

Chapter 1

Designing Your Tables

In this chapter:

Avoiding Common Problems.	4
Using Good Design Techniques.	23
The Bottom Line: Know Thy Data and Its Usage.	35

As discussed in the Introduction, this book assumes that you understand how to design tables for a relational database like Microsoft Access. In truth, a single chapter cannot teach you all you need to know about table design. See Appendix A for some recommendations of books you should consider reading to improve your design skills.

However, no book on building database applications would be complete without some discussion of the most common errors, why these errors are bad, and what you can do to fix them. This chapter also discusses design techniques unique to Access that you can employ to make your application run more smoothly.

You can find many of the recommendations in this chapter scattered throughout several chapters in either my *Running Microsoft Access 2000* (Microsoft Press, 1999) or *Microsoft Office Access 2003 Inside Out* (Microsoft Press, 2003) book. However, those books focus on the mechanics of creating tables and relationships, and offer design recommendations primarily in sidebars or notes. This chapter assumes that you already understand the mechanics of table and relationship design in Microsoft Access, and instead provides a discussion of specific design techniques you can use to improve your table structures. In fact, you can apply many of the lessons learned in this chapter to designing any relational database structure.



Note Before writing this chapter, I polled my fellow Microsoft Access MVPs for suggestions about topics that I should include. (You can find a list of all who participated in the Acknowledgments section at the beginning of this book.) Although we could all agree on certain fundamental principles, we had some rather interesting (and sometimes heated) discussions about some of the design recommendations that I proposed. You can find the topics upon which we all reached consensus in the next section, “Avoiding Common Problems.” “Using Good Design Techniques,” page 23, includes recommendations that you should consider when designing your tables, but you should apply only those that make sense for your specific application.

4 Part I: Designing Your Application

Avoiding Common Problems

Even experienced Access developers occasionally commit design errors. If you have been developing Access applications for a while, you have probably already learned how expensive a small design error can be after you have designed dozens of forms and reports and need to implement a new feature. If your tables were well designed, adding a new feature is easy. If you took any shortcuts, making changes is costly and time consuming. This first section describes the 10 most common problems and how to avoid them.

Problem #1: Creating Compound Fields

Especially if you have imported data from another source, you might find that some fields contain data that should be broken into multiple fields. A very common problem is using a compound field for an address that contains house or building number, street name, apartment or suite number, city, region or state, postal code, and country—perhaps formatted with carriage returns to print as a single value on mailing labels or envelopes. Another common error is storing a person's full name in a single field.

This is great for a word processing application, but it's a bad idea for a database. If the postal code is embedded within a long string, how will you sort the records by postal code? How can you find records for a specific city? How can you sort on a person's last name or search on last name if it's embedded within a long string? Sure, you can use a LIKE search, but that is very inefficient. And you certainly cannot build an index on any of the parts to improve performance.

Figure 1-1 shows you a typical record layout with compound fields and suggests a more appropriate layout for the same data in a database table. If you study the relational model for database design, this process of separating compound fields into simple ones is called decomposing data into *atomic* values.

The better design allows you to search and index on discrete values, and you can always re-create the more complex values using string concatenation. Of course, your final design depends on how your application needs to use the data. Some applications might require even further granularity—such as splitting out the house or building number into a separate field or adding a person's title.

Bad:

CustName	CustAddress
John Allen Smith, Jr.	1234 Main Street, Suite 5A, Oakland, CA, 94622, USA

Better:

CustLName	CustFName	CustName	CustSuffix	CustAddress1
Smith	John	Allen	Jr.	1234 Main Street

CustAddress2	CustCity	CustRegion	CustPostal	CustCountry
Suite 5A	Oakland	CA	94622	USA

Figure 1-1 Decomposing complex fields into atomic values.

Problem #2: Including Repeating Groups

A *repeating group* is a set of one or more columns that appears multiple times in a table definition. That is, the table design includes multiple columns that all have the same kind of data. Figure 1-2 shows a simple repeating group of classes in which each particular student is enrolled.

StudentID	Period1	Period2	Period3	Period4	Period5	Period6
55	Algebra	History	French	Physics	Phys Ed	Comp Sci
68	Geometry	French	Chemistry	Soccer	Orchestra	Debate
87	Geography	Math I	German	Biology	Speech	Band

Figure 1-2 A simple repeating group of classes for which a student is registered.

Repeating groups *can* be useful for displaying or analyzing data. In fact, you'll often see repeating groups in a spreadsheet or in a PivotTable. However, they're terrible for storing data in a relational database. For example, how would you find out how many students are registered for each subject? If you know the subject names in advance, you can certainly search for rows that have a particular subject in any of the class periods and then count the rows. But you would have to run a separate query for each known subject and then UNION the results to get the complete answer. Also, if the school decides to add a seventh class period, the table design must be changed. And if some students attend only part time, the table will contain rows with empty and unused columns, thereby wasting space in the table.

To correctly store data like this in a relational table, you need to “un-pivot” the grid and use the repeating group column names as data values. Figure 1-3, on the next page, shows the data in Figure 1-2 restructured for a relational table.

6 Part I: Designing Your Application

StudentID	Period	Subject
55	1	Algebra
55	2	History
55	3	French
55	4	Physics
55	5	Phys Ed
55	6	Comp Sci
68	1	Geometry
68	2	French
68	3	Chemistry
68	4	Soccer
68	5	Orchestra
68	6	Debate
87	1	Geography

Figure 1-3 The correct design for a class registration table that avoids repeating groups.

When you restructure the table in this way, finding out how many students are registered for each subject is easy.

```
SELECT Subject, Count(StudentID) As SubjectCount
FROM StudentRegistrations
GROUP BY Subject;
```

If the school decides to add an early “zero” class period or a late seventh class period, the table structure does not need to be changed because the new period numbers are simply new values in the Period column.



Tip You can often identify repeating groups by examining the field names. You probably have a repeating group if any of the following is true:

- Field names are similar and differ only by the presence of a numeric or alphabetic suffix or prefix.
- Field names are lists of the names of months in a year, days in a week, or hours in a day.
- Field names are descriptions of the same type of object. For example, apples, pears, bananas, and peaches are all fruits. Mathematics, history, biology, and rhetoric are all course names.

Note that a repeating group isn't always just one field repeated several times. Repeating groups can also be sets of two or more fields repeated in a table design. Extending the example shown in Figure 1-2, a table containing StudentID, Period1Class, Period1Instructor, Period1Classroom, Period2Class, Period2Instructor, Period2Classroom, and so on contains sets of three fields repeated several times in the table design. This table structure not only contains compound repeating groups but also includes the problem of including fields for multiple subjects in a single table, as discussed in the next section.

Problem #3: Putting Multiple Subjects in One Table

Each table should describe one and only one subject or action. An example of a common error is including the full name and address information for the customer of each order in the Orders table rather than placing the customer information (a separate subject) in its own table. When you fail to create a separate table, you repeat data unnecessarily, potentially introduce errors, and must potentially update many rows in the Orders table if you want to correct the spelling of a customer name or update a billing address.

To consider another example, assume that the table shown previously in Figure 1-2 not only has the repeating group for classes (Period1Class, Period2Class, and so forth) but also has additional fields within each group: Period1Instructor, Period1Classroom, Period2Instructor, Period2Classroom, and so on. If Period1Instructor contains a person's name and Period1Classroom contains a classroom name and location, these are quite probably separate subjects that belong in their own tables. You should have a separate table for Instructors and include only the primary key value from a specific row in that table as the foreign key in this table to link the class registration to the details about the instructor. Likewise, Classrooms (and perhaps Buildings) should also be separate subject tables, and the specific class location in the class registration row should be a foreign key link to those tables.

You can perform a simple test to find multiple subjects in one table: Can I change the value of any field (other than the primary key) in this table and not have to change a similar value anywhere else in my database? Note, however, that it is acceptable to copy certain values to dependent tables depending on your business rules. For example, an Order Detail record might include a *copy* of the Price from the Products table because that copy represents the price at the time the order was placed. Or, an Order might need a Shipping Address that could be different from the customer's address at the time the order is placed.

I call this type of data *Point in Time* data—data that was current at a specific time. The copy of the Price in the Order Detail record is the price that was *current at the time the order was placed*, but the Price in the Products table is always the current price. The price in the Order Detail table is not a duplicate field because its meaning is different from the price in the Products table. Likewise, the Shipping Address is the location to which the order was shipped, but the Address in the Customers table is always the customer's current mailing address.

8 Part I: Designing Your Application

If you look at the table shown previously in Figure 1-3, you might conclude that the Subject column is actually a separate subject that should be in its own table. If the subject has additional related information, such as the grade level or full course description, you should probably have a separate table containing this information. In this case, the contents of the Subject column in the sample could well be the foreign key that links to the Subjects table containing the details about each subject—provided it is also the primary key in the Subjects table.

See the discussions about artificial primary keys in “Using Good Design Techniques,” page 23 for tips about defining primary keys and foreign keys.

Problem #4: Oversimplifying Your Tables

It’s also possible to oversimplify or “over-normalize” your table design. Did you create a separate table for customer addresses, but each customer has only one address? Did you create a separate table for “categories” or “types,” but that table contains only the one field? Perhaps you have oversimplified your table design. Consider the table designs in Figures 1-4 and 1-5.

You should always be suspicious of tables that contain a single field or an AutoNumber primary key and a single field. Why is a subject in its own table if it contains only one descriptive field? In both cases shown in Figures 1-4 and 1-5 the table for contact types has only one field. However, note the referential integrity relationship shown between contact types and contacts. The referential integrity relationship ensures that a customer type cannot be entered in the Contacts table that does not exist in the ContactTypes table, so the ContactTypes table might be a valid use of a table with a single field. Keep in mind that you could achieve the same result by defining a validation rule on the ContactType field in the Contacts table, but using a separate table like this provides the flexibility to add a new valid contact type in the future by adding a row to ContactTypes without having to modify the Contacts table design.

In Figure 1-4, you can see two sets of addresses and phone numbers. Is this an invalid repeating group? If the application never needs to track more than two addresses (work and home), and most contacts have both a work address and a home address, this design is more efficient than creating a separate table of addresses because the application can fetch names and addresses without having to join two tables. Figure 1-5 shows an alternative design that can accommodate an unlimited number of addresses and phone numbers for any one contact. Should the phone numbers remain in the Contacts table in case you have no address for a contact? Should phone numbers be in a separate table altogether? It probably makes sense to leave the mobile phone number field in the Contacts table in any case—very few people have more than one mobile phone number.

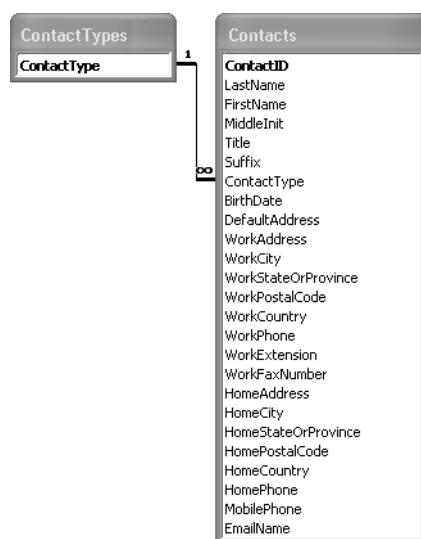


Figure 1-4 Tables to store contacts and contact types.

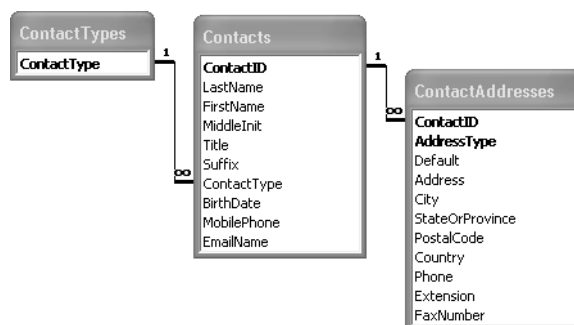


Figure 1-5 Tables to store contacts, contact types, and multiple contact addresses.

What's the correct answer? As the French would say, "ça dépend" ("it depends"). The bottom line is you must fully understand the data you need to store in your tables and how that data will be used.

- Do not create a separate table where doing so forces you to always join two or more tables to solve most problems.
- Do create a separate table when you might need to store an unknown and unrestricted number of related items (addresses and phone numbers in this example) for another subject table.
- Do create a separate table to hold repeating groups as rows when you must perform an analysis on the members of the groups—such as counting the phone numbers in a particular area code.

10 Part I: Designing Your Application**Problem #5: Segmenting Tables**

When you're working with large amounts of data in a spreadsheet, you are often forced to segment your data in some way because the number of rows that you can store in a worksheet is limited. You might segment your active data by a date range (perhaps sales details by month) or by some other natural value such as sales region or department.

Segmenting data in a database application is a bad idea. First, you must either design separate queries, forms, and reports to edit and view the segmented data, or you must write code to dynamically load the desired recordset from the correct segment. Second, if you want to analyze data across more than one segment (for example, analyze the sales data for a full year from several monthly files), you must use a union query to gather the data from multiple segments, and union queries can be very slow.

Although you are limited to a total of 2 gigabytes in a single JET .mdb file, you can easily store several hundred thousand rows within this limitation. You can also place a single very large table in its own .mdb file, define other supporting tables in another .mdb file, and use linked tables to your application to bring all the data together. If you are using SQL Server to store your data—other than the Microsoft SQL Server Desktop Engine (MSDE), which is also limited to a total of 2 gigabytes—you are limited only by the amount of available hard drive space on the server.

You might argue that segmented tables make it easier to secure data so that users can have access only to those segments to which they have authority. However, you can solve this problem when all data is in a single large table by creating With OWNERACCESS queries that filter the rows in Access, or by defining secured views in SQL Server.

So, other than for historical summary tables, it does not make sense to segment your active data into separate tables for an Access application. Of course, you should define indexes on fields that might be used by your application to filter the data for a specific task or user. For example, if some users should be able to work with data only from their own department, you should define an index on the department field.

Problem #6: Storing Calculated Values

Do not store calculated values in your tables. It is preferable to perform the appropriate calculation in a query because Access provides no mechanism in the database engine to automatically maintain a calculated value for you. The simple test for compound subjects applies here as well. If changing a field value means you must also change a dependent field, that second field is probably a calculated value.

Note, however, that including a calculated value might be necessary for performance reasons—such as maintaining the current sum of items in inventory over thousands of inventory transaction rows. You should plan such a calculated value carefully. In the forms you provide to update the other fields in the calculation, you must write code to recalculate the value and

save it. You might also need to write verification code that the user can run periodically to ensure that the stored calculated values match what's in the fields that participate in the calculation. Keep in mind that if you allow access to your application tables outside your application code, users might change values that would result in your calculated value being incorrect.

Problem #7: Defining Field Lookup Properties

Microsoft Access for Windows 95 introduced the ability to define lookup properties for fields in tables to allow you to predefine how you want the field displayed in a datasheet, form, or report. For example, if you have a DepartmentID field in an Employees table that stores the foreign key value of the department for which the employee works, you can display the department name rather than the number value when you look at the data. If you're displaying a Yes/No field, you can provide a drop-down list that shows options for "invoiced" and "not invoiced" instead of yes and no or true and false. Using a lookup—a combo box or list box on a form to select primary key values from a related table to set a foreign key—is not a bad idea in some instances. For example, you can open the frmOrders form in the Inventory Management application (Inventory.mdb), and you'll see combo boxes for both Customer and Tax Status. However, I avoid defining lookup properties for fields in my tables unless the "looked up" value is also the foreign key.

In Chapter 3, "Designing a Client/Server Application," you'll learn that using a combo box or list box to display a row source of several thousand rows is detrimental to obtaining good client/server performance. Chapter 3 also shows you techniques to either minimize the number of rows or use an alternate method to choose a value from a lookup.

First and foremost, a lookup masks the actual underlying value for sorts and searches. If you open tblOrders in the Inventory application, as shown on the next page in Figure 1-6, you can see that it has no lookup properties defined for any of the fields. You see the numeric value of the CustomerID foreign key field (the Caption property for the field is set to Customer), not customer names, and you see the numeric tax status code, not a text description.

To see how a lookup on customer name appears, open qxmplLookup in the Inventory application, as shown in Figure 1-7, on the next page.

You can now see the customer name instead of the CustomerID field value, and opening the combo box list shows you additional useful information about each customer. However, if you look at this query in Design view, you'll find that the query is sorted in ascending order on the CustomerID field. Do you think "Ottolies Käsäladen" should appear before "Great Lakes Food Market" if the query is sorted correctly? The query output is correctly sorted on the underlying numeric customer ID value, not the name.

Order ID	Customer	Order Date	Ship Date	Invoiced	Tax Status	Tax Rate
1	68	9/1/2004	9/2/2004	✓	0	0.000
2	119	9/1/2004	9/2/2004	✓	0	0.000
3	63	9/2/2004	9/3/2004	✓	0	0.000
4	1	9/2/2004	9/3/2004	✓	0	0.000
5	47	9/2/2004	9/3/2004	✓	0	0.000
6	64	9/2/2004	9/3/2004	✓	0	0.000
7	46	9/2/2004	9/3/2004	✓	0	0.000
8	93	9/2/2004	9/3/2004	✓	0	0.000
9	166	9/2/2004	9/3/2004	✓	0	0.000
10	78	9/2/2004	9/3/2004	✓	0	0.000
11	83	9/2/2004	9/3/2004	✓	0	0.000
12	20	9/2/2004	9/3/2004	✓	0	0.000
13	68	9/2/2004	9/3/2004	✓	0	0.000
14	82	9/2/2004	9/3/2004	✓	0	0.000
15	102	9/2/2004	9/3/2004	✓	0	0.000
16	140	9/3/2004	9/4/2004	✓	0	0.000
17	87	9/3/2004	9/4/2004	✓	0	0.000
18	168	9/3/2004	9/4/2004	✓	0	0.000
19	89	9/3/2004	9/4/2004	✓	0	0.000
20	141	9/3/2004	9/4/2004	✓	0	0.000
21	113	9/3/2004	9/4/2004	✓	0	0.000

Figure 1-6 The tblOrders table in the Inventory application has no lookup properties defined for fields.

Order ID	Customer	Order Date	Ship Date	Invoiced	Tax Status	Tax Rate
4	Ottiles Kasseladen	9/2/2004	9/3/2004	✓	0	0.000
34	Ottiles Kasseladen	9/4/2004	9/5/2004	✓	0	0.000
82	Ottiles Kasseladen	9/9/2004	9/10/2004	✓	0	0.000
129	Ottiles Kasseladen	9/16/2004	9/16/2004	✓	0	0.000
207	Paris spécialités	9/23/2004	9/23/2004	✓	0	0.000
479	Pericles Comidas clásicas	10/27/2004	10/27/2004	✓	0	0.000
602	Piccolo und mehr	11/10/2004	11/10/2004	✓	0	0.000
634	Princesa Isabel Vinhos	11/13/2004	11/13/2004	✓	0	0.000
823	Que Delicia	12/4/2004	12/4/2004	✓	0	0.000
1002	Queen Cozinha	12/27/2004	12/27/2004	✓	0	0.000
1124	QUICK-Stop			✓	0	0.000
1169	Ottiles Kasseladen	1/16/2005		✓	0	0.000
1185	Ottiles Kasseladen	1/18/2005		✓	0	0.000
1203	Ottiles Kasseladen	1/19/2005		✓	0	0.000
1353	Ottiles Kasseladen	2/6/2005		✓	0	0.000
1419	Ottiles Kasseladen	2/13/2005		✓	0	0.000
1529	Ottiles Kasseladen	2/27/2005		✓	0	0.000
33	Great Lakes Food Market	9/4/2004	9/5/2004	✓	0	0.000
73	Great Lakes Food Market	9/6/2004	9/9/2004	✓	0	0.000
108	Great Lakes Food Market	9/12/2004	9/13/2004	✓	0	0.000
181	Great Lakes Food Market	9/19/2004	9/20/2004	✓	0	0.000

Figure 1-7 The qxmplLookup query has lookup properties defined for the CustomerID field. The query is sorted on the CustomerID field, but the result appears to not be in the correct sequence because the lookup masks the underlying value.

You can actually click in the Customer column and then click the Sort Ascending button on the toolbar to see the rows in sequence by customer name, not customer ID. When you do this, Access sets the OrderBy property of the query to something strange:

`Lookup_CustomerID.CustomerName`

The field CustomerName does not exist in the query at all! Clearly, a field name prefixed with “Lookup_” has special meaning to Access. To correctly honor the sort request, Access is forcing the JET engine to perform a hidden join with the tblCustomers table so that it can sort on what you see rather than the actual underlying value.

To see one of the other bad side effects of lookup properties, open the frmOrders form in the Inventory application, click in the Customer field, and click the Find button (binoculars) on the toolbar. The first order should be for customer Around the Horn (unless you've reloaded the sample data), so you should be able to find all customers whose names begin with the letter A. Type that letter in the Find What box, choose **Start of Field** in the Match box, clear the **Search Fields As Formatted** option, and click Find Next. Your result should look like Figure 1-8.

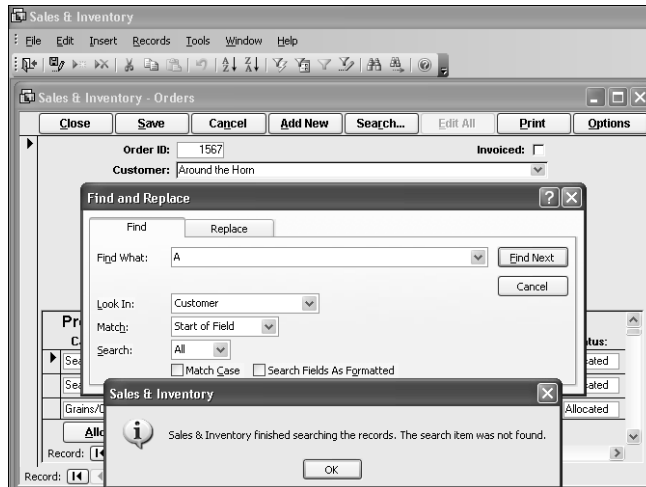


Figure 1-8 Displaying an apparently incorrect Find result in a combo box lookup.

The problem, of course, is Access is searching on the underlying numeric customer ID value (to which the combo box is bound), not what you see displayed in the control. So, there is no match for a numeric value starting with the letter A. Of course, you can execute the Find command again, but first select the **Search Fields As Formatted** option. You'll find the records you want, but Access is performing the Find on the looked-up values, not the actual field value. If you have ever performed a Find on a very large table using the **Search Fields As Formatted** option, you know that using this option is very slow and inefficient.

As with a query, you can click in the Customer field on the form, click the Sort Ascending button, and see a correct result sorted on the looked-up value. If you ask Access to do this in the frmOrders form, you can find out what really happened by opening the Immediate window (press Ctrl+G) and entering

```
?Forms!frmOrders.OrderBy
```

Visual Basic will respond

```
Lookup_cmbCustomerID.CustomerName
```

The name of the combo box control is "cmbCustomerID." Again, the "Lookup_" prefix has a special meaning. Access is performing a hidden join with the row source of the combo box

14 Part I: Designing Your Application

(a query on tblCustomers) to be able to sort the rows by what you see. Interestingly, the record source of the frmOrders form uses a query that does include tblCustomers along with tblOrders, so the extra join is redundant and wasteful of resources.

You can also highlight the customer name in the combo box and click the Filter By Selection button on the toolbar to see only the orders for that customer. Again, you can find out what happened by going to the Immediate window and entering

```
?Forms!frmOrders.Filter
```

Visual Basic will respond with something like

```
((Lookup_cmbCustomerID.CustomerName="Alfreds Futterkiste"))
```

As it did with the sort on the combo box, Access is forcing the JET engine to do an extra hidden join on the row source of the combo box to get the correct result. Even worse, unless you've secured your database to disallow Modify Design permission on your forms, Access silently saves the modified Filter and OrderBy properties when the user closes the form! If your code depends on the contents of the Filter property to correctly filter related objects (such as a report to display the currently filtered orders or customers), your code will likely break if you try to borrow the form filter. Another form might not have a cmbCustomerID control, and reports don't understand the strange "Lookup_" filter syntax.

What about the Lookup Wizard entry under Data Type in table Design view? I recommend that you *never* use this wizard. It often builds strange SQL for the Row Source property, it always defines a relationship between the table you're editing and the lookup table, and it defines additional indexes on both fields. If the lookup table contains only a few rows, the index is a waste of time and resources. As you probably know, there's a limit of 32 indexes on a table. I have seen some cases where I haven't been able to build all the indexes I need because the wizard built these unnecessary indexes.

Finally, if you have secured your data, using a lookup forces you to grant permissions not only on the record source of the form but also the row source of all combo boxes. If you use a With Owner Permissions query as the row source, filters and sorts will fail because the hidden join required to do a filter or sort on a lookup control does not directly use the query.

For more details about the evils of lookup properties, go to <http://www.mvps.org/access/lookupfields.htm>.

Problem #8: Defining Validation Rules Without Validation Text

The ability to define validation rules for both fields and tables in an Access database is a great feature. However, if you specify a validation rule but no validation text, Access generates an ugly and cryptic message that your users might not understand. The tblPOProducts table has a complex validation rule that prohibits setting the Invoiced field to True without also

supplying an InvoiceNumber, and vice versa. Without any validation text defined, violating this rule results in the interesting error message shown in Figure 1-9.



Figure 1-9 What the user sees when a complex validation rule has no companion validation text.

A user seeing this message is likely to be completely bewildered and have no idea what is wrong. Unless you like getting lots of support calls, I recommend that you always enter a custom validation text message whenever you specify a validation rule. Figure 1-10 shows the actual validation text defined for this rule.

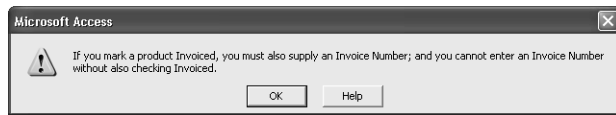


Figure 1-10 Validation text can describe what's wrong in terms a user can understand.

Of course, you must be careful to write validation text that you're confident your users will understand. A simple message such as "You entered an invalid value" probably won't solve the problem, either!

Problem #9: Incorrectly Using Data Types

Designing tables for your application is a big challenge. You must not only identify the subject for each table, the attributes (fields) required to define each subject, and the relationships between subjects, but also select the best data type for storing each attribute. Table 1-1 summarizes the data types available in an Access desktop (.mdb) database.

Table 1-1 Access Data Types

Data Type	Usage	Size
Text	Alphanumeric data	Up to 255 characters
Memo	Alphanumeric data—sentences and paragraphs	Up to about 1 gigabyte (but controls to display a memo are limited to the first 65,535 characters)
Hyperlink	A link address to a document or file on the World Wide Web, on an intranet, on a local area network (LAN), or on your local computer	Up to about 1 gigabyte
Number	Numeric data	1, 2, 4, 8, or 16 bytes
AutoNumber	Unique value generated by Access for each new record	4 bytes (16 bytes for Replication ID)
Currency	Monetary data, stored with 4 decimal places of precision	8 bytes

Table 1-1 Access Data Types

Data Type	Usage	Size
Yes/No	Boolean (true/false) data; Access stores the numeric value zero (0) for false, and minus one (–1) for true.	1 bit
Date/Time	Dates and times	8 bytes
OLE Object	Pictures, graphs, or other ActiveX objects from another Windows-based application	Up to about 2 gigabytes

Storing Character Data

For character data, you should normally select the Text data type. You can control the maximum length of a Text field by setting the Field Size property. Use the Memo data type only for long strings of text that might exceed 255 characters.

The Memo data type has some serious limitations, as listed here:

- Although later versions of Access allow you to define an index on a field of the Memo data type, Access indexes only the first 255 characters of each field.
- You cannot use a Memo field in a union query.
- If you include a Memo field in a totals (aggregate) query and use the field in the Group By clause, Access displays only the first 255 characters of the field in the query output.
- Form and report controls in an Access application can display only the first 65,535 characters stored in a Memo field.
- Unless you have installed the latest version of the JET database engine (SP 8 or later), Access in many cases truncates Memo fields to 256 characters when importing from or exporting to Excel. You can obtain the latest JET database engine update via Windows Update.

The Hyperlink data type lets you store a simple or complex link to an external file or document. Internally, Hyperlink is a Memo data type with a special flag set to indicate that it is a link—so the limitations that apply to the Memo data type also apply to Hyperlink. The link can contain a Uniform Resource Locator (URL) that points to a location on the World Wide Web or on a local intranet. It can also contain the Universal Naming Convention (UNC) name of a file on a server on your local area network (LAN) or on your local computer drives. The link can point to a file that is in Hypertext Markup Language (HTML) or in a format that is supported by an ActiveX application on your computer.

The link itself can have up to four parts:

- An optional descriptor that Access displays in the field when you're not editing the link. The descriptor can start with any character other than a pound sign (#) and must have

a pound sign as its ending delimiter. If you do not include the descriptor, you must start the link address with a pound sign.

- The link address expressed as either a URL (beginning with a recognized Internet protocol name such as `http:` or `ftp:`) or in UNC format (a file location expressed as `\\server\share\path\filename`). If you do not specify the optional descriptor field, Access displays the link address in the field. Terminate the link address with a pound sign (`#`).
- An optional subaddress that specifies a named location (such as a cell range in a Microsoft Excel spreadsheet or a bookmark in a Microsoft Word document) within the file. Separate the subaddress from the ScreenTip with a pound sign (`#`). If you entered no subaddress, you still must enter the pound sign delimiter if you want to define a ScreenTip.
- An optional ScreenTip that appears when you move your mouse pointer over the hyperlink.

For example, a hyperlink containing all four items might look like the following:

Viescas Download Page#<http://www.viescas.com/Info/links.htm#Downloads>#Click to see the files you can download from Viescas.com

A hyperlink that contains a ScreenTip but no optional subaddress might look like this:

Viescas.com Books#<http://www.viescas.com/Info/books.htm>##Click to see recommended books on Viescas.com

Storing Numbers

When you select the Number data type, you should think carefully about what you enter as the Field Size property because this property choice will affect precision as well as length. (For example, integer numbers do not have decimals.)



Note The table design grid in Access allows you to define fields that can store numbers, including AutoNumber, Byte, Currency, Decimal, Double, Integer, Long Integer, and Single. You can find AutoNumber and Currency listed as separate data types in the Data Type list in the top part of the design grid, but all the other data types are subtypes of the Number data type, and you indicate the data type you want by setting the Field Size property. Internally, each of these numeric types is unique—the JET database engine does not have a Number data type with subtypes. So, when you see the text in this book reference the Single data type, it means the Number data type in the table design grid with the Field Size property set to Single.

The AutoNumber data type is specifically designed for automatic generation of primary key values. (See “Avoid Artificial Primary Keys in Tables,” page 26, to learn why I think this is a bad idea.) Depending on the settings for the Field Size and New Values properties you choose for an AutoNumber field, you can have Access create a sequential or random long integer. You

can include only one field using the AutoNumber data type in any table. If you define more than one AutoNumber field, Access displays an error message when you try to save the table.

Table 1-2 lists the available Field Size settings for the Number and AutoNumber data types.

Table 1-2 Field Size Properties for Numeric Fields

Data Type	Options, Description
Number	Byte. A single-byte integer containing values from 0 through 255.
	Integer. A 2-byte integer containing values from -32,768 through +32,767.
	Long Integer. A 4-byte integer containing values from -2,147,483,648 through +2,147,483,647.
	Single. A 4-byte floating-point number containing values from -3.4×10^{38} through $+3.4 \times 10^{38}$ and up to seven significant digits.
	Double. An 8-byte floating-point number containing values from -1.797×10^{308} through $+1.797 \times 10^{308}$ and up to 15 significant digits.
	Replication ID.* A 16-byte globally unique identifier (GUID).
AutoNumber	Decimal. A 12-byte integer with a defined decimal precision that can contain values from -10^{28} through $+10^{28}$. The default precision (number of decimal places) is 0 and the default scale (total number of digits) is 18. You can store up to 28 significant digits in the Decimal field size.
	Long Integer. A 4-byte integer containing values from -2,147,483,648 through +2,147,483,647 when New Values is Random or from 1 to +2,147,483,647 when New Values is Increment.
	Replication ID. A 16-byte globally unique identifier (GUID).

* In general, the Replication ID field size should be used only in a database that is managed by the Replication Manager.

Single and Double field sizes use an internal storage format called floating point that can handle very large or very small numbers but is somewhat imprecise. If the number you need to store contains more than 7 significant digits for Single or more than 15 significant digits for Double, the stored number will be rounded. For example, if you try to save 10,234,567 in a Single, the actual value stored will be 10,234,570. Likewise, Access stores 10.234567 as 10.23457 in a Single. If you want absolute fractional precision, use the Decimal or Currency data type instead.

In addition, floating-point numbers store the fractional part of any number as the sum of negative powers of two. That is, the first bit to the right of the decimal point is 0.5, the second bit is 0.25, the third bit is 0.125, and so on. You're more likely to get an imprecise fractional number using a Single data type because that data type has a maximum of only 24 bits (1 bit is used for the sign and 7 bits are used for the exponent) in which to resolve an accurate fractional value. You're also more likely to see a precision error when you multiply two fractional values. For example, you can go to the Immediate window (Ctrl+G) and enter

```
?1.131212 * 3.3
```

Visual Basic responds with the correct answer:

3.7329996

However, if you force one of the numbers to a Single data type, you can see the effect of imprecise representation of the fractional value. Try entering

?CSng(1.131212) * 3.3

Visual Basic responds with an imprecise answer:

3.73299958705902

Floating-point numbers are particularly nasty when you're attempting to calculate precisely rounded numbers—such as a percentage of a currency amount rounded to pennies. In general, you should use Single and Double only in scientific applications or applications that must deal with extremely large or extremely small numbers.

Use the Currency or Decimal data type to obtain a precise value—up to 4 decimal places for Currency and up to 28 decimal places for Decimal. You should generally use the Currency data type for storing money values, but you can also use it to store any number for which you need precisely 4 decimal places.

What makes Currency and Decimal precise is the fact that Access stores the entire number as an integer, but with an assumed number of decimal, not binary, digits to the right of the decimal point. For example, Access stores the number 1.131212 in a Decimal field with scale 12 and precision 8 as the integer value 000113121200. Access keeps track of the fact that the decimal place occurs 8 digits in from the right, and the math processor (actually a component of Windows) makes the appropriate adjustment when you multiply or divide decimal values.

For more information about problems with calculations using numeric data types in Access, go to <http://www.fmsinc.com/tpapers/math/index.html>.

The Yes/No data type in Access is actually a Byte numeric field. Use the Yes/No data type to hold Boolean (true or false) values. This data type is particularly useful for flagging accounts paid or not paid or orders filled or not filled. When a Yes/No field contains a zero, Access interprets the value as False or No. When a Yes/No field is nonzero, Access interprets the value in most cases as True or Yes. In fact, you can test any numeric field for True in a query, and the query will return all rows that have a nonzero value in the field.

If you assign True or Yes to a Yes/No field, Access turns on all the bits in the byte. (All bits on in a Byte field is the numeric value -1.) If you assign a False or No to a Yes/No field, Access sets all the bits to 0.



Caution If you upsize your application tables to SQL Server in the future, be aware that SQL Server stores a Yes/No data type as a single bit. A False is still 0, but a True value is 1, not -1. If you explicitly test for -1 as a True value, your test will fail on data that has been upsized.

Storing Dates and Times

The Date/Time data type is useful for calendar or clock data and has the added benefit of allowing calculations in seconds, minutes, hours, days, months, or years. Internally, the Date/Time data type is a Double floating-point number. The integer portion is the number of days since December 30, 1899 (negative values represent days prior to that date), and the fractional portion is a fraction of a day, accurate to seconds. When you store only a time value in a Date/Time field, the integer portion is 0, and the fractional portion contains the time. For example, Access stores the value for 12:00 noon as 0.5.

Of course, Access never displays the underlying numeric value for a Date/Time field in the user interface. You can control what portion of the value Access displays by setting the Format property for the field. If you don't specify a format, Access displays the Date/Time value using the Short Date format defined in the regional settings section of Windows Control Panel. Many novice users think that what they see in the user interface is what is actually stored in the field. Access always stores a full Date/Time value even though the application might be displaying only month and year.

Unless you restrict what users can enter with an input mask, a user can enter both a date and time component into a field that displays only the date portion. If you do not restrict what a user can enter, you might obtain incorrect results when you create a query (or a search string in code) that includes criteria to search only for date values. For example, to search for rows in the month of January 2005, you might enter the following criteria for a Date/Time field:

```
tblSales.DateSold BETWEEN #01 JAN 2005# And #31 JAN 2005#
```

This criteria is equivalent to entering

```
tblSales.DateSold >= #01 JAN 2005# And tblSales.DateSold <= #31 JAN 2005#
```

If users have been entering a time as well as a date, you will potentially not see all the records for the last day of the month because the simple date value for January 31, 2005, is a simple integer value that is less than a Date/Time value of January 31, 2005, 9:15 A.M.—an integer plus a fraction that represents 9:15 A.M. To correctly perform the search, you should use the following:

```
tblSales.DateSold >= #01 JAN 2005# And tblSales.DateSold < #01 FEB 2005#
```

When you understand the internal format of the Date/Time data type, you can subtract one Date/Time value from another to obtain the interval between the two values. (You can also use the *DateDiff* function, but subtracting is more efficient.)

You can also add Date/Time values that contain a time component to calculate a total time—but be aware that when the total exceeds 24 hours, the total time overflows into the date portion of the Date/Time value. For example, adding 9 hours (9:00 A.M.) and 4 hours (4:00 A.M.) yields 13 hours (1:00 P.M.), but adding 14.5 hours (2:30 P.M.) and 16 hours (4:00 P.M.) yields 1 day and 6.5 hours (which equates to December 31, 1899, 6:30 A.M.). However, because you know the internal representation of Date/Time is days and fractions of days, you can add a series of elapsed times and then multiply by 24 (the number of hours in a day) to obtain the total elapsed hours and fractions of hours.

Arithmetic operations with Date/Time values follow these rules:

- Adding or subtracting two Date/Time values yields a Date/Time value.
- Adding or subtracting a number to or from a Date/Time value yields a Date/Time value. For example, you can add 1 to “today” to calculate “tomorrow,” or you can subtract the integer portion of a Date/Time value (using the *Int* function) from itself to obtain the time component.
- Multiplying a Date/Time value by a number or dividing a Date/Time value by a number yields a Double data type.



Tip When you create a sum of elapsed times in a query or report, and the total time might exceed 24 hours, you can display the correct result by using this expression (substitute the name of your summary field for *TotElapsed*):

```
Format((Int([TotElapsed]) * 24) + Hour([TotElapsed]), "#,##0") & ":" &  
Format([TotElapsed], "nn:ss")
```

The *Int* function extracts the integer number of days, the *Hour* function extracts the hours that are less than a day, and the second *Format* function extracts the minutes and seconds.

Storing Objects

The OLE Object data type allows you to store complex data, such as pictures, graphs, or sounds, which can be edited or displayed through a dynamic link to another Windows-based application. For example, Access can store and allow you to edit a Microsoft Word document, a Microsoft Excel spreadsheet, a Microsoft PowerPoint presentation slide, a sound file (.wav), a video file (.avi), or pictures created using the Paint or Draw application.

Be aware, however, that Access stores not only the object but also a “thumbnail” of the object in bitmap format to enable it to display a representation of the object in a bound object frame in a form or report. As a result, storing compacted images in formats such as JPEG or GIF can require much more space in your .mdb file than the size of the original file because of the space occupied by the bitmap thumbnail.

Also, if you attempt to display an OLE Object field on a Web page, your browser cannot handle the internal format of the OLE Object data type. To solve both problems, consider using a Text or Hyperlink field to store a link to the OLE Object. To display a picture object stored as a text link on a form or report, you can write code to load the object path into an image control.

In summary, consider the following five points when deciding on a data type:

1. Understand the limitations of the Memo data type when sorting on a Memo field, using a Memo field in union or totals queries, and exporting a Memo field before choosing the Memo data type to store text that might be longer than 255 characters.
2. Do not use Single or Double when your application requires precise calculations; use Currency or Decimal instead.
3. Understand how SQL Server handles the Yes/No data type differently if you need to upsize your data.
4. Know the storage format of the Date/Time data type so that you create correct search criteria and use it in calculations.
5. Use a link to external objects rather than storing object data in your tables.

Problem #10: Naming Fields Incorrectly

Microsoft Access gives you lots of flexibility when it comes to naming your fields. A field name can be up to 64 characters long and can include any combination of letters, numbers, spaces, and special characters except a period (.), an exclamation point (!), an accent grave (`), and brackets ([]); however, the name cannot begin with a space and cannot include control characters (ANSI values 0 through 31).

In general, you should give your fields meaningful names and should use the same name throughout for a field that occurs in more than one table. You should avoid using field names that might also match any name internal to Microsoft Access or Visual Basic. For example, all objects have a Name property, so it's a good idea to qualify a field containing a name by calling it something like CustomerName or CompanyName. You should also avoid names that are the same as built-in functions, such as *Date*, *Time*, *Now*, or *Space*. See Microsoft Access Help for a list of all the built-in function names.

Although you can use spaces anywhere within names in Access, you should try to create field names and table names *without* embedded spaces. Most SQL databases to which Access can attach do not support spaces within names. If you ever want to move your application to a client/server environment and store your data in an SQL database such as Microsoft SQL Server or Oracle, you'll have to change any names in your database tables that have an embedded space character.

Using Good Design Techniques

As you can see from the previous section, avoiding design mistakes saves you time and effort later as you develop or change your application. However, you can also greatly improve the way your application works and enhance your users' experience by employing additional design techniques as you build your tables in Access.

Avoid Allowing Both Null and Zero Length Values in Text

Relational databases support a special value in fields, called a *Null*, that indicates an unknown value. In contrast, you can set Text, Memo, or Hyperlink fields to a zero-length string to indicate that the value of a field is known but the field is empty.

Why is it important to differentiate Nulls (unknown values) from zero-length strings? Here's an example: Suppose you have a database that stores the results of a survey about automobile preferences. For questionnaires on which there is no response to a color-preference question, it is appropriate to store a Null. You don't want to match responses based on an unknown response, and you don't want to include the row in calculating totals or averages.

On the other hand, some people might have responded "I don't care" for a color preference. In this case, you have a known "no preference" answer, and a zero-length string might be appropriate. You can match all "I don't care" responses and include the responses in totals and averages.

But you should not allow a Text, Memo, or Hyperlink field to be both Zero Length (Allow Zero Length property set to Yes) and Null (Required property set to No and no setting in the Validation Rule property) unless you truly want to be able to store either an empty string or a Null. The user interface in Access displays both a Null and a zero-length string ("" the same way (both appear blank), so allowing both can be confusing to the user. In truth, you should probably store "No Preference" rather than allow both a Null or Zero Length String value because the user won't be able tell the difference when looking at the data displayed in a datasheet, form, or report.

You can join tables on zero-length strings, and two zero-length strings will compare to be equal. However, for Text, Memo, and Hyperlink fields, you must be sure that the Allow Zero Length property is set to Yes to allow users to enter zero-length strings. Otherwise, Access converts a zero-length or all-blank string to a Null before storing the value. In fact, Access removes all trailing blanks from any text string before storing the string. So, if you type nothing but blanks in a field and save the record, Access reduces the string to zero length—and stores a Null if Allow Zero Length is No. If you also set the Required property of a character field to Yes, Access stores a zero-length string if the user enters either "" (two double quotes with no space) or nothing but blanks in the field.

Nulls have special properties. A Null value cannot be equal to any other value, not even to another Null. This means you cannot join (link) two tables on Null values. Also, the question

24 Part I: Designing Your Application

“Is A equal to B?” when A, B, or both A and B contain a Null can never be answered “yes.” The answer, literally, is “I don’t know.” Likewise, the answer to the question “Is A not equal to B?” is also “I don’t know.” Finally, Null values do not participate in aggregate calculations involving such functions as *Sum* or *Avg*. You can test a value to determine whether it is a Null by comparing it to the special keyword NULL or by using the *IsNull* built-in function.



Caution As of Access version 10 (also known as Access XP or Access 2002), Microsoft changed the default setting for the Allow Zero Length property of Text, Memo, and Hyperlink fields to Yes. If you do not require an entry in a character field (either by setting the Required property to Yes or by defining a validation rule), you should be sure to reset the Allow Zero Length property to No.

Use a Validation Rule Instead of Required = Yes

You certainly should take advantage of the validation features in Access to ensure that users supply a valid value for fields in your tables. When you want to require that a user enter a value, but you don’t need to perform any other validation, you might be tempted to set the Required property of the field to Yes. However, when a user fails to supply a value or attempts to clear a required value, Access displays a cryptic message as shown in Figure 1-11.

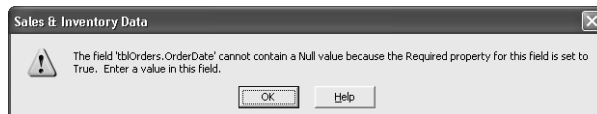


Figure 1-11 Access displays a confusing error message when you set Required to Yes and the user fails to provide a value.

Instead of setting the Required property to Yes, set the Validation Rule property to **Is Not Null** and include an appropriate message in the Validation Text property. When you leave the Required property set to No and define an Is Not Null validation rule, Access displays your more meaningful message, as shown in Figure 1-12.



Figure 1-12 When you set the Validation Rule property to **Is Not Null**, you can define a more meaningful error message that Access displays when the user fails to enter a value.

Note that you must leave the Required property set to No because Access checks Required before testing the validation rule.

Never Define Subdatasheets for Tables (or Queries)

You should set the Subdatasheet Name property to [None] in the Table Properties window for all tables and leave the property blank in the Query Properties window for all queries. If you define a subdatasheet, Access also fetches the rows in the subdatasheet when you open the table or query. This can have a huge negative performance impact if the table is linked over a network.

If you're using linked tables and the database containing the data has no relationships defined, in some cases Access will scan all other tables looking for a related table based on key names. This scanning process can take from several seconds to up to a minute before Access gives up and finally displays the data.

Also, if you have secured your tables, all users must have permissions to both the table(s) displayed in the main datasheet as well as all of the tables in any subdatasheet.



Caution Consider also that any production application should not allow the user to see table or query datasheets because you cannot enforce complex business rules. Any data validation in a table or query datasheet depends entirely on the validation and referential integrity rules you defined for your tables because you cannot define any Visual Basic code behind tables or queries.

Know That Referential Integrity Creates Extra Indexes

When you design your tables, you should always take advantage of the referential integrity feature in Access to ensure that the relationships between your tables are enforced. However, you should understand that Access always builds a hidden system index on the foreign key(s) in the dependent table—it needs to do this to be able to enforce the integrity rule. Also, this additional index counts toward the maximum number of 32 indexes that you can define for each table.

Access does not display these extra indexes in the Indexes window in a table's Design view. The only way to discover the indexes is to examine the Indexes collection of the table by using Visual Basic. For example, if you open the tblPOProducts table in the InventoryData.mdb database in Design view and open the Indexes window, you can see three defined indexes—the primary key, an index on the LineNo field, and an index on the Posted field. To find out the actual count of indexes, open the Immediate window (press Ctrl+G) and enter

```
?CurrentDb.TableDefs("tblPOProducts").Indexes.Count
```

Visual Basic will tell you that there are actually six indexes defined! The additional indexes support the referential integrity relationships to the tblProducts table, the tblPurchaseOrder table, and the tblVendorProducts table. In the InventoryData.mdb database, I created a little

26 Part I: Designing Your Application

function that you can copy to any other database and run to list all tables and their indexes. You can find this function in the modExamples module in the Inventory and Membership sample databases. The code follows:

```
Public Function ListIndexes()
Dim db As DAO.Database, tdf As DAO.TableDef
Dim idx As DAO.Index, fld As DAO.Field
' Point to this database
Set db = DBEngine(0)(0)
' Loop through all tables
For Each tdf In db.TableDefs
' Skip system and hidden tables
If (tdf.Attributes And (dbSystemObject + dbHiddenObject)) = 0 Then
' Display the table name and index count
Debug.Print "Table: " & tdf.Name, _
" No. Indexes: " & tdf.Indexes.Count
' Loop through all indexes for this table
For Each idx In tdf.Indexes
' Display the index name
Debug.Print " Index: " & idx.Name
' Loop through all fields in this index
For Each fld In idx.Fields
' Display the field name
Debug.Print " Field: " & fld.Name
Next fld
Next idx
End If
Next tdf
Set fld = Nothing
Set idx = Nothing
Set tdf = Nothing
Set db = Nothing
End Function
```

You might well ask: “Because I can’t see the hidden index, I might define the index again in the table’s Design view. Does this create an unnecessary index in my database and waste space and resources?” Yes, that will create another index definition (which counts toward the total of 32 for the table), but Access recognizes when another index already exists with the same attributes, so the “duplicate” index is only an alias to the actual physical index. Access does use up a little space for the index definition, but it doesn’t waste any space or resources creating another physical index. To actually delete the index, you would need to delete both the index that you can see in the table’s Design view and the referential integrity rule.

Avoid Artificial Primary Keys in Tables

One of the critical rules of database design is each table must have a primary key to uniquely identify each row. If you adhere strictly to relational theory, you’ll collect all the attributes (fields) for a subject (table) and then identify one or more sets of attributes that are potentially

unique for each row—these are called *candidate keys*. From the set of candidate keys, you should choose the simplest key (one field if possible) to use as the actual primary key.

But in the real world, identifying a simple candidate key is often difficult if not impossible. For example, the typical customer table might have first name, middle initial, last name, address, city, region, postal code, country, phone number, fax number, and e-mail fields. Certainly the combination of first name and last name might be a candidate key, but the instant you encounter a second person with the same name at a different address, that candidate key fails. You could try adding another field, such as e-mail address, but this, too, fails when you encounter a customer who has no e-mail address.

Many Access developers simply give up, throw an AutoNumber field in the table, and define it as the primary key. I used to use AutoNumber in this manner all the time, but I have found that falling back on AutoNumber as a primary key creates a meaningless value (every field in a table should have meaning and be directly related to the subject of the table) and is just plain lazy. Because this artificial key is meaningless, there's a great temptation to never display it to the user at all.

I sometimes use a generated number as a primary key, but I display it to the user, and I write code to generate the “next” number for a new row. Also, whenever possible, I define one or more other fields in the table to be unique, or I write code to check for potential duplicates. For example, in the Inventory application, the tblProducts table has a numeric product ID field, but it also has a unique index defined on the combination of product category and description. So, a user cannot define two products with the exact same name in the same category. In the Membership Tracking application, each member has an application-generated numeric member ID field, but code in the form to edit members uses a Soundex comparison to check for a potentially duplicate name when adding a new record.

A purist might ask, “In the Inventory application, why not use the natural combination of the Category and Description fields as the primary key of the tblProducts table?” I chose to use a simple application-generated number to avoid carrying a compound key down to many related tables. If I were to use the combination of the Category and Description fields as the primary key, I would have to put copies of these two longer fields in all the dependent tables—tblOrderProducts, tblVendorProducts, tblPOProducts, and tblInventory.

I do expose the Product ID field to the user everywhere in the application. Over time, users will come to know that Jack's New England Clam Chowder is also product number 41. And if the application is ever extended in the future to use bar codes to track inventory, I'll need an associated numeric value to generate the codes stamped on incoming and outgoing shipments.

In truth, in a real inventory application, you'll probably have an actual product identifier (perhaps one provided by your suppliers) that you can use without reservation. Even large vendors like Amazon.com generate artificial keys for many products (they call it an ASIN), but they do use a “real” value wherever possible—such as the ISBN for books they sell. And many

retailers use an in-house generated product number—called an SKU or Stock Keeping Unit—to uniquely identify the products they sell. But the point is even these “artificial” numbers have meaning. The best way to find out if a neighborhood branch of a department store has a particular dress in a specific size and color is to know that product’s SKU when you call. If I want to create a link to a product on Amazon.com on my Web site, I must know that product’s ASIN.

The following sections discuss some of the additional pitfalls of using artificial primary keys.

Avoid Artificial Primary Keys in Lookup Tables

As noted in “Oversimplifying Your Tables,” page 8, it is quite common to create a lookup table that contains nothing more than a list of valid values for one field in a dependent table. Rather than hard-code a value list in a validation rule for the field in the dependent table, you can control the list of values by defining a referential integrity rule between the lookup table and the dependent table. Adding a new valid value later is easy—you add a row to the lookup table rather than change the dependent table design.

Consider the two pairs of tables shown in Figure 1-13.

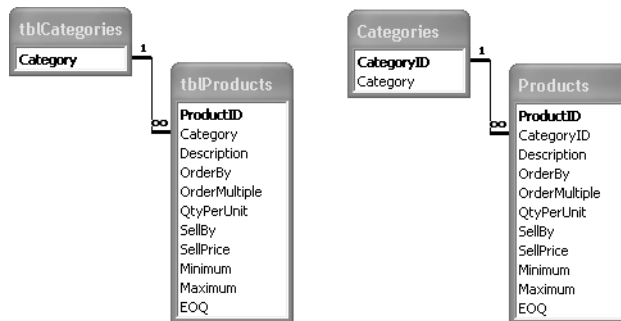


Figure 1-13 Two ways to use a lookup table to control the contents of a single field in a dependent table.

It’s a common mistake to create table definitions that look like the second pair, and the CategoryID field is usually a meaningless AutoNumber field. First and foremost, the design on the right encounters all the problems previously discussed in “Defining Field Lookup Properties,” page 11. Remember, you’re probably going to display the Category field in any combo box but bind the control to the CategoryID field, so any sort or filter the user applies generates an inefficient hidden join to the lookup table. Second, you’re not likely to ever expose the numeric value to the user—it’s the category description that has meaning, not the number.

You might try to argue that an AutoNumber or long integer key stores and indexes more efficiently than a varying length description field. Although that is true, the benefits of using the real text value far outweigh the small additional overhead of the text field. Unless you have hundreds of thousands of categories (not likely), you’re not going to notice the difference

either in the space required to store the data or the time it takes to index and search text instead of a number.

I must confess that I used to build many such single-value lookup tables with an AutoNumber primary key and a description. I finally broke the habit when a client asked me to build an application that provided a custom synchronization between a master database in the home office and several remote users. An administrator in the home office inadvertently deleted a lookup value that was being used by remote users, and there was no way to put it back—short of a brute-force insert query—with the original numeric ID.

The administrator tried to define the “category” again, but the system assigned it a new number under the covers. The data wouldn’t synchronize correctly anymore because the data coming in from remote users contained an invalid category identifier number that was forever gone from the main system. If I had used the description value as the primary key (resulting in a table with a single field), it would have been simple for the administrator to re-insert the missing value. I didn’t discover this problem until long after the application had gone into production. I ended up writing a lot of unnecessary code and some special data entry forms to solve the problem.

Do Not Use AutoNumber for Identifiers That Should Be Consecutive or That Have a Special Meaning

If you have worked with the AutoNumber data type at all, you know that it’s easy to end up with missing numbers—and not just because a user has deleted some rows. When a user starts to create a new row in a table that has a sequential AutoNumber field, Access generates the next number. If the user decides to use the Esc key to clear edits and abort the new row, that number is lost forever.

Sometimes, missing numbers in a sequence are no problem. However, particularly if you’re working with accounting data (such as checks, debits, credits, invoices, or purchase orders), maintaining a strict number sequence with no gaps can be very important.

Your business rules might dictate that your application generates numbers that have meaning and are in sequence. For example, in the Inventory application, a purchase order number for a related customer order is the order number followed by a three-digit sequence number. If the purchase order is generated to restock products to minimum inventory levels, the number is the digit 9 followed by the four-digit year number and a three-digit sequence number. You can look at a purchase order number and immediately identify the purpose of the order. The number has meaning—something you can’t achieve with an AutoNumber.

Avoid Artificial Primary Keys in Child Tables

When you have a set of related tables that have a cascading relationship down three or more levels, it’s often tempting to create an artificial simple primary key (often with an

AutoNumber) at lower levels to avoid having to cascade a compound primary key. Don't do it. Consider the two sets of tables in Figure 1-14.

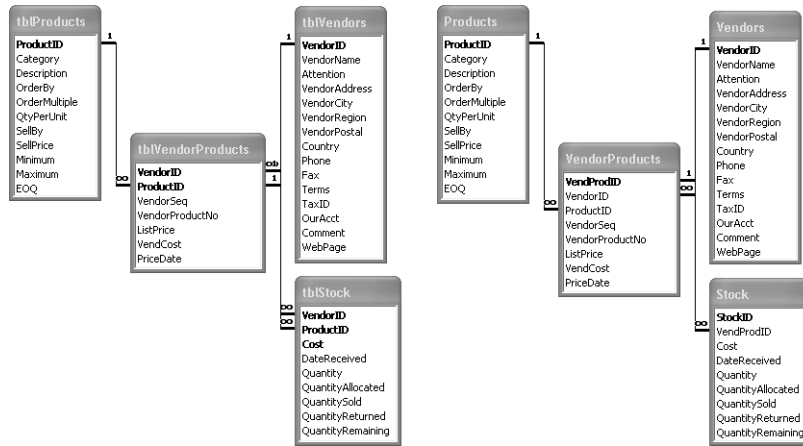


Figure 1-14 The correct and incorrect way to cascade relationships down three or more levels.

Again, the second design on the right is incorrect. Sure, VendorProducts and Stock have simple one-field primary keys, but at what cost? If I'm looking at a Stock record, what is the product in stock? Which vendor supplied the product? I'm forced to perform a join with the next table up the chain to answer these questions. Also, if I want the product description or the vendor name, I must join all three tables. I need the VendorID and ProductID in the VendorProducts table anyway, so why not use both fields as a compound primary key instead of using an artificial key? As you can see, using an artificial key masks the true identity of the parent rows when you navigate down to the third level.

In the correct design on the left, I have almost all the information about a stock entry that I need in the single tblStock table. If I need the product description, I can join tblProducts with tblStock and avoid the intermediate table. The same is true if I need the vendor name—I can join just two tables, not three. Yes, there's an ugly three-field compound key in tblStock, but the identity of the parent rows is unmistakable. (The Cost field is part of the primary key so that the application can track inventory valuation by each item's original cost and vendor.)

Turn Off Track Name AutoCorrect

Access version 9 (Access 2000) introduced a new feature called AutoCorrect for desktop databases (.mdb file extension) that attempts to automatically propagate changes you make to field names, table names, and query names to other dependent objects. You can enable this feature for individual databases, and the feature is turned on by default for new databases. AutoCorrect does a relatively good job of correcting field and table or query name changes. However, it won't fix references to these objects in Visual Basic code, and it misses name changes in certain types of expressions. It also won't fix changes to field names in linked tables.

To enable or disable this feature, open your database and choose Options from the Tools menu. Click the General tab to see the options as shown in Figure 1-15.

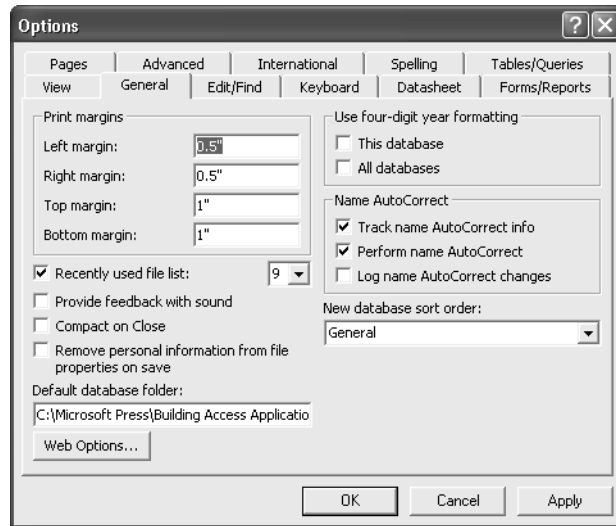


Figure 1-15 Setting the AutoCorrect options for an Access desktop database.

As you are developing your application, you might find it useful to turn on these options. Select **Track name AutoCorrect info** to instruct Access to generate the internal identifiers it uses to propagate name changes. Select **Perform name AutoCorrect** to instruct Access to check queries, forms, and reports each time you open them and automatically correct any names that have changed. Note that AutoCorrect works whether you open the object in Design view or in standard view—Datasheet view for queries; Form, PivotTable, or PivotChart view for forms; and Print Preview for reports. Select **Log name AutoCorrect changes** to ask Access to create a table called Name AutoCorrect Log containing rows that tell you the object type, object name, control name, property name, and old and new values for every change.

However, because this feature runs whenever you open any dependent object, it's a bad idea to leave this feature activated in a database that you distribute to users. If you change, for example, a field name in a table without opening and closing all objects that use that field, users must have Modify Design permission for all objects to be able to successfully run your application. In addition, the internal tracking codes take up additional (though nominal) space in your desktop database file.

And finally, if you enable **Perform name AutoCorrect**, you will not have any idea what changes Access makes on your behalf unless you also enable **Log name AutoCorrect changes** because Access makes the changes without notifying you. It doesn't even prompt you to save a changed object after you open it, make no changes yourself (other than the ones performed by AutoCorrect), and close it. It's this last "silent change" feature that causes me to recommend that you turn it off even when you're developing your application. To turn off this

32 Part I: Designing Your Application

feature for the current database, clear the **Track name AutoCorrect info** option in the Options dialog box.

Use the Description Property Wisely

Entering a Description property for every field in your table helps document your application. Because Access also displays the description on the status bar, paying careful attention to what you type in the Description field can later pay big dividends as a kind of mini-help for the users of your database. Also, since this data propagates automatically, you probably don't want to type something nonsensical or silly. Typing **My boss told me to add this field, but I don't have a clue why** is probably not a good idea—it will show up later on the status bar!

Create Meaningful Captions

As noted in “Naming Fields Incorrectly,” page 22, you should avoid using blanks in field names. This rule applies both to field names in tables and to alias field names in queries. (Remember, you create an alias field name in a query using an AS clause after an expression in the Select clause or by typing a name, a colon, and an expression on the Field line in the query design grid.) In the absence of a Caption property definition, Access uses the field name as the caption in attached labels on forms and reports or as column headings in Datasheet view.

When a field name is a single meaningful word, you probably don't need to define a Caption property. However, if the field name is a compound name (such as QtyOnHand) or follows a company naming standard for fields (naming standards often result in names meaningful to the developer but not the user), you should define a caption so that you don't accidentally display the nonfriendly field name anywhere in your application.

Understand the Interaction Between Format and Input Mask

Perhaps second only to the misunderstanding about the difference between how a Date/Time data type is stored and displayed (discussed earlier in this chapter) is the confusion about the difference between the Format property and the Input Mask property. The salient points are as follows:

- When a control or field has only a Format property, that property controls how Access displays the data when the control or field does not have the focus and the field has a value or has a default value on a new row. The format might not display all the data stored in the field or control—for example, a format might display only the year portion of a Date/Time field. When you move the focus to the field or control, Access removes the format and displays the actual data.
- When a control or field has only an Input Mask property, that property controls both how Access displays the data when the control or field does not have the focus and what

you can enter when you begin typing data in the control or field. If the field contained data before you defined the input mask or you insert data into the field without using the mask (perhaps with an insert query), the input mask might not display all the data.

- When you define an input mask for a text field, you can optionally instruct the mask to store any formatting characters in addition to the data you type. Access includes the formatting characters only when you update the field by typing in it with the mask active.
- When you define both a format and an input mask for a control or field and the format matches the input mask, Access displays the data using the format when the control or field does not have the focus, and it activates the input mask when you enter a control or field that already has data or a default value. If you tab into a control or field that does not have any data, the input mask appears as soon as you begin to type.

Notice that this all works nicely when the format and input mask are compatible—they display the same number of characters in the same pattern. But you should *never* define an input mask that doesn't match the Format property. The bad news is you can accidentally define an input mask that's not compatible in a Date/Time field even though you have defined no format. Here's what happens when Format and Input Mask properties are not compatible:

- The Format property takes precedence over the Input Mask property when the control or field contains data or a default value—whether the control or field has the focus or not.
- If you begin typing in a control or field that has data or a default value, the format disappears, but you will not see the input mask. However, you must enter characters that are compatible with the input mask that you cannot see. If you attempt to save a value that does not match the input mask, Access displays an error.
- If you begin typing in an empty control or field, the input mask appears and controls what you can enter. The Format property displays the result when you move the focus away from the control or field.
- The Date/Time data type is a special case. Even though you might not have defined a format, Access displays the data formatted with the General Date format. If your input mask does not match this format, the control or field acts as though the format and input mask are not compatible. The problem is the General Date format might display a date, a time, or both, depending on what is stored in the control or field. You might define an input mask to enter the date only, but if the field contains a time component, the field acts as though it has an incompatible format and input mask.

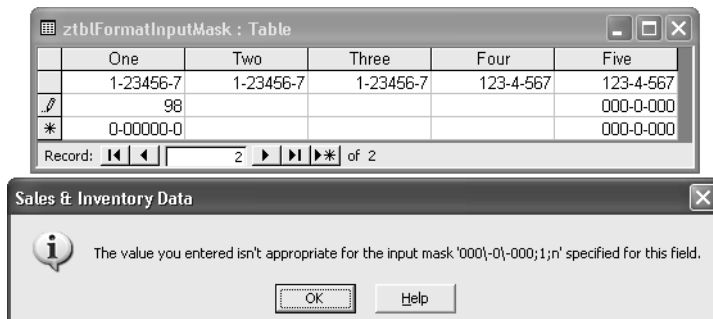
To help you understand this interaction more clearly, I created a small table in the data file for the Inventory sample application (InventoryData.mdb). The table, which is named

34 Part I: Designing Your Application

ztblFormatInputMask, has five long integer fields and one row containing the value 1234567 in all five fields. Here's how the fields are defined and what happened when I tried to enter data in a new row:

Field One

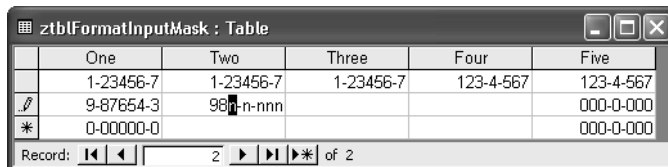
The Format property is set to 0-00000-0, the Input Mask property is set to 000\~0\~000;1;n (not compatible), and the Default Value property is 0.



This is the worst-case scenario. The input mask never appears because the field has a default value that the Format property displays. When I tried to tab away from this field without entering the number of digits required by the input mask, Access displayed the ugly message you see.

Field Two

The Format property is set to 0-00000-0, the Input Mask property is set to 000\~0\~000;1;n (not compatible), and there is no Default Value.



The input mask worked correctly when I first began to enter data because the field has no default value, but it won't work for fields that already have data. Notice that field One now displays the data using the format, not the input mask. Also, as you can see in the first row, Access formats any data that I enter in field Two using the format, not the input mask, when the field does not have the focus.

Field Three

The Format property is set to 0-00000-0, and there is no Default Value.

	One	Two	Three	Four	Five
	1-23456-7	1-23456-7	1-23456-7	123-4-567	123-4-567
	9-87654-3	9-87654-3	98		000-0-000
*	0-00000-0				000-0-000

This demonstrates a simple Format property. Notice that the format disappeared when I began typing, but the row above shows the field is formatted correctly.

Field Four

The Input Mask is set to 000\-\0\-\000;1;n, and there is no Default Value.

	One	Two	Three	Four	Five
	1-23456-7	1-23456-7	1-23456-7	123-4-567	123-4-567
	9-87654-3	9-87654-3	0-09876-5	98\-\n-nnn	000-0-000
*	0-00000-0				000-0-000

This demonstrates a simple Input Mask property. Notice that the input mask was activated as soon as I began typing, and the field in the row above displays the data formatted with the input mask.

Field Five

The Format property is set to 000-0-000, the Input Mask property is set to 000\-\0\-\000;1;n (compatible), and the Default Value is 0.

	One	Two	Three	Four	Five
	1-23456-7	1-23456-7	1-23456-7	123-4-567	123-4-567
	9-87654-3	9-87654-3	0-09876-5	987-6-543	98\-\n-nnn
*	0-00000-0				000-0-000

This demonstrates correct behavior because the format is compatible with the input mask. The field in the row above is actually being formatted by the Format property, and the field where I'm entering data shows the input mask activated.

The Bottom Line: Know Thy Data and Its Usage

As you have read through this chapter, you've encountered many "ifs, ands, and buts." This is intentional. If a "rule" truly doesn't apply to the way you need to use the data in your application, don't use it. But that assumes that you really know the application requirements inside and out.

In any database design exercise, it's absolutely critical that you thoroughly understand what your application needs to do and the business rules it needs to enforce. Look at the "classic" order entry application problem for a minute as an example. Will the price of a product ever change over time? If that's true (it probably isn't), then you don't need to save a copy of the price in the order detail rows. Does the company need to track product price changes over time? If so, you probably need a separate product price table that includes the price effective date, and you need to write code to look up the current price at the time an order is placed.

Is a product provided by only one vendor, or can you find the same or equivalent product from an alternate vendor? If an equivalent product is acceptable, do you need another table to track which products are equivalent to each other? If your business rules dictate that the second and subsequent purchase of the same product in one order receives a discount (buy one, get the second at half off), do you reflect the discount in a single order detail row, or must the salesperson enter the second purchase in another row at a discount?

As you create a database design, don't be afraid to ask a lot of questions or "think outside of the box." And don't be afraid to question *everything* your user tells you. Do you really need only one address for a customer? Can a registrar override the requirement for prerequisite courses to register a student for an advanced course? Will your business requirements ever change? I can answer the last question for you—the answer is YES!