APPENDIX

C

Working with Data Access Objects

IN THIS APPENDIX

As mentioned in Chapter 3, “Making Access Project and Data Technology Choices,” you now have the choice of three technologies for manipulating data in VBA. All have extremely powerful programming models that let you control your database. The first technology, Data Access Objects (DAO), has been used for several versions of Access and is discussed in this appendix. The current version of DAO is 3.6, the same as in Access 2000. Because Microsoft’s aim is to use ADO now instead of DAO, the company has chosen not to put the resources into revving DAO.

Understanding Data Access Objects

If you read Chapter 4, “Working with Access Collections and Objects,” you’re already prepared for DAO and any other object model that might come your way. All the concepts presented about properties, methods, and collections work the same way. The only changes are the names of the objects and the types of members they contain.

Chapter 13, “Driving Office Applications with Automation,” focuses on programming Excel, Word, and other VBA-supported applications through Automation. Again, using these object models with DAO just requires that you understand the different types of objects and their specific properties and methods; you don’t need to learn any new basic concepts.

DAO is one of Microsoft’s models for accessing databases. This object model coexists with the Access programming model. However, the DAO feature is also a separate component available in all Microsoft applications that support VBA. Therefore, you can use what you learn in this appendix for programming in Office 2002 as well as in Visual Basic 4 and greater.

DAO provides a consistent object-oriented interface for performing all database-related functions. Although this might sound intimidating, DAO is not only extremely powerful, but is also very easy to use. Some of its features are as follows:

- The capability to create and edit databases, tables, queries, indexes, fields, referential integrity rules, and security.
- The capability to access data by using SQL. Methods are also available for navigating and searching through the data in tables and data resulting from queries.
- Support for beginning, committing, and canceling transactions. Transactions, which can be nested, are useful for performing bulk operations on the database as a single action.
- The capability to create custom properties on any DAO.
- The capability to repair and compact databases from the programming language.
- Support for attaching remote tables to your database, as well as for managing the connection.

Understanding Your Database's Anatomy

Although you consider Access to be your database application, the actual database engine is in a Microsoft component called Jet. This component handles all aspects of your database. Access is merely the way Jet is exposed graphically to users.

Because Jet is a separate component, other applications can use it. To make interaction with the database engine simpler, the DAO programming model is exposed. DAO is an Automation interface for accessing the Jet engine. In simple terms, this means that any application that supports Automation—including all Microsoft applications that support VBA—can access individual DAO and, hence, the database engine.

If you understand this relationship, you can see why DAO isn't part of the Access object hierarchy and why the Access hierarchy consists mainly of user-interface–related elements (your forms and reports). However, because a close relationship exists between Access and DAO, you need to learn about many DAO properties that serve specific functions within Access. You also learn how to create your own custom-defined properties and add them to your database.

NOTE

DAO has been referred to as an *object-oriented programming model*. Don't confuse this term with *object-oriented database*. Access isn't an object-oriented database; it's a relational database. Object-oriented databases are fundamentally different from relational databases, and the two shouldn't be confused.

As an object-based programming model, DAO exposes collections, properties, and methods for manipulating the individual components. All these concepts are discussed in this appendix. By now, you should begin to see how the basic VBA concepts are being carried throughout each component of Access.

When examining DAO, it helps to analyze the composition of your database. Databases in Access are composed of the following pieces:

Tables	Reports
Queries	Code modules and macros
Forms	Security information
Data Access Pages	Relationships

All these elements make up your database. DAO provides an interface to obtain a tremendous amount of information about each element. You can obtain information about a table's definition, all its fields, indexes, and relationships. You can quickly and easily add or delete more tables, queries, or relationships. You can even add to these objects custom properties that are saved with the database. All this power is easily accessible through that consistent programming paradigm known as DAO.

Getting Started with DAO

Because DAO consists of such a large number of objects, it helps to see a road map of how all these objects relate. As with most object models, DAO has a distinct object hierarchy. In Access you start with the `Application` object, whereas in DAO you start with `DBEngine`. In this section, you learn how to maneuver through the object hierarchy by manipulating each object.

Figure C.1 is a diagram representing the DAO object hierarchy. This diagram demonstrates the *hierarchy* or tree of objects that DAO exposes. From Figure C.1, you can see the numerous available objects and the path you must follow to reach them.

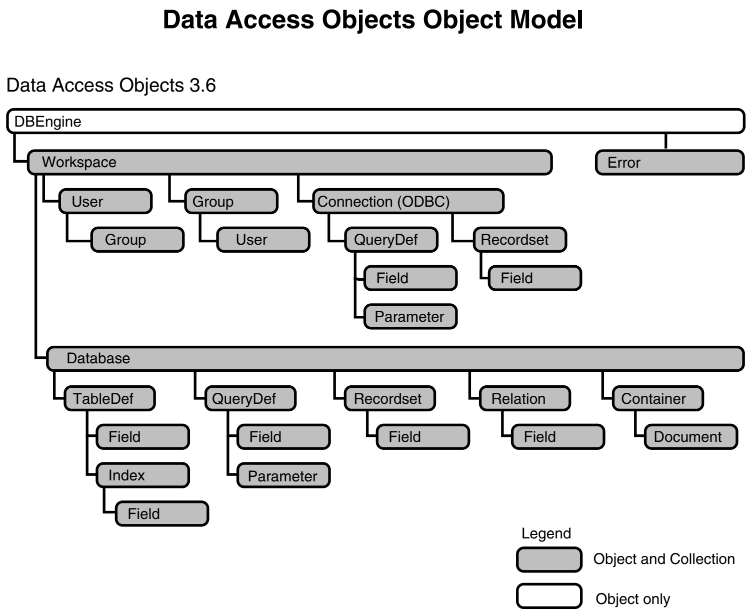


FIGURE C.1
The DAO object hierarchy starts with `DBEngine`, through which all other objects must be referenced.

In Figure C.1, you can see that `DBEngine` is the root object and consists of its own set of properties and methods, as well as a collection of objects called *workspaces*. Workspaces allow you to manage your open databases, open more databases, and create transactions. They have their own set of properties and methods, as well as a collection of databases. After retrieving `DBEngine` and the `Workspace` object, you can access your database. You do most of your work in the `Database` object. The Database collection, as you can guess by now, consists of its own set of properties and methods. It also contains collections of different objects that define your database. An .mdb database file corresponds to the `Database` object and everything below it in the hierarchy.

The next sections introduce you to how to manipulate your database by using the `Database` object and all its collections. When reading the following sections, it might help to refer to Figure C.1, which gives you directions to every object available in DAO. Figure C.1 shows no shortcuts, so you must first stop at any objects along the way to get to the specific object you're interested in.

Getting into Your Database

All programs must start somewhere. In Access, you often have an `AutoExec` macro that starts your application. In the DAO world, you also have an entry point—the `DBEngine` object. At this entry point, you can obtain a reference to your database. This is where all your journeys into DAO must start. From here, you can travel to any object in the hierarchy.

When Access starts up, it initializes the database engine and creates an initial workspace. Opening a database sets up a `Database` object for you. To reach your `Database` object, you must first reference the engine and the workspace as follows:

```
Dim dbsCurrentDatabase as Database
Set dbsCurrentDatabase = CurrentDb()
```

NOTE

If you're an Access 2 developer, you might be wondering why the `CurrentDb()` function is being recommended because it was marked as obsolete from Access 1. `CurrentDb()` made a comeback in Access 95 and is the preferred way of accessing your database. `CurrentDb()` returns a second instance of the database, which helps minimize possible conflicts in multiuser scenarios.

You can use the `Database` object for

- Examining queries and tables
- Obtaining information about your database's relationships
- Enumerating the forms, reports, and macros in the database
- Executing an action query or passing a SQL string to an ODBC database

NOTE

Action queries modify the contents of the database but don't return a recordset. Chapter 8, "Using Queries to Get the Most Out of Your Data," discusses in detail the four types of action queries: make table, delete, append, and update

- Opening recordsets to create recordsets of existing tables, queries, or even custom SQL statements
- Closing the database

Examining Your Database

By using your knowledge of collections, you can easily list all the tables and queries in your database. Listing C.1 shows how easily it can be done.

LISTING C.1 WebC.mdb: Listing All Tables and Queries in Your Database

```
Sub ViewTablesAndQueries()
   Dim dbsCurrent As DATABASE, tdfTable As TableDef, qdfQuery As QueryDef
   Set dbsCurrent = CurrentDb()

   For Each tdfTable In dbsCurrent.TableDefs
      Debug.Print tdfTable.Name
   Next

   For Each qdfQuery In dbsCurrent.QueryDefs
      Debug.Print qdfQuery.Name
   Next

End Sub
```

This subroutine outputs every table and query in your database. It does nothing to distinguish between system and hidden tables or action queries. Often, when manipulating tables or queries as a whole, you want to skip action queries as well as hidden and system tables.

TIP

In Access 2002, you can access all Access objects through the `CurrentProject` and `CurrentData` objects off the `Application` object. This way, you can use the same code in DAO or ADO. For more information, see Chapter 4.

Examining Table Attributes

This section shows you how to use properties of a DAO collection. The example uses the `TableDef`'s `Attributes` property to skip over any hidden or system tables.

Hidden tables and *system tables* are tables that Access requires for its own internal management. By default, Access doesn't list these tables in the table view. To view system and hidden objects, you must change a few options:

1. From the Tools menu, choose Options.
2. Select the View tab.
3. Select the Hidden Objects and System Objects options.
4. Click Apply and then click OK.

Now when you return to your database view, you should see a list of system tables prefixed by `MSys`. When you enumerate all tables in a database, these tables are listed. To eliminate these tables, you can try ignoring tables that begin with `MSys`. However, this does nothing to help you ignore *hidden tables* (temporary tables or your own system tables created while using your database). You also can never guarantee that other users won't use the same prefix in their tables. Because these methods don't work, there must be a better way—and there is by using the `Attributes` property.

Most objects in DAO have attributes, which vary from object to object. The attributes on a table provide information about whether and how the table is attached, as well as whether it is a system or hidden table.

Rewriting the code that lists the tables to ignore system and hidden tables is done as shown in Listing C.2.

LISTING C.2 WebC.mdb: Ignoring System and Hidden Tables

```
Sub ViewTables()
   Dim dbsCurrent As DATABASE, tdfTable As TableDef
   Set dbsCurrent = CurrentDb()

   For Each tdfTable In dbsCurrent.TableDefs
      If ((tdfTable.Attributes And dbSystemObject) Or _
           (tdfTable.Attributes And dbHiddenObject)) Then
           ' Ignore these tables
      Else
         Debug.Print tdfTable.Name
      End If
   Next

End Sub
```

Notice the use of the logical `And` operator instead of a test for equivalency with `tdfTable.Attributes = dbSystemObject`. You're doing a bitwise comparison between the two values, as the attribute field can contain a combination of settings.

Because Access 2002 provides constants for each attribute, you don't need to declare your own constants. You need to use only the existing ones.

Examining Query Types

Just as you often want to ignore system and hidden tables, you might also want to ignore or locate only action queries. Action queries modify your data and don't return a recordset. They are executed rather than opened.

Determining the type of query is done through the `Type` property. The routine in Listing C.3 returns select queries, which return a set of records to your form, datasheet, or report. Many select queries allow you to modify the base tables from the datasheet or report.

LISTING C.3 WebC.mdb: Listing Only Select Queries

```
Sub SelectQueries()
   Dim dbsCurrent  As DATABASE, qdfSelect As QueryDef
   Set dbsCurrent = CurrentDb()

   For Each qdfSelect In dbsCurrent.QueryDefs
      If qdfSelect.Type = dbQSelect Then
         Debug.Print qdfSelect.Name
      End If
   Next

End Sub
```

Other query types include action queries, which modify data; data-definition queries, which can create tables and indexes; pass-through queries, which allow you to send SQL statements directly to the server; union queries, which combine two or more tables; and crosstab queries, which display data in a spreadsheet-like format. Chapter 8 discusses how to create different types of queries.

Creating a Database with DAO

To help you understand the different objects that make up your database, you're going to learn how to create a copy of your database by using nothing but DAO. You'll copy tables, queries, and relationships property by property. You'll also copy each record in every table field by field.

NOTE

This example doesn't provide the most efficient method for copying a database. It does, however, demonstrate how to access and use each object in DAO.

While you move through the examples, you're told where you can go in the Access user interface to view how Access displays the information available in DAO. This should help give you a clearer picture of how DAO is organized.

You can find the WebC.mdb database on this Web site. This database consists of a single form and a single code module. All the code for managing the copying of the database is contained in the code module `modCopyDatabase`. In this code module, you find the subroutines listed in Table C.1. (These subroutines are discussed in more detail in the following sections.)

TABLE C.1 Subroutines Used to Copy the Database

Subroutine	*Description*
`CopyDatabase`	Manages the copying of the database. It opens the source database, creates the destination database, and calls all support routines to perform the copy.
`CopyData`	Copies all records in each table field by field. No attached table's data is copied. To help increase performance, data is added by using a transaction.
`CopyFields`	Copies the Fields collection for the table, indexes, and relationship objects.
`CopyIndexes`	Copies the indexes for each table.
`CopyProperties`	Copies all the properties for the table, index, field, and query objects.
`CopyQueries`	Copies each query.
`CopyRelations`	Copies each relationship.
`CopyTables`	Copies all the tables to the new database, except system tables.

The code in WebC.mdb doesn't copy forms, code modules, or reports. Although DAO provides a method for enumerating these pieces as document objects, there's no mechanism through DAO to create new document objects. Nor can you copy individual documents through DAO to another database. But you can copy these objects from the currently active database to another database by using Access's `CopyObject` or `TransferDatabase` method.

The Database Copy Utility form, `frmDatabaseCopy`, is a simple dialog that lets you select a source and destination database to copy (see Figure C.2). You can also specify on this form whether to copy just the structures of the tables or both the structure and the data. To get the filenames, this form uses the API call to the File Open common dialog for requesting the file to copy and the new file to copy to. More information about using API calls can be found in Chapter 15, "Extending the Power of Access with API Calls."

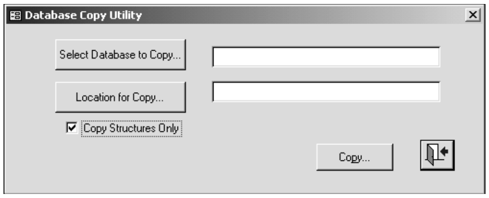


FIGURE C.2
Use the Database Copy Utility form to make a copy of your database.

Demonstrating that DAO is a true component, all code in the code module can also be written and run in any Microsoft application that supports VBA and DAO.

Creating the Database Object

Listing C.4 shows the `CopyDatabase` subroutine. This subroutine takes a path to the source database, the destination database, and a Boolean value that indicates whether only the structures should be copied, or the structures and the data should be copied.

LISTING C.4 WebC.mdb: Copying a Database with DAO

```
Sub CopyDatabase(strSourceFile As String, strDestFile As String, _
  blnCopyData As Boolean)
  ' Given the path to the source database, create a duplicate copy of the
  ' database using DAO. The blnCopyData parameter specifies whether
  ' to copy just the structure, or the structure and the data.
  ' While this is not the fastest method for copying databases, it
  ' demonstrates how to create and manipulate all the available objects
  ' in your database.
  Dim dbsSrc As Database, dbsDest As Database

  ' Create the database
  On Error GoTo errExists
  Set dbsSrc = DBEngine.Workspaces(0).OpenDatabase(strSourceFile)
  Set dbsDest = DBEngine.Workspaces(0).CreateDatabase(strDestFile, _
    dbLangGeneral)
  On Error GoTo 0
```

LISTING C.4 Continued

```
  CopyTables dbsSrc, dbsDest
  CopyQueries dbsSrc, dbsDest
  ' Copying of the data occurs before copying the relationships. This
  ' is so you do not have to worry about whether the order the data
  ' is copied in violated referential integrity rules.
  If blnCopyData Then
    CopyData dbsSrc, dbsDest
  End If
  CopyRelationships dbsSrc, dbsDest
  dbsDest.Close
  dbsSrc.Close
  Exit Sub

errExists:
  If Err = 3204 Then
    MsgBox "Cannot copy to a database that already exists!"
  Else
    MsgBox "Error: " & Error$
  End If
  Exit Sub

End Sub
```

Creating or opening databases with DAO is very easy with the `CreateDatabase` and `OpenDatabase` methods. Both methods must be called on a `Workspace` object.

Opening Existing Databases

The `OpenDatabase` method lets you specify how to open the database. In Listing C.4, just the database name is supplied. You can also optionally supply information to open the database, as shown in the following table. The syntax for the `OpenDatabase` method is as follows:

```
Set database = workspace.OpenDatabase(dbname, exclusive, read-only, source)
```

Argument	*Description*
`exclusive`	A Boolean value that specifies whether the database is to be opened exclusively or shared. Databases opened exclusively can be opened by only a single user at a time. When omitted, the database is opened shared.
`read-only`	A Boolean value that specifies whether the database is to be opened as read-only. When omitted, the database is opened as read/write.
`source`	A string expression that supplies the database's password and connect information for connecting to ODBC data sources.

Creating New Databases

Creating a database requires a path and filename to store the new database and the database's locality. The *locality* defines how your database sorts data and locates matches. Most often, the locality you use is `dbLangGeneral`, which specifies the ordering used by English, German, French, Portuguese, Italian, and Spanish. If your database is being used in other locales, you can open the database so that it sorts for their locality. For example, in Russia you might want sorting to occur differently.

You also can choose to have the newly created database encrypted. Omitting the *`options`* argument creates an unencrypted database. The syntax for the `CreateDatabase` method is

```
Set database = workspace.CreateDatabase (databasename, locale, options)
```

Compacting Existing Databases

To create an encrypted database or to switch the locale of the database, you need to create a new copy from the existing database. Although you can easily modify the sample code in Listing C.4 to do this, Access provides the `CompactDatabase` method for doing this quickly.

The compact database is a method of the `DBEngine` object. The syntax for `CompactDatabase` is

```
DBEngine.CompactDatabase olddb, newdb, Locale, options, Password
```

At a minimum, you must supply the existing database name (*`olddb`*) and a new database name (*`newdb`*). The new database name must be different from the existing one. By using the *`Dstlocale`*, *`SrcLocale`*, *`options`*, and *`Password`* parameters, you can specify new localities, encryption, and password.

This method doesn't return the newly created database. To use the new database after calling the `CompactDatabase` method, you must open the database with the `OpenDatabase` method.

NOTE

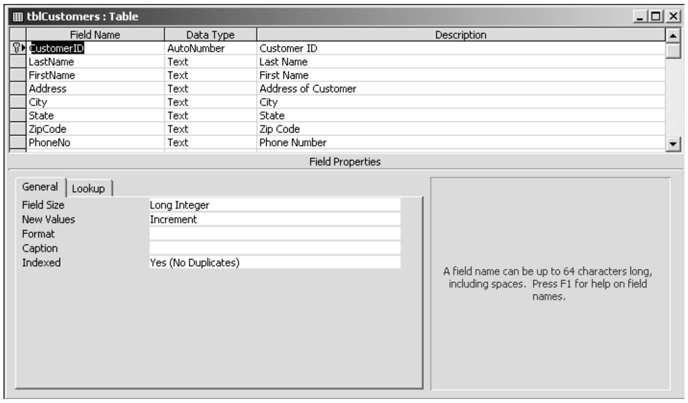
To encrypt a database from within Access, first close all open databases. Then from the Tools menu, choose Database Utilities to compact, repair, and convert databases from previous Access versions, or choose Security to encrypt and set up user accounts. For more information about encrypting a database, see Chapter 20, "Securing Your Application." You can also convert to previous Access versions (namely, 97/2000) in Access 2002, and can compact the open database in place.

The `modGlobalUtilities` module of the book's sample application (VideoApp.mdb) provides code to demonstrate how you can request your users in a multiuser system to log off, as well as automatically try to compact the back-end database. This database can be found on this Web site.

Copying Table Structures

In the DAO object hierarchy, notice that tables consist of fields, indexes, and properties. Thus, with each table, the fields, indexes, and properties collections must also be copied. The `TableDef` object represents all the characteristics available when designing a table in the table designer.

The window in Figure C.3 shows the derivation of most information in the `TableDef` object. (I say "most" because the descriptions aren't copied.) The Fields collection corresponds to the field list in the table. Properties about each field are displayed at the bottom of the window, on the General page. To view most of the properties of a `TableDef` object, choose Properties from the View menu.



Field Name	Data Type	Description
CustomerID	AutoNumber	Customer ID
LastName	Text	Last Name
FirstName	Text	First Name
Address	Text	Address of Customer
City	Text	City
State	Text	State
ZipCode	Text	Zip Code
PhoneNo	Text	Phone Number

Field Size	Long Integer
New Values	Increment
Format	
Caption	
Indexed	Yes (No Duplicates)

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

FIGURE C.3
Creating and editing tables in Access actually sets the properties of the `TableDef` object.

To view the Indexes collection of the `TableDef` object, from the View menu choose Indexes. Notice that each index can have a list of fields, which are contained in a Fields collection of the `Index` object.

Listing C.5 is a good example of when to use code to ignore system tables. You don't want to copy the system tables because they are created and managed automatically by the database engine as you copy information. Actually, trying to create a table with the name of an existing system table generates errors.

LISTING C.5 WebC.mdb: Copying the `TableDefs` Collection

```
Sub CopyTables(dbsSrc As Database, dbsDest As Database)
  Dim tbfSrc As TableDef, tbfDest As TableDef
  For Each tbfSrc In dbsSrc.TableDefs
```

LISTING C.5 Continued

```
    If (tbfSrc.Attributes And dbSystemObject) Then
    Else
      Set tbfDest = dbsDest.CreateTableDef(tbfSrc.Name, _
        tbfSrc.Attributes, tbfSrc.SourceTableName, tbfSrc.Connect)
      If tbfSrc.Connect = "" then
         CopyFields tbfSrc, tbfDest
         CopyIndexes tbfSrc.Indexes, tbfDest
      End If
      CopyProperties tbfSrc, tbfDest
      dbsDest.TableDefs.Append tbfDest
    End If
  Next
End Sub

Sub CopyIndexes(idxsSrc As Indexes, objDest As Object)
  Dim idxSrc As Index, idxDest As Index, propSrc As Property
  For Each idxSrc In idxsSrc
    Set idxDest = objDest.CreateIndex(idxSrc.Name)
    CopyProperties idxSrc, idxDest
    CopyFields idxSrc, idxDest
    objDest.Indexes.Append idxDest
  Next

End Sub
```

In Listing C.5, the `objDest` argument is being passed as a generic object so that the `CopyIndexes` subroutine can be reused to copy indexes of the `TableDef` and `Relations` objects.

The generic `CopyProperties` subroutine in Listing C.6 copies all the object's properties. DAO exposes a Properties collection on almost all its objects. To cover cases where a property might not have a value, a simple error handler is included in the code.

LISTING C.6 WebC.mdb: Copying All Properties for the Object

```
Sub CopyProperties(objSrc As Object, objDest As Object)
  Dim prpProp As Property, temp As Variant
  On Error GoTo errCopyProperties
  For Each prpProp In objSrc.Properties
    objDest.Properties(prpProp.Name) = prpProp.Value
  Next
  On Error GoTo 0
  Exit Sub
```

LISTING C.6 Continued

```
errCopyProperties:
  Resume Next
End Sub
```

Notice that creating a `TableDef` object in code consists of the following steps:

1. The `CopyTables` subroutine creates a blank `TableDef` object by declaring a new instance of the object for each table in the source database. Creating a `TableDef` object doesn't automatically append it to the database's `TableDefs` collection. Until the `TableDef` object is appended to the database, you can modify most of the properties. When appended, some properties, such as the `TableDef` object's `Attributes` property, become read-only.
2. By using the `CopyFields` subroutine (this subroutine's code appears later in the section "Fielding the `Field` Object"), the fields are copied from the source `TableDef` and appended to the newly created `TableDef`. (The `Field` object works similarly to the `TableDef` object.)
3. After the fields are appended, indexes based on those fields can be created. Indexes are composed of their own sets of fields. The fields in the index must be existing fields from the table. For this reason, the fields are appended to the `TableDef` before the indexes are. Again, the index must be created and then appended to the `TableDef`. For most objects in DAO, you need to create the object and then append it to the appropriate collection. However, this isn't always true, as you'll learn in the next section.
4. After the table is completely defined, append it to the `TableDefs` collection to save the newly created table with your database.

These steps explain the simple process of creating a `TableDef`. However, they don't point out the details of the `TableDef` object. When creating a `TableDef`, you need to answer the following questions:

- What do you want to name the table?
- Is the table an attached table? If so, what are the attributes relating to this attachment, what's the connect string, and what's the source table's name?

You can answer these questions immediately when you create the `TableDef` object. By examining the following code, you can see that four properties are being set:

```
Set tbfDest = dbsDest.CreateTableDef(tbfSrc.Name, tbfSrc.Attributes, _
      tbfSrc.SourceTableName, tbfSrc.Connect)
```

The four properties being set are as follows:

- Name is the name of the table.
- Attributes identifies the various attributes that can be set for a table. For an examination of the Attributes property, see the earlier section "Examining Table Attributes."
- SourceTableName is used for linked tables.
- Connect is used for ODBC tables, giving the connect string.

You can set these four properties when creating the table or at any time up to appending the table to the TableDefs collection. After a table is appended, you can still change the name and connect string, but you can't modify the attributes or the source table name.

Fielding the Field Object

The most popular of all DAO objects is Field. It exists in TableDefs, indexes, and relations. The code in Listing C.7 is the CopyFields subroutine called by the CopyTableDefs subroutine. The CopyFields subroutine copies fields from any source object with a Fields collection to any destination object.

LISTING C.7 WebC.mdb: Copying the Fields Collection

```
Sub CopyFields(objSrc As Object, objDest As Object)
  Dim fldSrc As Field, fldDest As Field
  For Each fldSrc In objSrc.Fields
    If TypeName(objDest) = "TableDef" Then
      Set fldDest = objDest.CreateField(fldSrc.Name, _
        fldSrc.Type, fldSrc.Size)
    Else
      Set fldDest = objDest.CreateField(fldSrc.Name)
    End If
    CopyProperties fldSrc, fldDest
    objDest.Fields.Append fldDest
    Next
    Exit Sub

End Sub
```

In the arguments for the CopyFields subroutine, notice that the source and destination tables are passed as objects rather than as TableDefs. This is a good example of where you can use a generic object type to write reusable code. The same CopyFields subroutine copies fields between indexes and relations.

Also, how the field is created depends in the object. Based on the object, different properties are available on the field. For example, a field in a table must have a specific size and type. In a relationship and index, however, all that's necessary is the field name because on those objects the field name refers back to the `TableDef`.

NOTE

One disadvantage to using the `CopyFields` routine and generic object types is that VBA fails to do any type checking on the calling parameters. Any pair of objects can be passed into this routine without causing a compile-time error. The only indication of improperly calling this routine would be a runtime error. You can use `TypeOf()` to determine what type of field you are dealing with, thus making the code more robust in critical places.

Copying Queries

The `QueryDef` object stores all your query definitions. This object exposes SQL queries in an object-oriented approach. You can access the list of fields that make up a query, as well as the parameters in parameterized queries. These collections are created from the SQL statement. Therefore, when you create a query through DAO, you supply only the appropriate SQL statement; DAO does the rest. You can't append to the parameters or the Fields collection on a `QueryDef` object.

Creating queries is the exception to the pattern you've been seeing. Creating a `QueryDef` automatically appends the query to the database. You don't call the `Append` method on the `QueryDef` object.

The `CopyQueries` subroutine (see Listing C.8) shows how to copy a query from a source database to a destination database. Different from all the other copy subroutines provided earlier, the query is created directly in the database and isn't appended to the database.

LISTING C.8 WebC.mdb: Copying the `QueryDefs` Collection

```
Sub CopyQueries(dbSrc As Database, dbDest As Database)
  ' Querydefs are automatically appended to the database at the time
  ' of creation.
  Dim qrySrc As QueryDef, qryDest As QueryDef
  For Each qrySrc In dbSrc.QueryDefs
    Set qryDest = dbDest.CreateQueryDef(qrySrc.Name, qrySrc.SQL)
    CopyProperties qrySrc, qryDest
  Next
End Sub
```

Creating Temporary Queries

Use the `CreateQueryDef` method to create temporary queries. To create a temporary query, call the `CreateQueryDef` method but supply an empty string as the query's name.

If, after creating this query, you want to append it to the database, you must provide the query with a name and then append the query to the `QueryDefs` collection. Trying to append a query that doesn't have a name generates a runtime error.

TIP

Because creating temporary queries tends to "bloat" the database, you want to take advantage of a feature that allows you to compact the current database when exiting the application. To set this feature, choose Options from the Tools menu. On the General page, check the Compact on Close option. (The compacting will occur only if the database size will be reduced by 256KB.)

Compiling Queries

Queries executed and opened from `QueryDef` objects run more quickly than executing SQL statements. For example, to create a query that returns all the customers in a specific state, write the following function that creates the SQL statement and executes it:

```
Function StateQuery(strState as String) as Recordset
  Dim dbsCurrent as Database
  Set dbsCurrent = Currentdb()
  ' Chr(34) is used to insert a quotation mark.
  Set StateQuery = dbsCurrent.OpenRecordset("Select * From [Customers]" _
         & "Where [State] = " & chr(34) & strState & chr(34))
End Function
```

A more efficient way to execute this query is to create it as a parameterized query through the query designer and save the query. The database engine can then pre-optimize saved queries. You can view the query for this example in the WebC.mdb sample application, which you can find on this Web site. The query name is `qryCustomersInState`. Rewriting the preceding `StateQuery()` function to use the compiled query results in the following:

```
Function StateQuery(strState as String) as Recordset
  Dim qdState as QueryDef
  Dim dbsCurrent as Database
  Set dbsCurrent = Currentdb()
  Set qdState = dbsCurrent.QueryDefs("qryCustomersInState")
  qdState.Parameters("WhatState") = strState
  Set StateQuery = qdState.OpenRecordset
End Function
```

In the function, the `QueryDef` for the query is referenced. When you execute parameterized queries, you must first set the parameters through the `QueryDef` object. The parameters are stored in a Parameters collection.

NOTE

A *parameterized query* requires arguments to be run. If the arguments aren't supplied, you receive an error when the query is executed.

After you set the parameters, call the `OpenRecordset` method on the `QueryDef` object to return the recordset for the currently specified parameters. To execute queries that don't have a Parameters collection, just open the recordset from the database object. For example, to open a query called `qryMoviesbyCategory`, use the following code:

```
Function MoviesByCategory() as Recordset
  Dim dbsCurrent as Database
  Set dbsCurrent= Currentdb()
  Set MoviesByCategory = dbsCurrent.OpenRecordset("qryMoviesbyCategory")
End Function
```

You don't need to go through the `QueryDefs` collection when running queries that aren't parameterized. It's much simpler just to use the `OpenRecordset` method on the current database object.

Copying Relationships

Relationships can exist between any two tables or queries in your database. A relation consists of a primary key and a foreign key. The *primary key* is one or more fields in a table that uniquely identify a record; the *foreign key* is one or more fields from another table that refer to the primary key.

For example, in a customer order-entry system, you can have one table containing all your customers and another table containing all the customer orders. In the Customers table, the CustomerID field uniquely identifies a customer. Therefore, in designing the tables, the Orders table would also have a CustomerID field. However, quite a few records for each customer are probably in the Orders table because one record represents one order. In defining this relationship, the primary key is the CustomerID in the Customers table, and the foreign key is the CustomerID in the Orders table. Because each customer can have many orders, this type of relationship is called *one-to-many*.

All these characteristics are available on the `Relation` object. The `Attributes` property represents the joining relationship between the two tables or queries. The `Attributes` property is used to distinguish all the information about the relationship available in the Define Relationship dialog box. To display the relationships available on the current database, from the Tools menu choose Relationships. This view corresponds to the Relationships collection in DAO. Figure C.4 shows an example of this.

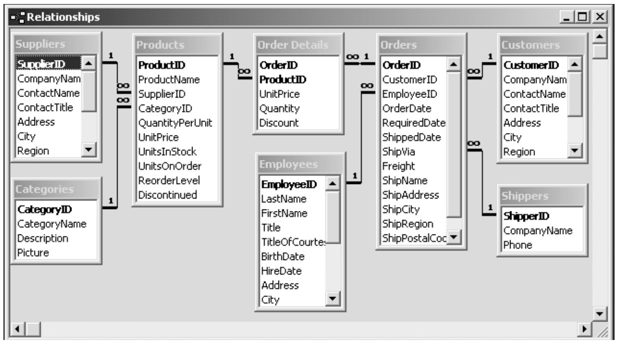
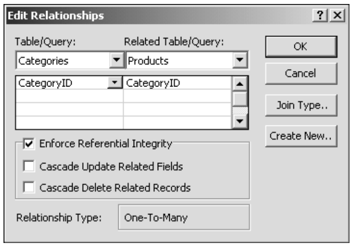


FIGURE C.4
This window shows the relationship layout for the sample application Northwind.mdb, which comes with Access.

To view a specific relationship object, double-click any lines connecting two tables. This brings up a dialog similar to the one in Figure C.5.



Table/Query: Categories
Related Table/Query: Products

Table/Query	Related Table/Query
CategoryID	CategoryID

[✓] Enforce Referential Integrity
[ ] Cascade Update Related Fields
[ ] Cascade Delete Related Records

Relationship Type: One-To-Many

FIGURE C.5
In this dialog, you can define the type of relationship and whether referential integrity should be enforced.

Defining a relationship occurs through the window in Figure C.4 and the dialog in Figure C.5. The Table/Query column in the Edit Relationships dialog represents the primary key in the relationship, whereas the Related Table/Query column represents the foreign key. The fields selected in these lists are represented by the Fields collection of the relationship object. In this collection, you can get each field that's taking part in the relationship, the primary key, and the foreign key.

The referential integrity and the join information are available on each relation object. Referential integrity ensures that every foreign key has a primary key. Listing C.9 shows how to copy the relationships, adding the C to the name of the relationship copied just so that you know it's a copy.

LISTING C.9 WebC.mdb: Copying the Relations Collection

```
Sub CopyRelationships(dbsSrc As Database, dbsDest As Database)
  ' This routine copies all the relationships to a new database.
  ' There is no properties collection on a relation, and all the
  ' properties can be set at the time of creating the new relation.
  Dim relSrc As Relation, relDest As Relation
  For Each relSrc In dbsSrc.Relations
    If Left(relSrc.Name, 4) <> "MSys" Then
      Set relDest = dbsDest.CreateRelation("C" & relSrc.Name, _
        relSrc.Table, relSrc.ForeignTable, relSrc.Attributes)
          CopyFields relSrc, relDest
          dbsDest.Relations.Append relDest
    End If
  Next
End Sub
```

Copying Data

Up to this point, you've created a copy of your database's table, query, and relationship definitions. Now, your new database consists of no data. In this section, you perform the data transfer by using just DAO calls. Opening tables in DAO is done by using the `OpenRecordset` method on the database object (see Listing C.10).

LISTING C.10 WebC.mdb: Copying Data by Using Workspaces

```
Sub CopyData(dbsSrc As DATABASE, dbsDest As DATABASE)
  Dim tbfSrc As TableDef, rstDest As Recordset, rstSrc As Recordset
  Dim wspTransact As Workspace
  Dim fldSrc As Field
  Set wspTransact = DBEngine.Workspaces(0)
  wspTransact.BeginTrans
  On Error GoTo errRollback
  For Each tbfSrc In dbsSrc.TableDefs
    If (tbfSrc.Attributes And dbSystemObject) Or _
      (tbfSrc.Connect <> "") Then
         ' No system tables or attached tables
      Else
        Set rstSrc = dbsSrc.OpenRecordset(tbfSrc.Name, dbOpenTable, _
          dbForwardOnly)
        If Not rstSrc.EOF Then   ' Make sure it is not empty
          Set rstDest = dbsDest.OpenRecordset(tbfSrc.Name, _
```

LISTING C.10 Continued

```
            dbOpenDynaset, dbAppendOnly)
          Do While Not rstSrc.EOF
            rstDest.AddNew
            For Each fldSrc In rstSrc.Fields
              rstDest(fldSrc.Name) = fldSrc.Value
            Next
              rstDest.UPDATE
              rstSrc.MoveNext
          Loop
            rstDest.Close
        End If
        rstSrc.Close
    End If
  Next
  wspTransact.CommitTrans
  Exit Sub
errRollback:
  MsgBox "Error:" & Error$
  wspTransact.Rollback
  Exit Sub
End Sub
```

When the recordsets are opened, extra parameters are specified. The source table is specified to be opened as a read-only table. These customizations give you more flexibility and can be used to increase performance. Opening for read-only increases performance because the database engine knows you'll never be making any modifications.

TIP

If the tables you are copying over aren't replicable, you can also use an `INSERT` SQL statement with `dbs.Execute` to copy a whole table. This might work faster than the record-by-record, field-by-field copy method shown here. The purpose for using the method displayed here is to show how to use DAO.

You can use many other options to open recordsets. You can specify in a multiuser environment that no one else can modify or add records (`dbDenyWrite`), that other users can't even view the records (`dbReadOnly`), and whether in a multiuser environment you get a runtime error if you try to edit data that another user is editing (`dbSeeChanges`). Many of these options can be combined with the `And` operator to give you further control. To make a recordset that denies reads and writes to other users, specify the options as `dbDenyWrite And dbDenyRead`.

Increasing Speed with Transactions

In the `CopyData` subroutine in Listing C.10, notice the `wspTransact.BeginTrans` line. Access supports transactions, which, in some cases, greatly increase the speed of modifying and updating data. By using a transaction, you can group a large number of updates into a single operation.

However, transactions are more than just faster updates. By using transactions, you can modify one or many tables. At any point during the update, you can decide to cancel, or *roll back*, the transaction. Rolling back a transaction causes no updates to be saved to your database.

NOTE

If you plan to use transactions for performance reasons, be sure to run benchmarks with and without the transaction commands. In prior versions of Jet and Access, transactions could be counted on for increasing speed; in Access 97 and Jet 3.5, this wasn't always the case. As of Access 97 and Jet 4, the main purpose for using transactions should be to take advantage of the rollback capability, if necessary.

Using transactions can be a very efficient and effective way to perform bulk operations. If, during the bulk update, an error or unexpected condition occurs, you can roll back the transaction, leaving your database in its original state.

CAUTION

Although transactions might sound wonderful, you must be aware of many complexities. For every `BeginTrans` you call, you must always have a `CommitTrans` or `Rollback`. Leaving transactions open can cause unpredictable and possibly dangerous effects because you'll leave recordsets open with locks on them. This is especially important to recognize when debugging an application. If you stop your program in the middle of a transaction, don't reset your application without committing or rolling back the transaction. For example, in the `CopyData` subroutine in Listing 3.17 (in Chapter 3), if you stopped the procedure with Ctrl+Break with the intention of aborting the procedure, you must execute the following code line in the Immediate window:

```
wspTransact.Rollback
```

Omitting the rollback leaves your application in a potentially dangerous state, with recordsets left locked.

Transactions encompass any and all modifications and updates to any database that occur within the `Workspace` object. Therefore, you can create transactions that span multiple tables and even databases. This is important to understand, especially when you roll back, because you'll be rolling back all the actions that have occurred since the beginning of the transaction.

Transactions can be nested. This means that you can create a transaction and within that transaction create another transaction. You can roll back the inside transaction and continue to process the outside transaction. You can also commit the inside transaction and then roll back the entire operation by rolling back the outside transaction.

Sometimes you might actually want to run two transactions independently and simultaneously. You can easily do so by creating a clone of your `Workspace`. On the `Application` object of Access, call the `DefaultWorkspaceClone` method to obtain a second `Workspace` on your database. In this `Workspace`, you can initiate a second, independent transaction that can run at the same time as the transaction on the default `Workspace`.

NOTE

The `DefaultWorkspaceClone` method doesn't require the user to log back on. The cloned workspace has the same characteristics as the original workspace. It's equivalent to the user logging on a second time with the same password.

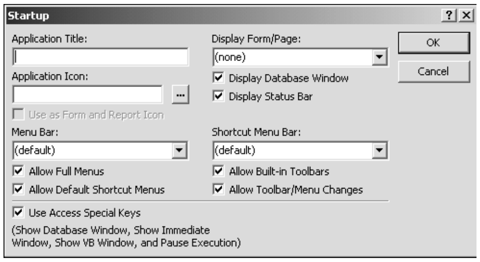
Using Custom Properties

Access uses DAO to track different pieces of information about your application. For example, the application title, application icon, summary information about the database, and much more information is stored in DAO.

To view your application title, icon, and other information about the application, choose Startup from the Tools menu. The dialog shown in Figure C.6 appears.

Access stores the information in this dialog in the database by using custom properties. However, unless you've set the property through the user interface, the property might not yet be added to the database. Therefore, whenever you write code that sets or retrieves a custom property, you must first check to see whether the property exists.

Listing C.11 shows the sample code that attempts to set a property. This code takes the object the property resides on, the property name, and the value to assign to the property. If the property is successfully set, `True` is returned; otherwise, `False` is returned.



Application Title:

Display Form/Page: (none)

Application Icon:

☐ Use as Form and Report Icon

✓ Display Database Window

✓ Display Status Bar

Menu Bar: (default)

Shortcut Menu Bar: (default)

✓ Allow Full Menus

✓ Allow Built-in Toolbars

✓ Allow Default Shortcut Menus

✓ Allow Toolbar/Menu Changes

✓ Use Access Special Keys

(Show Database Window, Show Immediate Window, Show VB Window, and Pause Execution)

FIGURE C.6
The Startup dialog lets you define the startup options for your application, such as the initial form to open and what menu bars to display.

NOTE

Sometimes it might be better for the application to store custom properties in the `CurrentProject` object or its collections, especially if the database might be upsized. To learn more about `CurrentProject`, see Chapter 4.

LISTING C.11 WebC.mdb: Assigning Values by Using the Properties Collection

```
Function AssignProperty(objSource As Object, prpName As String, _
    prpType As Variant, prpValue As Variant) As Boolean
  Dim prp As Property
  On Error GoTo Assign_Err
  objSource.Properties(prpName) = prpValue
  AssignProperty = True
AddProp_Exit:
  Exit Function
Assign_Err:
  AssignProperty = False
End Function
```

You can use the code in Listing C.12 to add a property to an object in DAO. The function takes the DAO object to add the property to, the name of the property, the data type, and an optional value to assign to the property.

LISTING C.12 WebC.mdb: Adding a Custom Property to a DAO Object

```
Function AddProperty(objSource As Object, strName As String, _
    varType As Variant, Optional varValue As Variant) As Boolean
  Dim prpProp As Property
  On Error GoTo Err_AddProp
```

Listing C.12 Continued

```
  If Not IsMissing(varValue) Then
    Set prpProp = objSource.CreateProperty(strName, varType, varValue)
  Else
    Set prpProp = objSource.CreateProperty(strName, varType)
  End If
  objSource.Properties.Append prpProp
  AddProperty = True
Exit_AddProp:
  Exit Function
Err_AddProp:
  AddProperty = False
  Resume Exit_AddProp
End Function
```

Not all objects in DAO support defining custom properties. The following objects in DAO support user-defined properties:

- `Database`
- `Index`
- `QueryDef`
- `Field` (in `TableDef` and `QueryDef`)
- `Document`

In the database container's Documents collection, *document* is a `UserDefined Document` object. The `UserDefined Document` is an object in the Documents collection where you can store application attributes. For example, in VideoApp.mdb, the `UserDefined` document is used to store the name of the back-end database. To access the number of properties on the `UserDefined` object, in the Immediate window, you can write the following line of code:

```
? CurrentDb().Containers!Databases.Documents
[ic:ccc]("UserDefined").Properties.Count
```

Any major item of information that you want to store for the database itself can be stored in the `UserDefined` document.

Note

Remember that if you are just beginning to work with Access and need to manipulate data with code, you'll be better off using ADO in Access 2002. This is covered in the book in Chapter 5, "Introducing ActiveX Data Objects."