug.Print "Not Authorized"
      End If
      Debug.Print myDoc.Permissions
   Exit Sub
ErrorHandler:
   MsgBox Err.Number & ": " & Err.Description
End Sub
```

Assigning Specific Permissions to Groups

Listing A-29 E illustrates how to assign a delete permission for a table in the currently open database to the PERSONNEL group. Each member of the PERSONNEL group will have the right to delete the specified table from this database.

Listing A-29 E Assigning a delete permission to a group

```
Private Sub cmdGiveDeletePermission_Click()
   Dim db As DAO.Database
   Dim doc As DAO.Document
   Dim tlbRight As String
   Dim strUserOrGroup As String

   strUserOrGroup = "PERSONNEL"
```

```
    tblRight = "tblClients"

    Set db = CurrentDb
    Set doc = db.Containers("Tables").Documents(tblRight)
    doc.UserName = strUserOrGroup
    doc.Permissions = doc.Permissions Or dbSecDeleteData
End Sub
```

Transaction Processing

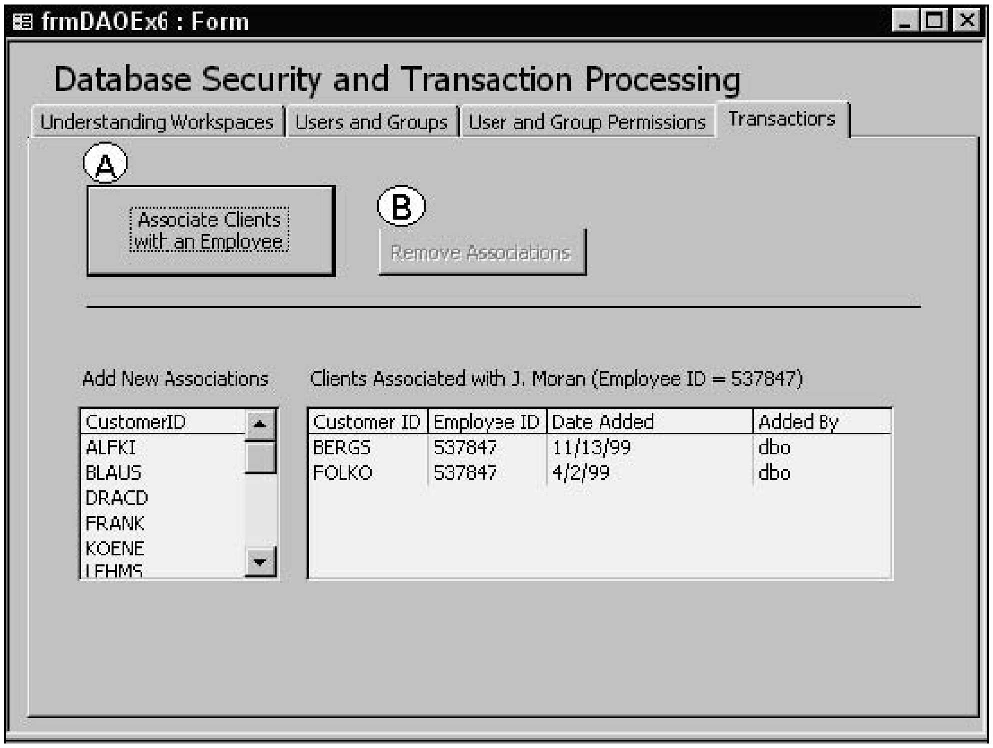
To improve your application's performance and to ensure that database activities can be recovered in case an unexpected hardware or software error occurs, you should group sets of activities into a transaction. Database transactions involve modifications and additions of one or more records in a single table or in several tables.

A transaction is a set of operations treated as a single unit. If you use an ATM (Automatic Teller Machine), you are already familiar with transaction processing. Every time you make cash withdrawal, the bank debits your checking or saving account and credits their cash account. For the transaction to be successful you must receive cash and your account must be debited. This transaction is a two-sided operation. If one side fails, the entire transaction fails. When a transaction has to be undone or cancelled, it is said that the transaction is rolled back. Often, when you perform batch updates to database tables and an error occurs, updates to all tables must be cancelled, or the database could be left in an *inconsistent state*.

To maintain database consistency, VBA supports transaction processing with three methods of the Workspace object: BeginTrans, CommitTrans, and Rollback.

Use the BeginTrans method to specify the beginning of a transaction. Use the CommitTran method to save the changes. The BeginTrans and CommitTrans are used in pairs. The data-modifying instructions you place between these keywords are stored in memory until Visual Basic encounters the CommitTrans statement. When the CommitTrans is reached, Access writes the changes that occurred since the BeginTrans to the disk, therefore any changes you've done in the tables become permanent. If an error is generated during the transaction process, the Rollback statement placed further down in your procedure will undo all changes made since the BeginTrans statement. The rollback ensures that the data is returned to the state it was in before you started the transaction. Because not all recordset objects support transactions, you should use the Transactions property of a Recordset object to see whether it supports transaction processing.

Figure A-31 Implementing transaction processing (Example 1).


frmDAOEx6 : Form
Database Security and Transaction Processing
Understanding Workspaces
Users and Groups
User and Group Permissions
Transactions
A
Associate Clients with an Employee
B
Remove Associations
Add New Associations
Clients Associated with J. Moran (Employee ID = 537847)

CustomerID
ALFKI
BLAUS
DRACD
FRANK
KOENE
LEHMS

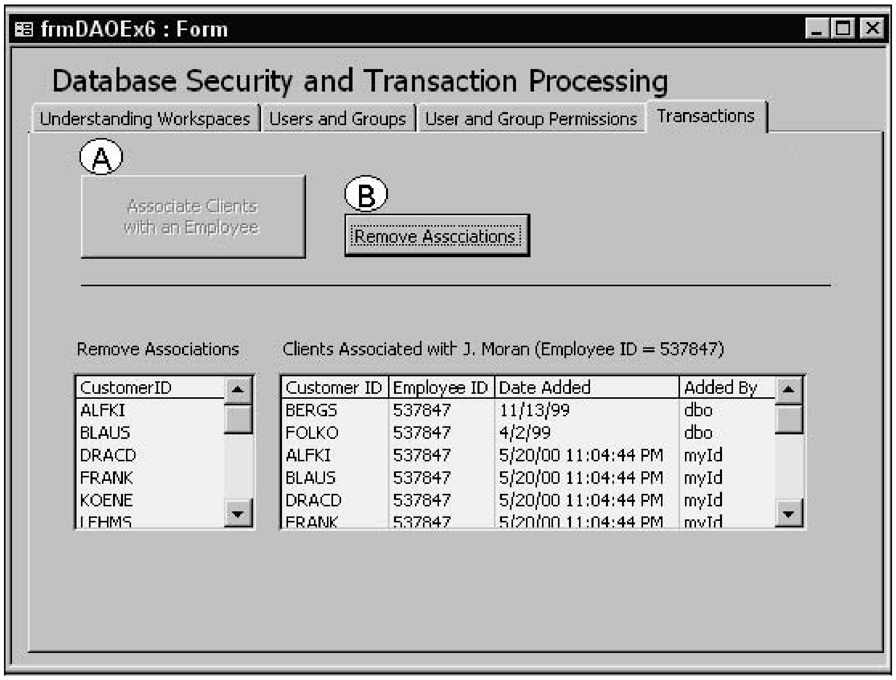
Customer ID	Employee ID	Date Added	Added By
BERGS	537847	11/13/99	dbo
FOLKO	537847	4/2/99	dbo

The procedure in Listing A-31 A demonstrates how you can use a transaction to add new records to a database table based on the information stored in another table. Table GermanClients contains client IDs that need to be associated with a specified Employee Id (537847). The procedure reads each record from the GermanClients table and creates a new record in the EmpAssociation table. This requires calling the AddNew method on the Recordset object and filling in data in four fields. Notice that this procedure contains two error-handling routines. The MainErrorHandler will take care of any errors that occur outside the transaction processing loop. Suppose you misspell the name of the GermanClients table. When you run the procedure, Visual Basic won't be able to fill the rst1 variable with the data from a non-existing table, therefore it will jump to the statements following the MainErrorHandler label. If an error occurs during transaction processing, program execution will jump to the error handling routine err_Rollback, and the Rollback instruction will be executed. The result of running the cmdAssociateClients_Click procedure in Listing A-31 A is illustrated in Figure A-32.

Listing A-31 A Executing a transaction (Example 1)

```
Private Sub cmdAssociateClients_Click()
   Dim myDb As DAO.Database
   Dim rst1 As DAO.Recordset
   Dim rst2 As DAO.Recordset
   Dim EmpID As String

   On Error GoTo MainErrorHandler
   Set myDb = CurrentDb
   Set rst1 = myDb.OpenRecordset("GermanClients")
   Set rst2 = myDb.OpenRecordset("EmpAssociation")
   EmpID = "537847"

  BeginTrans
   On Error GoTo err_Rollback
   With rst1
       Do Until .EOF
         Debug.Print rst1!CustomerID
           With rst2
               .AddNew
               rst2!CustomerID = rst1!CustomerID
               rst2!EmployeeID = EmpID
               rst2!DateAdded = Now()
       'uncomment next statement to force transaction rollback
               'rst2!UserAdded = "myId"
               rst2!AddedBy = "myId"
               .Update
           End With
           rst1.MoveNext
       Loop
   End With
CommitTrans
    rst1.Close
    Set myDb = Nothing
    Me.lboxAssociations.Requery
    Me.ChangeLabel.Caption = "Remove Associations"
    Me.cmdRemoveAssociations.Enabled = True
    Me.cmdRemoveAssociations.SetFocus
    Me.cmdAssociateClients.Enabled = False
   Exit Sub
MainErrorHandler:
   MsgBox Err.Number & ": " & Err.Description
   Exit Sub
err_Rollback:
   Rollback
   MsgBox "Transaction was rolled back.", _
       vbOKOnly + vbExclamation, "Unexpected Error"
   rst1.Close
   Set myDb = Nothing
End Sub
```

Figure A-32 Implementing transaction processing (Example 2).


frmDAOEx6 : Form
Database Security and Transaction Processing
Understanding Workspaces
Users and Groups
User and Group Permissions
Transactions
A
Associate Clients with an Employee
B
Remove Associations
Remove Associations
Clients Associated with J. Moran (Employee ID = 537847)

CustomerID
ALFKI
BLAUS
DRACD
FRANK
KOENE
LEHMS

Customer ID	Employee ID	Date Added	Added By
BERGS	537847	11/13/99	dbo
FOLKO	537847	4/2/99	dbo
ALFKI	537847	5/20/00 11:04:44 PM	myId
BLAUS	537847	5/20/00 11:04:44 PM	myId
DRACD	537847	5/20/00 11:04:44 PM	myId
FRANK	537847	5/20/00 11:04:44 PM	myId

The procedure in Listing A-31 B shows how you can use transaction processing to perform a bulk deletion of records. This procedure deletes from EmpAssociation table only the records that were added to this table when you ran the procedure in the previous listing.

Listing A-31 B Executing a transaction (Example 2)

```
Private Sub cmdRemoveAssociations_Click()
  Dim myDb As DAO.Database
  Dim rst1 As DAO.Recordset
  Dim EmpID As String
  Dim strSQL As String

  On Error GoTo err_RemoveAssociations
  Set myDb = CurrentDb
  Set rst1 = myDb.OpenRecordset("GermanClients")
  EmpID = "537847"

  BeginTrans
    Do Until rst1.EOF
      Debug.Print rst1!CustomerID
      strSQL = "DELETE FROM EmpAssociation " _
        & "WHERE ([CustomerID] = """ _
        & rst1!CustomerID & """) And" _
        & "(EmpAssociation.[EmployeeID]) = """ & _
          EmpID & """"
        Debug.Print strSQL
        myDb.Execute strSQL, dbFailOnError
        rst1.MoveNext
        Loop
   CommitTrans
   rst1.Close
   Set myDb = Nothing
   With Me
       .lboxAssociations.Requery
       .ChangeLabel.Caption = "Add New Associations"
       .cmdAssociateClients.Enabled = True
       .cmdAssociateClients.SetFocus
       .cmdRemoveAssociations.Enabled = False
    End With
    Exit Sub
err_RemoveAssociations:
   Rollback
   MsgBox "Transacton was rolled back.", _
       vbOKOnly + vbExclamation, "Unexpected Error"
 End Sub
```

Summary

This appendix has given you numerous, practical examples of using Microsoft Data Access Objects, commonly referred to as DAO. Although you can create an entire database with the information gained here, many of the discussed tasks are simply easier to accomplish through the user interface. Therefore, it is wiser to perform the easy tasks, such as creating queries or setting up relationships between tables, via the standard user interface, and use DAO programmable objects to automate more tedious tasks.