1

Creating a Database for an Application

An application is more than just a database. Anybody with Access can create a database, but a database with a bunch of disconnected tables, queries, forms, and reports is not an application. An application consists of a database—or possibly several databases—containing normalized tables with appropriate relationships between them; queries that filter and sort data; forms to add and edit data; reports to display the data; and possibly PivotTables or PivotCharts to analyze the data, with all of these components connected into an efficiently functioning and coherent whole by *Visual Basic for Applications* (VBA) code. This chapter covers preparation for creating an application (getting and analyzing the information you need from the client), and creating tables to hold the application's data.

Most Access books give you lots of information about Access database tables (and other database objects), but don't necessarily tell you the stuff you really need to know: how to divide up the raw data you receive from a client into separate tables, how to decide what field type to use for each field in a table, and what relationships to set up between tables to create an efficient and well-integrated application. Through a series of developer-client Q&A sessions, I'll show you how to extract the information you need to create the right tables for your application and link them into appropriate relationships.

I have always found it easier to understand a process by watching somebody do it, as opposed to reading abstract technical information about it, so in this chapter, I will explain what I am doing as I walk you through the preparation for creating an application and then the creation of its tables. Succeeding chapters will deal with creating the application's forms, queries, and reports. Some technical information is necessary, of course, but it will be interspersed with demonstrations of what you need to do, and the explanations will tend to follow the actions, rather than precede them. Sometimes, if you can see how something is done correctly, that is all you need to know in order to do it right yourself, while an abstract technical explanation by itself is rarely adequate to teach you how to do something correctly.

There won't be lots of step-by-step walk-throughs illustrating how to create database objects in this book, or long lists of properties and other attributes. I am assuming that you already know how to create tables, forms, and other database objects (and if you need detailed information, you know how to get it from Help), and that instead, what you need is help in making decisions on what kind of data goes into which table, how the tables should be related, and what types of forms, queries, and reports are best for working with the data in your application.

Although code is crucial to binding an application into a coherent whole, there won't be any code in this chapter, because in Access, code runs from event procedures, and tables don't have event procedures. Before writing event procedures, we need to create tables to hold the data, and that is what this chapter covers.

Gathering Data

To start creating an Access application, you need two things: a clear idea of what tasks the application should perform and the output it should produce, and an adequate quantity of realistic data. Rather than just asking the client for a list of the tasks the application needs to perform, I usually ask a series of questions designed to elicit the required information. A typical Q&A session is presented in the next section of this chapter. However, if there is already a functioning database, printouts of its reports and screen shots of its forms can be helpful as an indication of what tasks are currently being done.

For best results, there is no substitute for large quantities of real data, such as the ebook data in the sample EBook Companion database used in Chapter 9, *Reworking an Existing Application*. But if you have a reasonable quantity of representative data and a client who is willing to answer your questions, that should be sufficient to set up tables with the correct fields. With this information, you can create tables with the necessary fields; set up relationships between them; and proceed to create the queries, forms, reports, and VBA code that will let you create an application that does what the client wants.

Curiously, I'm often asked to start working on an application for a client without any data at all. It may be difficult to convey this concept to a client, but it is important to get real-life data (either in electronic or paper form) in order to set up tables and fields correctly. If you (the developer) have to create dummy data to have something to work with when creating tables and other database objects, you will probably end up having to make changes—possibly major changes—to tables later on, and find that further changes are needed to other database components; so, it really helps to have a substantial amount of representative data to work with.

However, realistically there are cases where you won't be able to get data from the client. There are two cases where data isn't available: a brand-new business (or other enterprise) that doesn't have any data yet, and a business whose data is confidential. In these cases, you just have to do the best you can, creating dummy data after questioning the client about what data needs to be stored in the application's tables.

Once you have obtained the data in electronic or paper form, it's best to just use it as raw material to help you determine what fields you need when designing tables and other components, rather than as ready-made components to plug into your application. Looking at real data, you can see (for example) whether or not there are unique IDs for products. If there is a unique Product ID, that field should be the key field of the Products table; otherwise, you will need to create an AutoNumber field. If you see multiple addresses for customers or clients, you will need to create linked tables for address data, for purposes of normalization; if there is only one address per customer or client, address data can be stored directly in the Customers or Clients table. In most cases, even if you are given a database with Access tables, you will

need to make some modifications for purposes of normalization, and create a number of supporting tables as well as lookup tables to use as the row sources of comboboxes used to select values or records.

Figuring Out Business Tasks and Objects

When designing an application for a client, after obtaining a reasonable quantity of representative data, you need to discuss the processes to be modeled in the application—not just how they are done currently, but how they could be done better and more efficiently. For example, users may have been typing data into textboxes; if the data is limited in nature (for example, sales regions or phone number types) a combobox with a lookup table as its row source will ensure that users don't mistype an entry, which would cause problems when sorting or filtering data later on.

A client may give you piles of paper documentation, or descriptions of business processes, which again may or may not be helpful, depending on how well thought out these processes are. Often, real-world business practices develop bit by bit over the years, with new procedures not being integrated with older processes as well as they might be. When designing an application, it's a good idea to review the existing procedures and consider whether they should be streamlined for greater efficiency when setting up the database.

Don't just attempt to duplicate existing business processes in your database—at least not without examining them closely. Upon examination, you will often find that there are serious gaps in procedures that need to be remedied in the database. Just because users have been manually typing customer letters in Word and typing the customer address off the screen from a database record doesn't mean that you shouldn't generate Word letters automatically. (See Chapter 11, *Working with Word,* for information about generating Word letters from an Access database.)

You may also see that your application could do some tasks that aren't being done at all, but that would be very useful, such as generating email to clients, or analyzing data in PivotTables or PivotCharts. See Chapter 12, *Working with Outlook,* for information on sending email messages from Access. But the application first needs tables to store data, so the initial task is to set up the database's tables.

Determining Your Entities

The first task in setting up a database is determining what things it works with and how they work with each other. (The technical term often used in database literature is *entity,* but as far as I am concerned, *thing* works just as well.) If you are developing an application for a client, there may be an existing database. Depending on the skills of the person who created the database, this may be more of a problem than a helpful first step.

As an example of how to figure out the things your application needs to work with, following is a hypothetical example of a client who wants an application to manage his business, called the Toy Workshop. Let's start by asking the client some basic questions:

Q: What does the business do?

A: We sell toys.

We need a Toys table.

Q: Do you have an ID or product number for each toy?

A: Yes, a combination of letters and numbers.

We need a text ToyID field as a key field in tblToys.

Q: Do you make the toys or purchase them from vendors and resell them?

A: Both.

We need a table of materials used in manufacturing toys. We might need two tables for toys—one for toys purchased for resale and one for toys manufactured in-house. We need to determine whether the two types of toys are different enough to require different tables or whether they can be stored in one table, with different values in a few fields, and a Materials table, for toy-making materials.

Q: What are the differences between manufactured and purchased toys?

A: For purchased toys we need to record the vendor name, vendor product number, purchase price and purchase date; for manufactured toys we need to know how much of each component is used, the labor costs, and when they were manufactured.

Sounds like we could use a single Toys table, with a Yes/No Purchased field to indicate whether the toy is purchased or manufactured; that field could then be used to enable or disable various controls on forms. We also need a Vendors table, to use when selecting a value for the VendorName field, and a Materials table, for toy-making materials.

Q: Are raw materials purchased from different vendors than toys to resell, or could one vendor sell you both materials and finished toys?

A: Most of the vendors we use sell only finished toys; some sell only materials, and just a few sell both materials and finished toys.

Then all the vendors could be stored in one table, with Yes/No fields to indicate whether they sell finished toys, materials, or both.

Q: When you make the toys, is this done in your own workshop or factory, or contracted out?

A: Done in our workshop.

No need for a Contractors table.

Q: Do you have just one workshop, or several?

A: Just one.

Don't need a lookup table for workshops.

Q: Do you do anything else other than selling toys?

A: Yes, we also repair broken toys.

Need a Repairs table.

Q: Just the ones you sell, or others too?

A: Our own and other similar toys.

We can't just identify the repaired toys by ToyID; we'll need an AutoNumber field to uniquely identify toys made or purchased elsewhere that come in for repair.

Q: Are the repairs done in-house, or contracted out?

A: In-house only.

We need a table of employees, with a field to identify those who do the repairs.

Q: Do you send out catalogs or other promotional materials?

A: Yes.

We need a table of customers, and also a table of potential customers or leads.

Q: By mail, email, or both?

A: Both.

The Mailing List table(s) should have both the mailing address and email address.

Q: Do you sell toys in a store, by mail, or over the Internet?

A: From a factory store and by mail or phone. No Internet sales yet, but maybe in the future that will be added.

We need an Orders table, with a field for sale type. Customers should be selected from a Customers table, with a provision for entering a new customer on the fly when taking an order. Since mail or phone orders will require both a shipping and a billing address, we need a linked table of shipping addresses.

From these answers, we know that the application needs primary tables for the following things (these are the application's entities):

- ❑ Toys
- ❑ Categories
- ❑ Vendors
- ❑ Customers
- ❑ Shipping addresses
- ❑ Mailing list
- ❑ Materials
- ❑ Repairs
- ❑ Employees
- ❑ Orders

Additionally, a number of linked tables will be needed, to store data linked to records in the main tables, and some lookup tables will be needed to store data for selection from comboboxes, to ensure data accuracy.

Creating Tables for an Application

Now that you have some information from the client, you can start creating tables and setting up relationships between them, turning a mass of inchoate data into a set of normalized tables representing the

things (entities) the database works with. Using the list of tables obtained from the Q&A session with the client, let's start creating tables for the Toy Workshop application. First, though, a note on naming objects: applying a naming convention right from the start when creating a database will make it much easier to work with. I use the Leszynski Naming Convention (LNC), which is described more fully in Chapter 9, *Reworking an Existing Application*. For tables, the LNC tag is *tbl,* so all table names will start with that tag.

Table Creation Methods

To create a new table, click the New button in the Database window with the Tables object selected in the object bar. You have several choices in the New Table dialog, as shown in Figure 1.1.

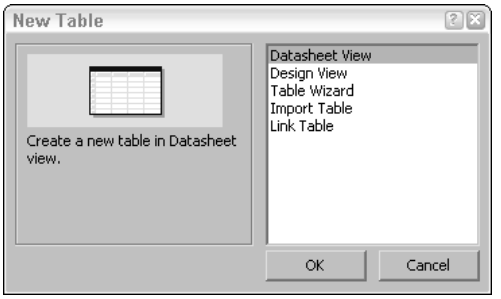

Figure 1.1

In many cases, it's best to just select the Design view choice and go ahead with creating the table fields, but some of the other choices are useful in certain cases. The primary consideration is whether your table is a standard table type (in which case the Table Wizard is a useful shortcut) or not (Design view is best). All the choices are discussed more fully in later sections:

- **Datasheet view.** This choice doesn't have much to offer. The new table opens in Datasheet view, and you can enter data into the first row. To name the fields in Datasheet view, you need to click several times on the Field*n* field name (until it is highlighted), then type the new name over it—much less convenient than just entering the field name on a new row in Design view. Access guesses at the field data type according to the data entered into the first row, not always accurately, so you will have to modify data types in Design view in any case. In the sample table shown in Figures 1.2 and 1.3, for example, when you switch to Design view you will see that the ToyID field is not identified as the key field, and the two price fields are Long Integer rather than Currency.

Figure 1.2

When creating a new table in Datasheet view, if you enter text into a field, a Text field is created; if you enter a number alone, a Long Integer field is created; if you enter a number with a dollar sign, a Currency field is created; and if you enter a recognizable date, a Date field is created. If you need any other data types (such as a Double numeric field, a Yes/No field, a Memo field, or an OLE Object field), you will have to create them in Design view.

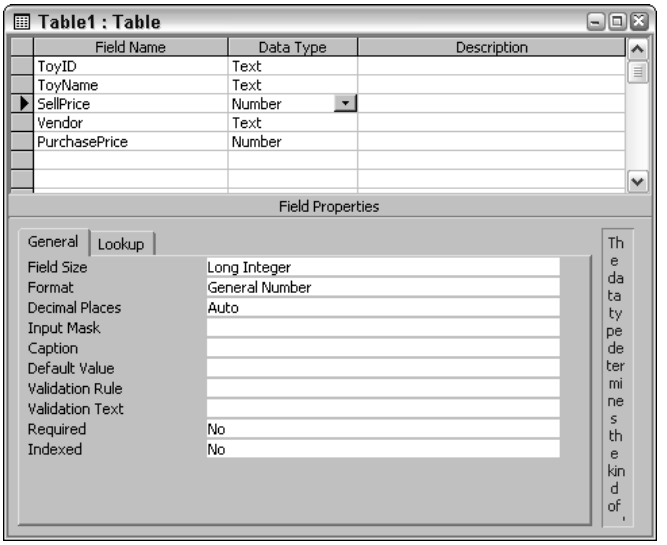

Figure 1.3

- **Design view.** The best choice for nonstandard tables you need to create from scratch. The table opens in Design view, letting you enter each field name on its own row and select the appropriate data type from the Data Type drop-down list (see Figure 1.4).

- **Table Wizard.** Useful as a shortcut when creating standard tables, such as a table of customer name and address data. However, these tables should be used with caution, because they are not always normalized. For example, the Contacts table shown in Figure 1.5 has a number of phone number fields (perhaps to match Outlook contacts), which (depending on the contact) could either provide too many or too few phone fields. With rare exceptions, phone and ID data should be stored in a linked table, which lets you enter exactly as many items as are needed for each contact.

- **ImportTable.** Lets you import data from an external source into an Access table. If you import outside data as a starter, you will need to examine the fields to make sure that they have the correct data type, and possibly break up the table into several linked, normalized tables.

- **LinkTable.** Links an Access table to data in another program, such as Excel. Linked tables aren't as useful as other tables because you can't modify their structure; use linked tables only when you need a quick view of current data maintained in an outside program.

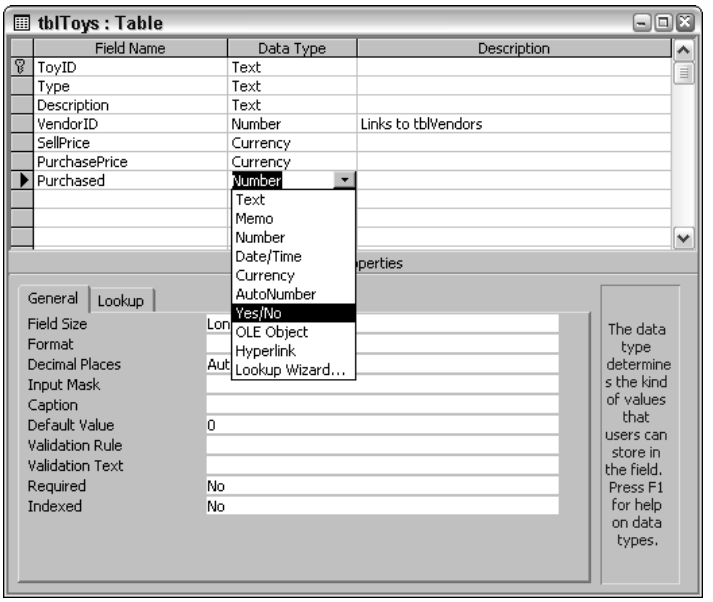

Figure 1.4

In the Database window, linked tables have an arrow to the left of the table name, and a distinctive icon for each data type, as shown in Figure 1.6, where you can see three linked tables—one a comma-delimited text file, one a dBASE file, and one an Excel worksheet. I use the tag *tcsv* for a linked comma-delimited text file, *tdbf* for a linked dBASE file, and *txls* for a linked Excel worksheet, so that I will know what type of linked file I am dealing with when I can't see the icons in the Database window.

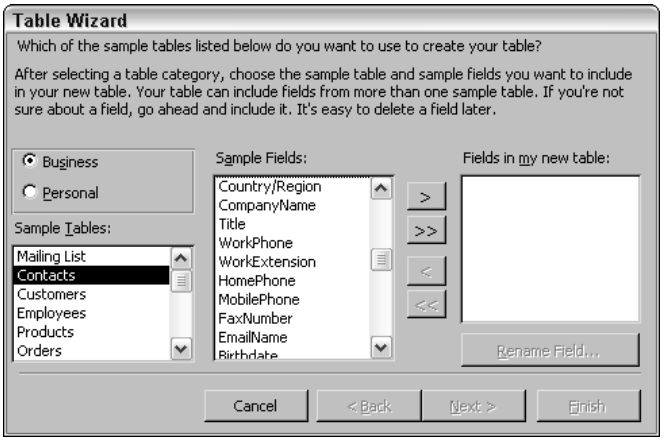

Figure 1.5

Figure 1.6

A *native* table is a table that contains data within Access; the great majority of tables you will work with in Access are native tables. When you create a table in Access, it is a native table, and when you import data from an outside program, the imported data is placed into a native table. In addition to native tables, you can also work with *linked* tables, which let you work with data in other programs, such as Excel or dBASE.

Creating the Tables

I'll start with tblToys, which is the database's main table, containing information about the toys sold (and in some cases, manufactured) by the client. Since the Table Wizard offers a Products table, let's start with that, and modify it as needed. Figure 1.7 shows the Products table in the Table Wizard; I selected most of the standard fields to get a head start on creating tblToys.

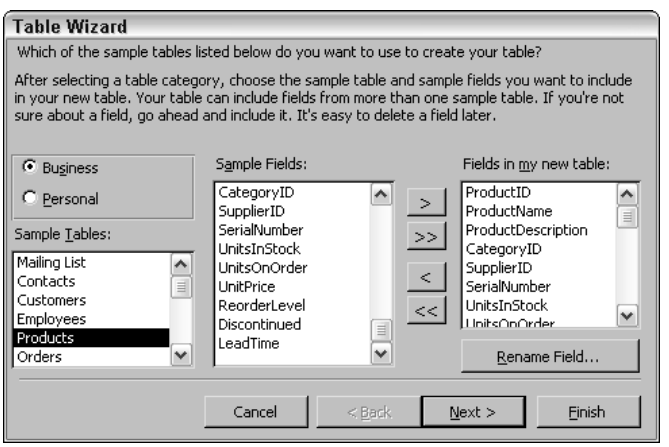

Figure 1.7

There is a button on the Table Wizard screen that lets you rename a field; you can either rename fields as desired in the wizard, or wait until the table opens in Design view and rename fields as needed there. After selecting and (optionally) renaming fields as desired, click the Next button to go to the next page of the wizard, where you give the table a name (tblToys in this case), and select the option to have Access set the primary key, or do it yourself; select *No, I'll set the primary key*, because you want to have control over the selection of the key field.

After clicking Next again, the wizard correctly assumes that the ToyID field should be the key field, and gives you three choices (shown in Figure 1.8). Select the third, because in this case the ToyID field contains a combination of letters and numbers. (If you need an AutoNumber ID, select the first option; for a numeric ID, the second option is appropriate.)

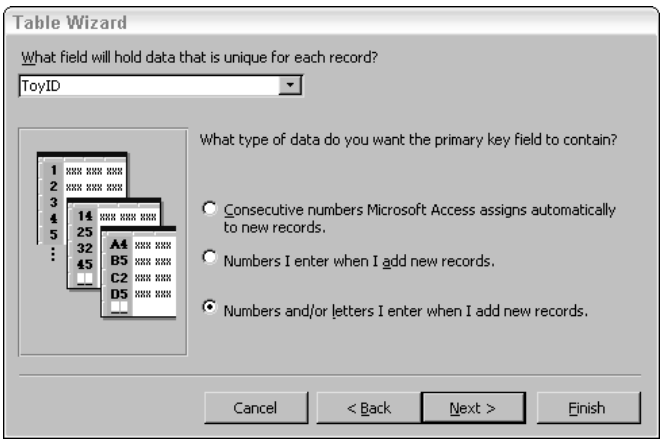

Figure 1.8

After clicking Next again, you are asked if you want to link the new table to any other tables in the database; because this is a new table, just click Next again. On the final screen, select *Modify the table design* to open the new table in Design view, where you can finalize its structure.

If you don't set table relationships in the Table Wizard, you can always set them later in the Relationships window—in fact, you may prefer to create all your relationships there, using its more intuitive visual interface.

The first step is to set up an input mask for the ToyID key field, to ensure that data entry into this field meets the client's specifications for this field. To create the input mask, you can either click the Build button to the right of the Input Mask property for the ToyID field to open the Input Mask Wizard, or just type in the input mask. Since the Input Mask Wizard doesn't have a standard selection of the appropriate type, we'll need to type it directly. The table below lists the characters you can use in input masks, to restrict data entry into the field.

Mask Character	Entries Allowed.
0	Required digit from 0 through 9; plus and minus signs not allowed.
9	Optional digit from 9 through 9, or space; plus and minus signs not allowed.
#	Optional digit or space; blanks converted to spaces; plus and minus signs allowed.
L	Required letter A through Z or a through z.
?	Optional letter A through Z or a through z.
A	Required letter or digit.
a	Optional letter or digit.
&	Required character or space.
C	Optional character or space.
. , : ; - /	Decimal placeholder and thousands, date, and time separators—the character used depends on the Regional settings in the Control Panel.
<	Converts following characters to lowercase.
>	Converts following characters to uppercase.
!	Causes the input mask to be displayed from right to left, instead of the standard left to right. In some versions of Office, this switch does not work correctly. See the Microsoft Knowledge Base (KB) article 209049, Input Mask Character (!) Does Not Work as Expected" for a discussion of the problem in Access 2000. KB articles can be viewed or downloaded from the Microsoft support Web site at `http://support.microsoft.com/`
\	Marks the next character as a literal character.
Password	Creates a password entry textbox—characters typed into the textbox are stored as entered, but displayed as asterisks, for security.

For the sample table's key field, the client says that the ToyID consists of two uppercase letters and three numbers, so we'll need the > character to make entered letters uppercase, then two L's and three zeroes entered into the InputMask property of this field:

```
>LL000
```

In addition to the standard fields from the Table Wizard, we'll need a few more fields to hold data related to manufactured toys. Figure 1.9 shows the table with the extra fields. There is another product ID field in the table, VendorProductID, but I won't put an input mask on this field, because vendors have their own ID formats.

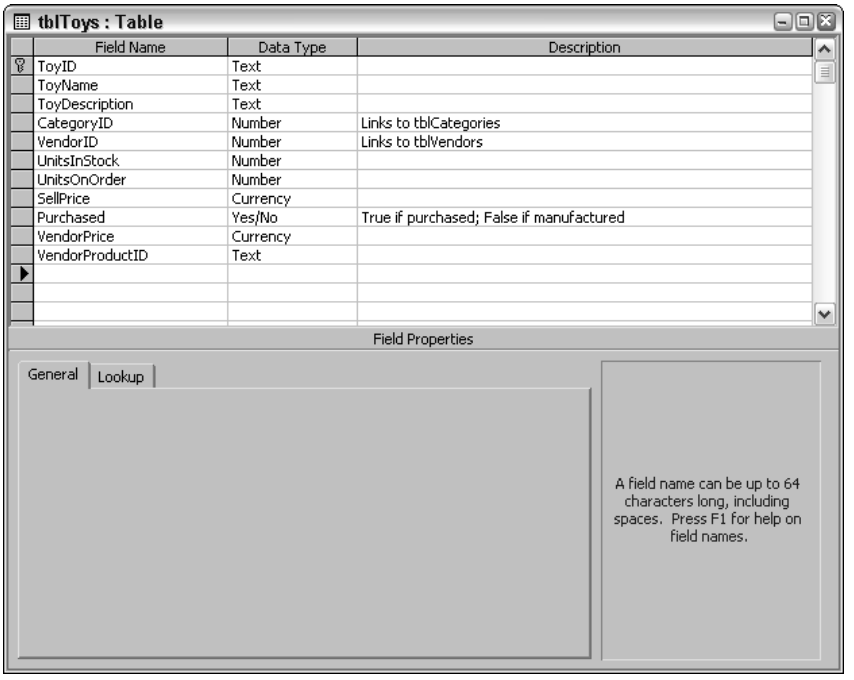

Figure 1.9

Information on materials isn't stored in this table, but in another table that will be created later.

Next, we need to create tblVendors and tblCategories, which will be linked to tblToys. After selecting the standard Categories table in the Table Wizard, click the Relationships button to set up a relationship with the CategoryID field in tblToys, as shown in Figure 1.10.

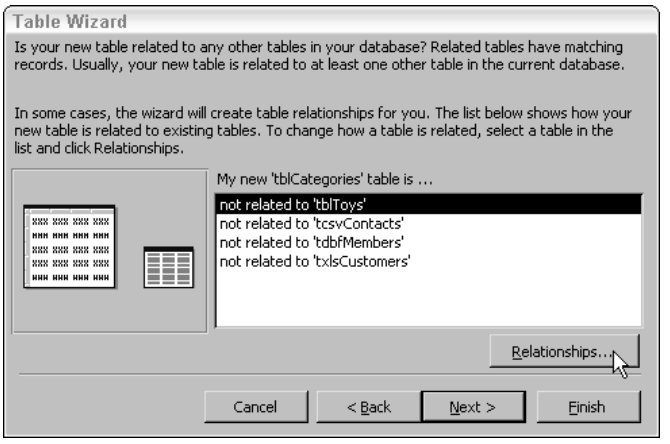

Figure 1.10

Select the middle choice on the next wizard screen (shown in Figure 1.11), because one toy category will be selected for many records in tblToys.

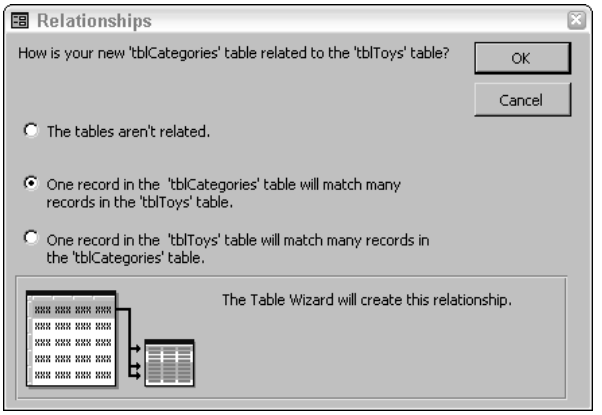

Figure 1.11

Now the wizard screen shows that tblCategories is related to tblToys (see Figure 1.12).

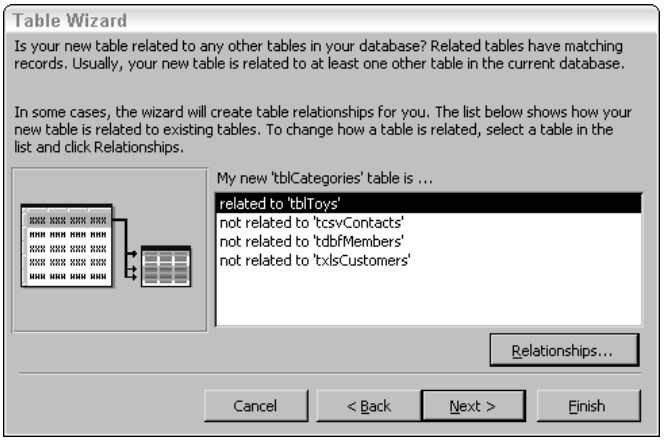

Figure 1.12

There is now a relationship between tblToys and tblCategories, which you can see in the Relationships window in Figure 1.13.

Note that although we selected *One record in the 'tblCategories' table will match many records in the 'tblToys' table,* in the Relationships page of the Table Wizard, the relationship between these two tables is not set up as a one-to-many relationship—although it should be one (another reason you may prefer to not create relationships in the wizard, and just set up all relationships in the Relationships window). See the Relationships section later in this chapter for information on modifying the relation types set up by the Table Wizard.

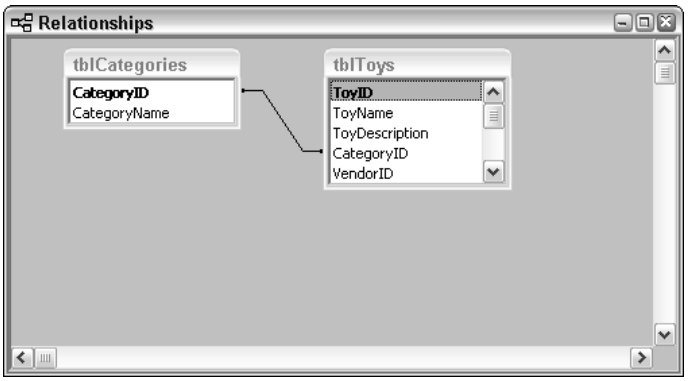

Figure 1.13

Continuing with table creation, tblVendors can be created using the Table Wizard, selecting the standard Suppliers table as the source, and linking it to tblToys on the VendorID field. The Table Wizard doesn't do all you need to link these tables, though. If you want to set up tblCategories or tblVendors as the row source of a lookup field, so the value can be selected from the linked table, you will need to do this yourself (see the lookup field information in the "Table Field Data Types" section in this chapter).

Table Field Data Types

When creating table fields, it's important to select the correct data type for each field, so that you can enter data into the field (you can't enter text into a numeric field!), and also use the data for sorting and filtering as needed. The table below lists the field data types available in Access tables, with comments. The primary data type is what you see in the drop-down Data Type list when creating or editing a field; some fields (Numeric and AutoNumber) also have subtypes, which are selected from the Field Size property in the field properties sheet.

Primary Data Type	Subtypes	Description	Comments
Text		Text data up to 255 characters in length.	Use for text data, and numbers (such as IDs) that are not used for calculations.
Memo		Blocks of text up to 65,535 characters in length.	Use for long text; only limited sorting is available on this field (just the first 255 characters are used).
Number	Byte	Whole numbers from 0 through 255.	Small numbers with no decimal points.
	Integer	Whole numbers from –32,768 through 32,767.	Medium-sized numbers with no decimal points.

Primary Data Type	Subtypes	Description	Comments
	Long Integer	Whole numbers from –2,147,483,648 through 2,147,483,647.	Long numbers with no decimal points. This is the default value. Matches AutoNumber fields when linking tables.
	Single	Numbers from –3.402823E38 through –1.401298E–45 for negative values and from 1.401298E–45 to 3.402823E38 for positive values.	Accurate to 7 decimal points.
	Double	Numbers from –1.79769313486231E308 through –4.94065645841247E–324 for negative values and from 4.94065645841247E–324 through 1.79769313486231E308 for positive values.	Accurate to 15 decimal points.
	Replication ID	Globally unique identifier (GUID), a 16-byte field used as a unique identifier for database replication.	Only used in replicated databases.
	Decimal	Numbers from –10^28–1 through 10^28–1.	Accurate to 28 decimal points.
Date/Time		Dates or times.	Always store dates in a Date/Time field, so you can do date and time calculations on them.
Currency		Currency values, or numbers that need great accuracy in calculations.	Use a Currency field to prevent rounding off during calculations. A Currency field is accurate to 15 digits to the left of the decimal point and 4 digits to the right.
AutoNumber	Long Integer	Incrementing sequential numbers used as unique record IDs.	Same data type as Long Integer for linking purposes. There may be gaps in the numbering sequence, if records are created and later deleted.
	Replication ID	Random numbers used as unique record IDs.	This is a very long and strange looking string.
Yes/No		Data that is either True or False.	Null values are not allowed.

Table continued on following page

Primary Data Type	Subtypes	Description	Comments
OLE Object		Documents created in programs that support OLE (such as Word or Excel).	You can't see the object in the table; use a form or report control to display it. Can't sort or index on this field.
Hyperlink		URLs or UNC paths.	You can click on a value in this field to open a Web site (if it is a valid link).
Lookup Wizard		Not a separate field type, but a wizard that lets you select a table or value list for selecting a value for a field.	Once a field is set up as a lookup field, you won't see the value stored in the field in table Datasheet view.

Whether they are created by the Table Wizard or manually in Design view, I prefer not to use lookup fields in tables, but instead select the lookup table in a combobox or listbox's RowSource property on a form. The reason for this is that if you set up a field (such as VendorID) as a lookup field, then you won't see the VendorID when you look at the table in Datasheet view—just the vendor name from the lookup table. If you need to see the actual VendorID, you will have to convert the field back to a standard field. Also, a lookup field will always be placed on a form as a combobox, while you may prefer to have the field displayed in a textbox, for example on a read-only form.

The tblVendors table as created by the Table Wizard is shown in Figure 1.14.

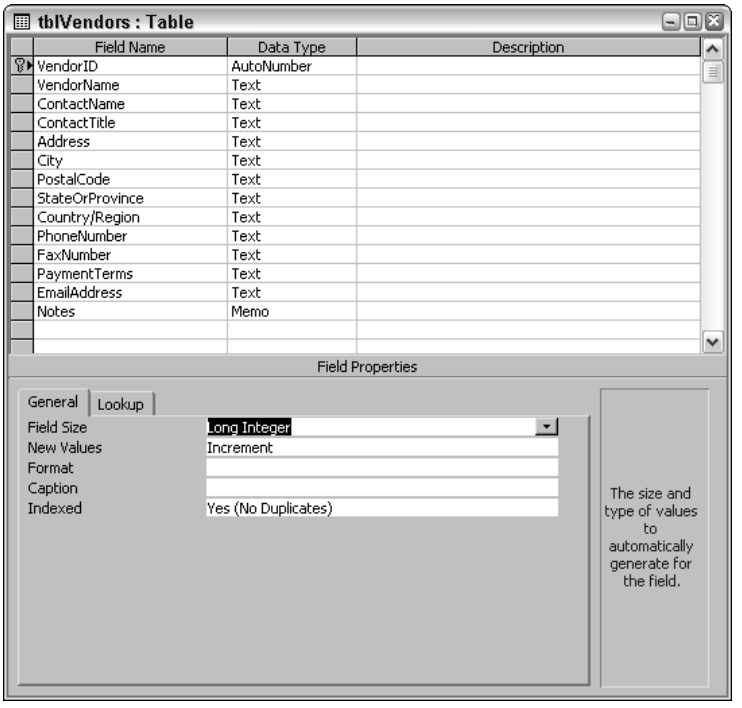

Figure 1.14

You can use the F6 function key as a shortcut for moving between a field in table Design view, and its properties sheet. This hot key is especially handy when setting the FieldSize property of numeric fields.

Because the client said that vendors could sell either finished toys or raw materials, the table needs two Yes/No fields to indicate whether the vendor sells finished toys, materials, or both, so I added these fields, setting the default value of the SellsToys field to True (since the client said that the majority of his vendors sell toys), and the default value of the SellsMaterials field to False.

Although Access lets you use spaces (and most punctuation marks) in field names—note the slash in the Country/Region field in tblVendors—I prefer not to use spaces or punctuation marks other than underscores in field names, to prevent problems when referencing fields in code and SQL statements, or when exporting table data to other applications that may not support spaces or punctuation marks.

As is often the case, on examining the table I realized that I needed to ask the client some more questions. It is rare to find all of the information you need to create tables at one time, right at the beginning. You will need to confer with your client from time to time while creating the application, asking more specific questions in order to refine table structure as needed. The question here is whether the vendors are all in the United States, or if there are some non-U.S. ones. If all the vendors are in the United States, we can eliminate the Country/Region field, and put input masks for U.S. state abbreviations or zip codes on the PostalCode and StateOrProvince fields; otherwise, the CountryRegion field is needed (I removed the slash in the field name), and we have to either leave the PostalCode and StateOrProvince fields without input masks, or take care of formatting by swapping input masks on a form or running event procedures to check that the correct data is entered into these fields.

Q: Are the vendors are all in the United States, or are there some non-U.S. ones?

A: There are some vendors outside of the United States.

Leave the table's fields as they are.

Now some more questions arise:

Q: Do you need just one phone number and one fax number, or could vendors have more than one phone number? Also, is one email address enough?

A: Some vendors have cell phone numbers too, and multiple email addresses.

We need to remove the phone and email fields from tblVendors, and create separate linked tables to hold this data.

Q: Do you need to send Word letters to vendor contacts?

A: No, the name is just so we know whom to ask for when we call the vendor.

Then we don't need to break up the contact name into its components, as would be required if we were going to create letters to them.

Figure 1.15 shows the final tblVendors.

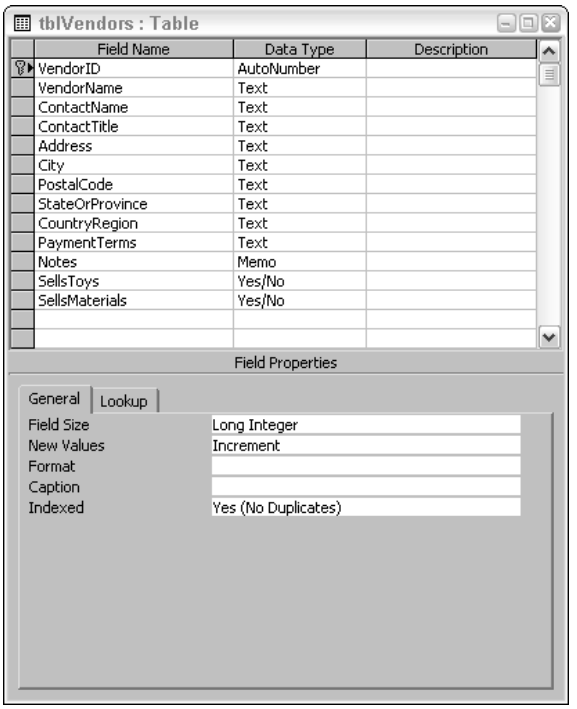

Figure 1.15

The linked tables require only the VendorID field and fields for (respectively) phone descriptions and numbers, and email addresses; the VendorID field in the linked tables is a Long Integer, to match the AutoNumber VendorID in tblVendors. When saving the new tables, don't create a key field; VendorID is a foreign key in tblVendorPhones and tblVendorEMails. (See the "Relationships" section later in this chapter for a definition of *foreign key*.) Since the vendors could be outside the United States, there is no need to create an input mask for the VendorPhone field. Figure 1.16 shows the two tables of vendor phone and email data; they will be linked to tblVendors in the Relationships window later in this chapter. tblVendorPhones has a phone number field and another field for the description (work, home, fax, and so forth), which allows you to enter as many different phone numbers as are needed for each vendor, each with its own description.

Continuing with table creation for the Toy Workshop application, tblCustomers can be created using the Table Wizard Customers table template, using all the fields to start with (just changing CompanyOrDepartment to Department), and setting up no relationships. The starter tblCustomers is shown in Figure 1.17.

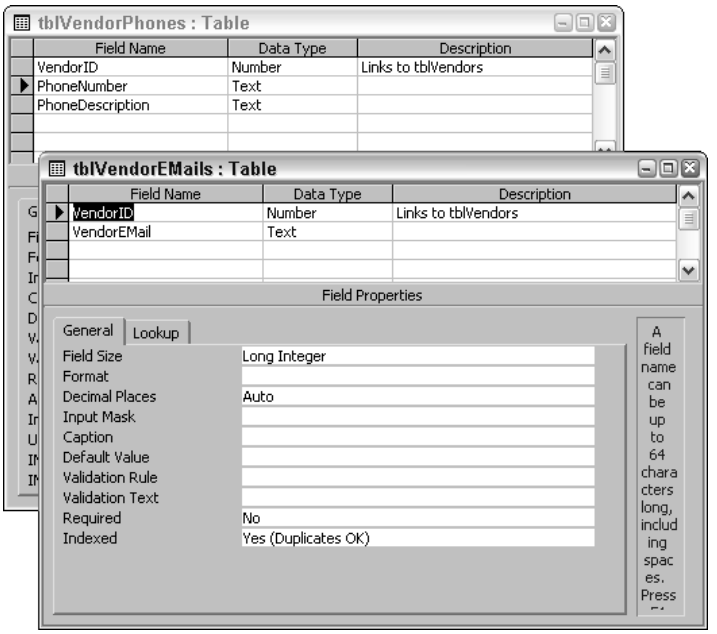

Figure 1.16

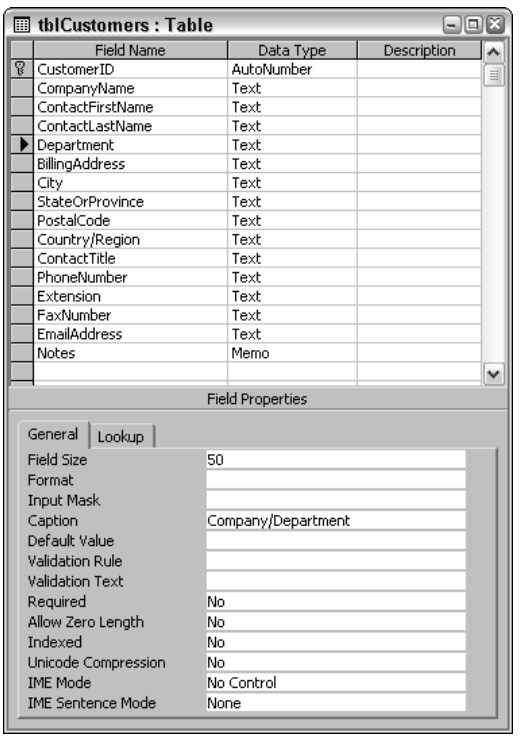

Figure 1.17

As with tblVendors, we have a few questions for the client:

Q: Do you need just one phone number and one fax number for customers, or could customers have more phone numbers? Also, is one email address enough?

A: Some customers have cell phone numbers too, and multiple email addresses. Come to think of it, some of them have Web sites, too.

We need to remove the phone and email fields from tblCustomers, and create separate linked tables to hold this data. And we need to add a WebSite Hyperlink field.

Q: Are the customers all in the United States, or do you have foreign customers too?

A: The customers are all in the United States.

We can remove the Country/Region field, and put appropriate input masks on the StateOrProvince and PostalCode fields.

The finished tblCustomers is shown in Figure 1.18.

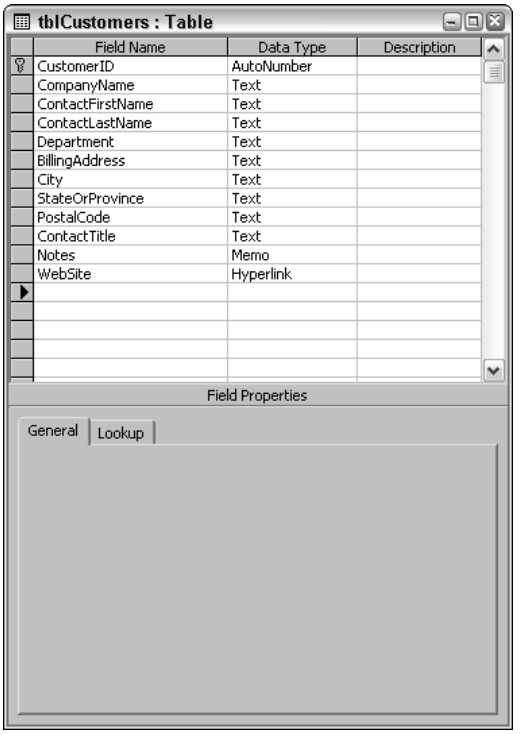

Figure 1.18

tblCustomerPhones and tblCustomerEmails are similar to tblVendorPhones and tblVendorEmails. We also need another linked table, to hold shipping addresses (this was determined in the initial Q&A

session). The billing address can be stored directly in tblCustomers, because there is only one billing address per customer, although there could be multiple shipping addresses. tblShippingAddresses has an AutoNumber ShipAddressID field, the linking CustomerID field, an address identifier field (for selecting a shipping address from a combobox on a form), and a set of address fields. Although the address fields could have the same names as the corresponding fields in tblCustomers, I like to prefix their names with "Shipping" or "Ship" so that if the billing and shipping addresses are combined in a query, we won't need to use the table name prefix to distinguish between them. tblShippingAddresses is shown in Figure 1.19.

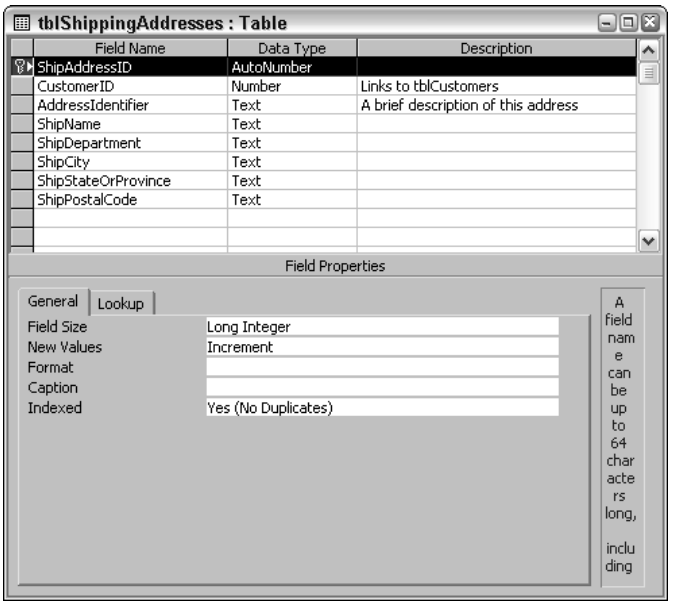

Figure 1.19

The next table is tblMailingList, which is created from the Mailing List template in the Table Wizard, omitting all fields except name, address, and email fields, plus DateUpdated and Notes. The tblMailingList table is shown in Figure 1.20.

On examining the initial version of this table, it occurs to me that the mailing list could contain several persons at the same company, so the company information should be broken out into a separate table, linked by CompanyID. However, some people on the mailing list might not be affiliated with companies, so we'll leave the address fields in the table, for entering personal address data, and add a CompanyID field to link to a separate tblMailingListCompanies table, for records that need it. The modified tblMailingList and tblMailingListCompanies tables are shown in Figure 1.21. When a mailing list record is entered on a form, the tblMailingList address fields will be enabled only if no company is selected for the CompanyID field; if a company is selected, its address will be used for mailings to that person.

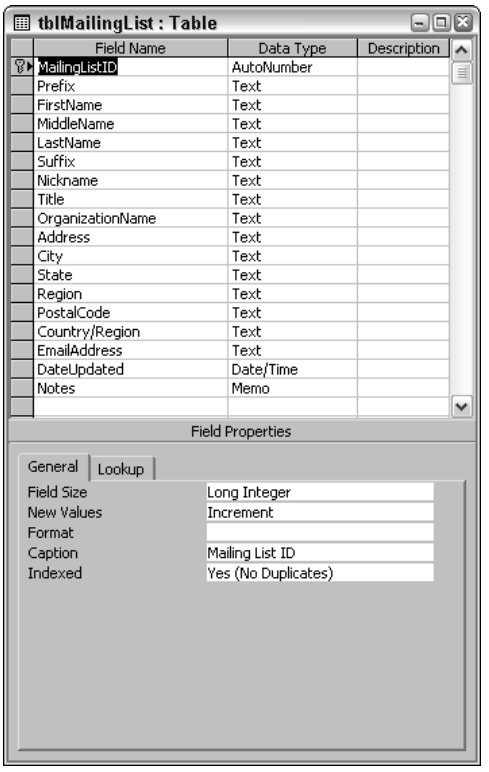

Figure 1.20

Customers will presumably get mailings too, but this doesn't mean that the whole mailing list needs to be in the tblCustomers table—we can combine data from tblCustomers and tblMailingList with a union query, when sending out mailings (See Chapter 4, *Sorting and Filtering Data with Queries,* for information on union queries.)

For date fields, I recommend selecting a date format that will display four digits for years, to avoid twentieth century/twenty-first century confusion. For an individual field, you can either select one of the standard formats from the Format property of a date field, or enter a format directly, such as m/d/yyyy. See the *Format Property—Date/Time Data Type* Help topic for full information on date and time formatting (you can locate this Help topic by entering *date format* in the Answer Box or Answer Wizard). Additionally, you can turn on 4-digit date formatting globally, overriding the Format property of fields and controls, by opening the Options dialog box from the Tools menu to the General page, and checking one of the checkboxes in the *Use four-digit year formatting* section, as shown in Figure 1.22. While you are on this page, take the opportunity to turn off Name AutoCorrect, which is nothing but trouble, because it doesn't make all the changes needed and sometimes makes changes when it shouldn't. Chapter 9, *Reworking an Existing Application*, lists better ways to rename database objects, using my LNC Rename add-in.

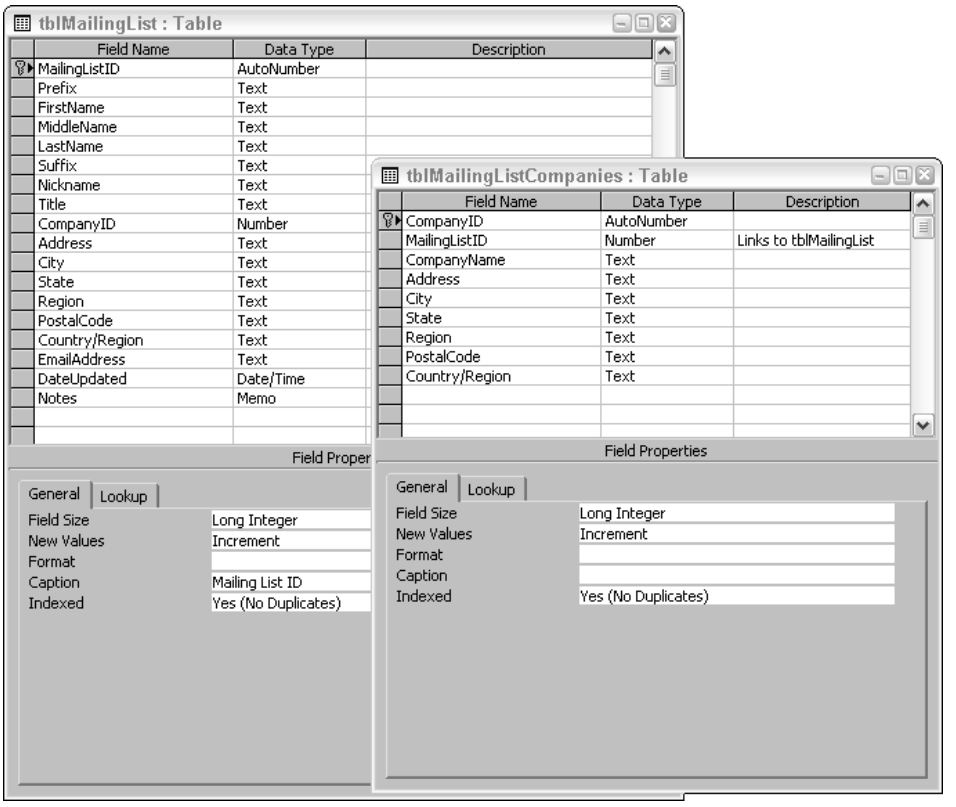

Figure 1.21

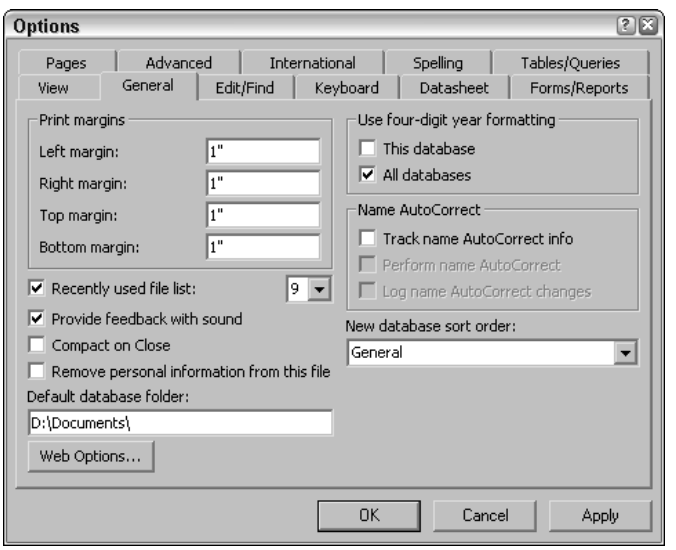

Figure 1.22

The next table to create is tblMaterials, which lists the materials used to make toys. The Table Wizard Products template is a good starter, omitting the fields that aren't needed and changing a few field names. The table is related to tblVendors on VendorID. tblMaterials is shown in Figure 1.23.

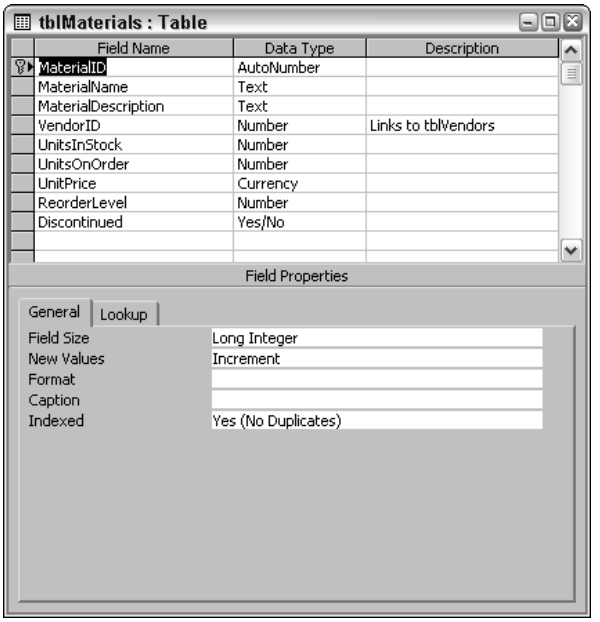

Figure 1.23

Since a material could be used for many toys, and a toy could use many materials, we need a many-to-many relationship between tblToys and tblMaterials; this is done by means of a linking table containing just the key fields; this linking table (tblToyMaterials) is shown in Figure 1.24.

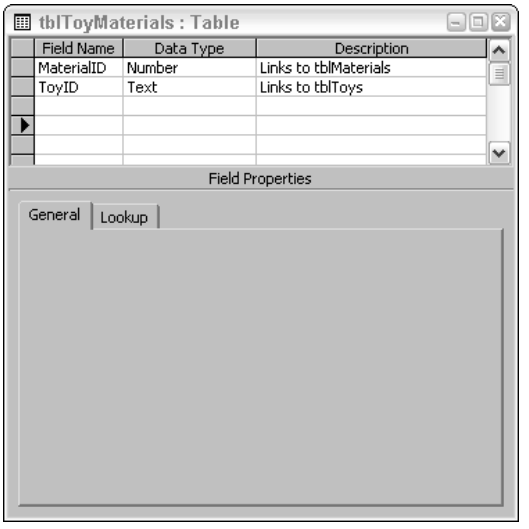

Figure 1.24

There is no suitable table template for tblRepairs, so I created this table directly in Design view, with just a few fields, as shown in Figure 1.25.

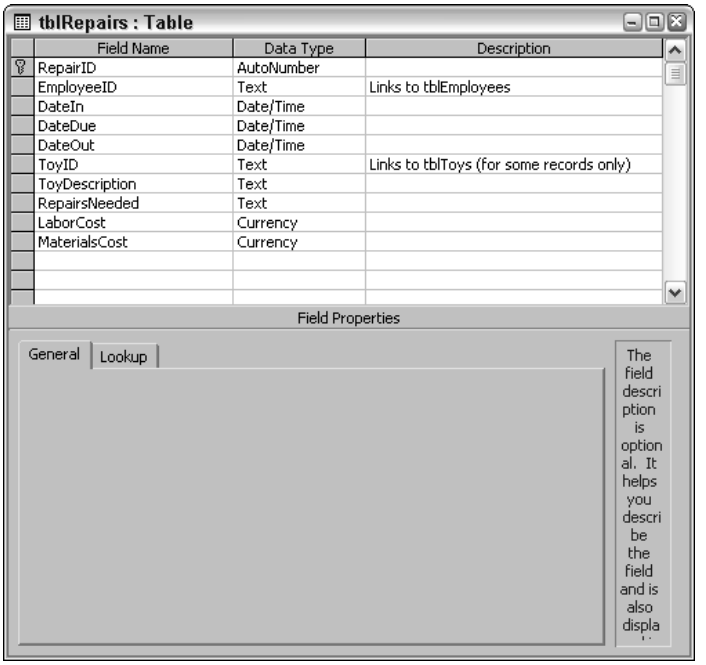

Figure 1.25

Following are some names you shouldn't use for fields: Name, Date, Month, Year, Value, Number, Sub. In general, any word that is a built-in Access function, property, or key word should be avoided, because it is highly likely to cause problems in VBA code and elsewhere. Just add another word to the field name (CustomerName, OrderDate), and you can avoid these problems.

Because repairs also use materials, we need a linked table, tblRepairMaterials, which lists the materials used to do repairs, and the quantity of each material. This table is shown in Figure 1.26.

The tblEmployees table is based on the default Employees table in the Table Wizard, with some unnecessary fields deleted. The client's company uses a numeric Employee ID, but since employees in this company already have IDs, we can't use an AutoNumber field; instead, EmployeeID is a text field, and it will be filled with existing employee numbers, with an incrementing number for new employees created by a procedure run from a form. tblEmployees is shown in Figure 1.27.

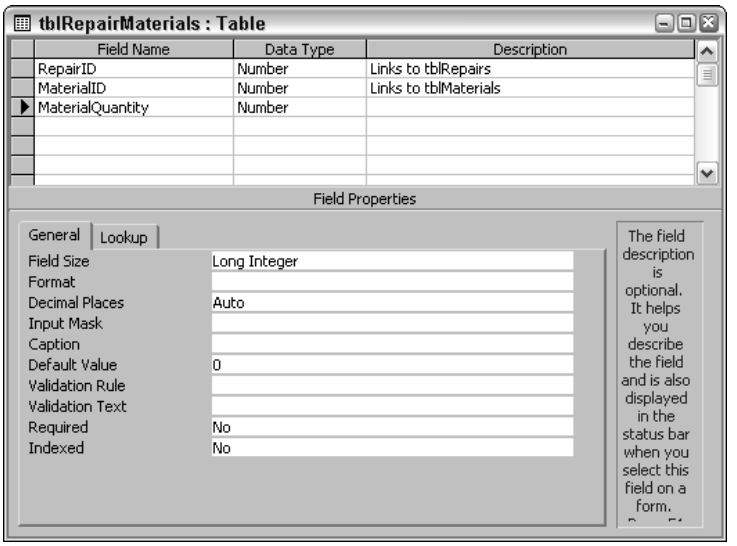

Figure 1.26

The last main table is tblOrders; it is also based on a template (Orders) in the Table Wizard, skipping the shipping address fields, and replacing them with a field (ShipAddressID) that links to tblShippingAddresses. We also need to add ToyID, to identify the toy that was purchased, and ToyQuantity (a curious omission from the Orders table template). tblOrders is shown in Figure 1.29.

The SupervisorID field takes an EmployeeID value that will be picked from a combobox on a form.

Some employee information should remain confidential, so the Social Security number (SSN) and salary are stored in a separate table, tblEmployeesConfidential, which is shown in Figure 1.28. (The input mask on the SSN field is one of the standard input masks, selected from the Input Mask Wizard.) Placing this information in a separate table lets you restrict its use to certain employees, using Access object-level permissions in a secured database. Even if you don't want to secure your database, there is a certain measure of confidentiality in just placing the information in another table so that it isn't visible when doing routine work on the main Employees form.

One default field in this table (ShippingMethodID) requires a lookup table of shipping methods; I created this table manually, and tblShippingMethods is shown in Figure 1.30.

Using an AutoNumber ShippingMethodID lets you select the shipper from an option group on a form (Access option group buttons have Integer values), and the selected value links to the shipper name in tblShippingMethods.

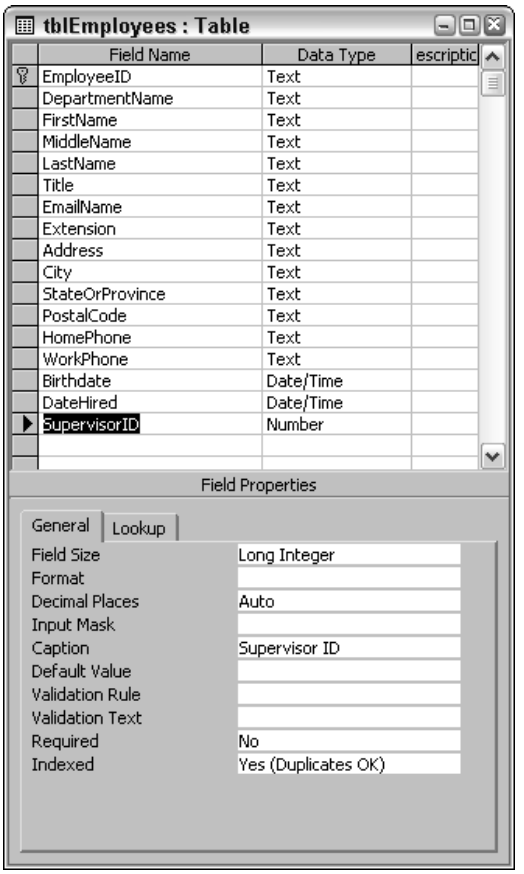

Figure 1.27

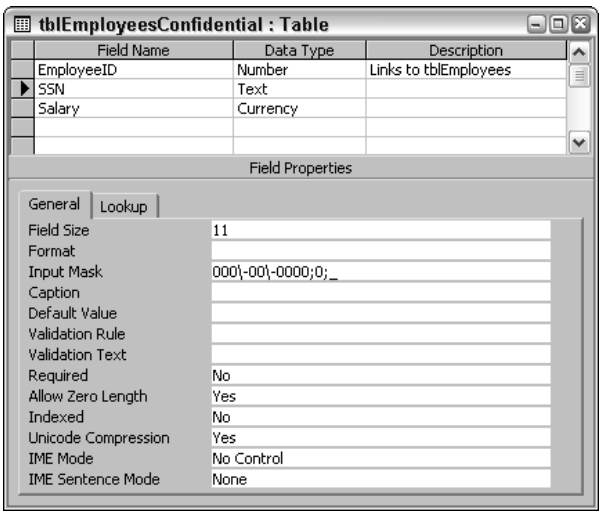

Figure 1.28

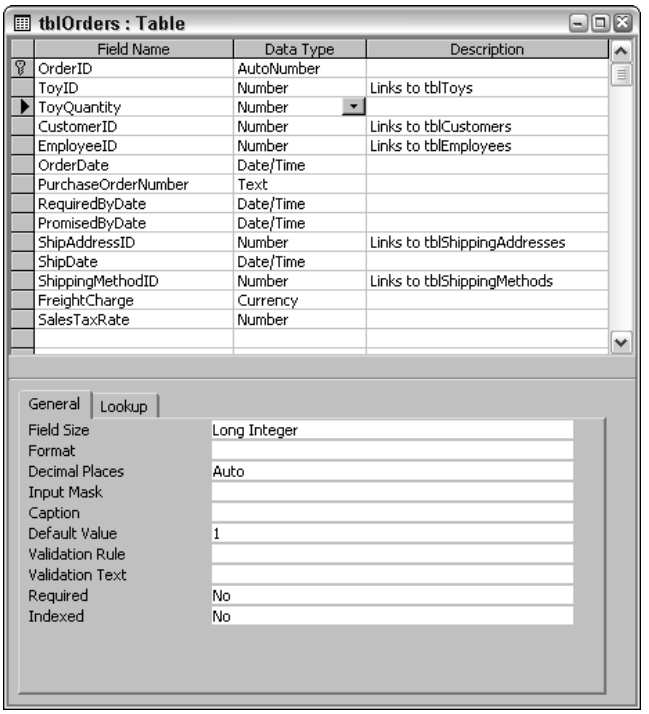

Figure 1.29

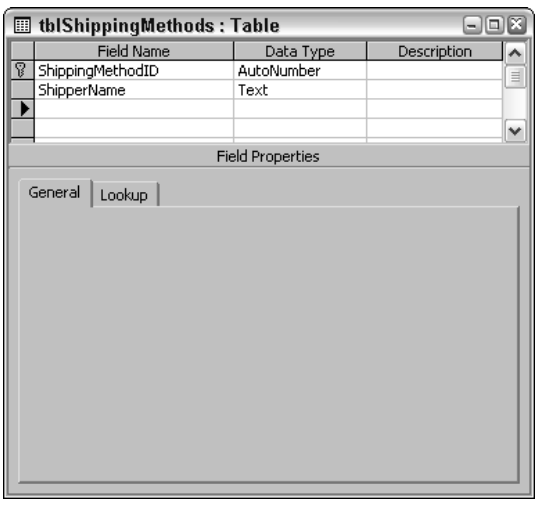

Figure 1.30

Normalization

Up to this point in the chapter, we have been *normalizing* tables—though without using that term. Database normalization can be (and often is) discussed in a very complex and opaque manner, bristling with technical terms, but this isn't necessary. When designing Access databases, normalization boils down to eliminating duplication of data in different tables, and using key fields to link tables so you can get the data you need from other tables through the links. There are five levels of database normalization (*first normal form* through the *fifth normal form*); generally, only the first three are used in Access databases. I define the five normal forms below, first in technospeak, then in regular English.

First Normal Form: Eliminate Repeating Groups

This means that you shouldn't have multiple fields for the same type of information in a table, such as multiple phone numbers or addresses for a customer. In some cases (generally only in tables imported from flat-file databases) the repeating data may be in a single field, such as a list of graduate degrees for faculty members, separated by commas. The problem with putting separate bits of information into a single field is obvious: If you wanted to create a query to view all the faculty with Ph.Ds, this would be a difficult task, requiring the creation of complex expressions to extract the different degrees for each faculty member, and you would probably not get all the records you need, because of differences in punctuation when the data was entered.

Several of the table templates in the Table Wizard violate the first normal form, such as the Contacts table with its multiple phone numbers. Instead of keeping multiple phone fields in tblVendors and tblCustomers, I created separate tables to hold phone numbers and email addresses for vendors and customers: tblCustomerPhones, tblCustomerEmails, tblVendorPhones, and tblVendorEmails. Breaking out this information into separate tables serves two practical purposes: It guarantees that you will always be able to enter another phone number or email address for a client (if you have just phone and fax fields, how are you going to enter the customer's cell phone number?), and it also makes it easier to use the information elsewhere in the database—you can pick up the phone numbers belonging to a customer by linking to tblCustomerPhones by CustomerID, instead of having to reference each of a set of named phone fields separately.

Repeating data in a single field (as in the faculty degrees example mentioned previously) should also be broken out into a separate table, both for accuracy of data entry (users should select degrees from a lookup table, rather than typing them into a field), and to allow entry of as many degrees as are needed for a faculty member.

Second Normal Form: Eliminate Redundant Data

There are two ways that redundant data can get into a database: One is by entering the same data into different records of a table. This could happen if you use the Table Wizard's Orders table template, with its address fields, and enter several orders from the same customer. If you enter that customer's shipping address into three different records, that is duplicate data. I avoided this situation by breaking out shipping address data into its own table, tblShippingAddresses, and placing a ShipAddressID field in tblOrders. This field links to the key field of the same name in tblShippingAddresses, which avoids the need to enter the same data into many records, and also guarantees that if there is a change in the shipping address, it needs to be made only once, not in every order record using that address.

You can also have redundant data when the same information is entered into two different tables. For example, if you have a Customers table and an Orders table, you should not put customer billing address fields in both tables. Either place the billing address fields in a separate table, linked one-to-one with tblCustomers by CustomerID, or place them in the Customers table and remove them from the Orders table. Shipping address fields should also not be duplicated in two tables; in this case, they should definitely be moved to a separate table (as I do in the sample database), because there can be multiple shipping addresses per customer.

In some cases, for recordkeeping purposes, it might be desirable to keep a record of the shipping address used at the time an order is shipped—even if that address changes later on. In that case, the shipping address fields could be retained in tblOrders, along with the ShipAddressID field, and when a shipping address is selected on a form, data from the selected shipping address could be pulled from tblShippingAddresses and written to the shipping address fields in tblOrders. This method eliminates the need to type the shipping address into every record, but preserves the shipping address data for each order even if the customer's shipping address is changed later on.

Third Normal Form: Eliminate Columns Not Dependent on Key

This means that any fields that don't belong to the record should be moved into a separate table. For example, the initial version of tblMailingList, made from the Table Wizard Mailing List table template, contains both information about the person receiving the mailings (name information, title, and so forth) and information about the company (company name and address). Because the company information doesn't belong to the person, I created a separate tblMailingListCompanies table linked by a CompanyID field to store mailing list company data. However, I left the address fields in tblMailingList, so they could be used for personal addresses for persons on the mailing list who are using their own addresses rather than company addresses.

Fourth Normal Form: Isolate Independent Multiple Relationships

In a database with many-to-many relationships, don't add irrelevant fields to the linking table that connects the two "many" tables. In a student records database, for example, using a many-to-many relationship between Students and Classes, with a linking table tblStudentClasses, you might have a Semester and Year field, indicating that a particular student took a particular class in a specific semester and year. That would be appropriate, but if you were to add a phone number field to the linking table, that would violate the fourth normal form because that field doesn't belong to the combined student-class record, but to the student record, so it should be placed in the Student table.

In the Toy Workshop sample database, there is a many-to-many relationship between Toys and Materials. For example, a toy can use multiple materials, and a material can be used for multiple toys. tblToyMaterials is the linking table for this many-to-many relationship. As is typical of such tables, it contains only the two foreign key fields that link to the key fields in the two "many" tables. If you add any extra fields other than the two key fields to such a table, they should be related to the combination of the two linked records, to avoid violation of the fourth normal form.

It is unlikely that you will have to worry about violating the fourth normal form because (unlike the first through third normal forms) it isn't likely that you'll be inclined to set up tables that violate it, or even have to rework tables that violate this form.

Fifth Normal Form: Isolate Semantically Related Multiple Relationships

Violation of this normal form requires a complex and unlikely scenario, and frankly there is a minimal chance that you will ever have to worry about it. In some circumstances, this form requires the separation of even related fields into a separate table. For example, in a many-to-many Students-Classes relationship, although semester and year information could appropriately be added to the linking tblStudentsClasses table, in some cases it would be preferable to maintain that information in a separate table, with information linking classes to specific Semester-Year combinations.

Setting Up Relationships

The Table Wizard gives you a start at setting up relationships between tables, but it doesn't do all the work. Even though you specify that a record in one table can match many records in another table, the relationship is not set up as a one-to-many relationship; you need to do this manually, in the Relationships window. I'll describe the three types of relationships you can create in an Access database, and show you how to set them up in the Relationships window.

Let's start with some definitions of terms used in creating relationships between tables:

- **Primary key.** A field (or, less commonly, a set of fields) with a different value (or value combination) for each record in a table. The key field must be unique and can't be Null.
- **Foreign key.** A nonunique field in a table that links to the primary key field in another table. In a one-to-many relationships, the primary key is in the "one" table, and the foreign key in the "many" table.
- **Cascading update.** When referential integrity is enforced, if you change the primary key value in a record in the primary table (for example, EmployeeID in tblEmployees), that value will be changed to match in all the matching records in any related tables. This is generally a good idea.
- **Cascading delete.** When referential integrity is enforced, if you delete a field in a primary table, all matching records in any related tables are also deleted. This is dangerous, and generally should be avoided.
- **Inner join.** There must be a matching value in the linked fields of both tables. With an inner join between tblCustomers and tblOrders, for example, you will see only records for customers with orders.
- **Left outer join.** All records from the left side of the `LEFT JOIN` operation in a query's SQL statement are included in the results, even if there are no matching records in the other table. A left outer join between tblCustomers and tblOrders includes all the Customer records, even those with no orders.
- **Right outer join.** All records from the right side of the `RIGHT JOIN` operation in a query's SQL statement are included in the results, even if there are no matching records in the other table. A right outer join between tblMailingListCompanies and tblMailingList includes all the tblMailingList records, even those with no company selected (they won't have matching records in tblMailingListCompanies).

- **Referential integrity.** A set of rules that ensures that relationships between records in linked tables are valid and that related data isn't changed or deleted inappropriately. Setting referential integrity on a link between tblCustomers and tblOrders (on the CustomerID field) would ensure, for example, that you can't enter a new order without selecting a customer. With referential integrity set, you can't delete a record from the primary table if there are matching records in the related table (unless you also choose to turn on cascading deletes), and you can't change the primary key value if there are matching records (unless you also choose to turn on cascading updates).

One-to-Many Relationships

Although Access doesn't require that linked fields have the same names (only the same data type), to make it easier to match up corresponding fields when setting up relationships, I recommend using the same name for linked primary and foreign key fields.

A one-to-many relationship (by far the most common type of relationship) is needed when a single record in one table can match several records in another table. In the Toy Workshop database, a number of one-to-many relationships are needed; they are listed below, with the "one" (or *primary*) table on the left and the "many" (or *related*) table on the right. Some of these relationships are also part of many-to-many relationships, covered below:

- tblCategories—tblToys
- tblCustomers—tblCustomerEmails
- tblCustomers—tblCustomerPhones
- tblCustomers—tblOrders
- tblEmployees—tblRepairs
- tblMailingListCompanies—tblMailingList
- tblMaterials—tblRepairMaterials
- tblMaterials—tblToyMaterials
- tblRepairs—tblRepairMaterials
- tblShippingAddresses—tblOrders
- tblShippingMethods—tblOrders
- tblToys—tblToyMaterials
- tblVendors—tblMaterials
- tblVendors—tblToys
- tblVendors—tblVendorEMails
- tblVendors—tblVendorPhones

If you get an error message "Toy Workshop can't create this relationship and enforce referential integrity" when trying to create a relationship, this indicates that data in one of the tables violates referential integrity (for example, you might have a tblOrders record without a value in the CustomerID field); fix the data, and you will be able to create the relationship.

Similarly, an error message that says, "Relationship must be on the same number of fields with the same data types" most likely indicates that the fields to be linked are of different data types; change the data type of one field so that it matches the other field (with AutoNumber matching Long Integer), and you should be able to set up the relationship.

As an example of how to set up a one-to-many relationship in the Relationships window (all the others are done similarly), let's set up the relationship between tblCustomers and tblCustomerPhones. Start by opening the Relationships window and dragging tblCustomers and tblCustomer Phones to it from the Database window (or alternately, selecting them using the Show Table dialog opened from the similarly named toolbar button). Note that the CustomerID field in tblCustomers is bold; that indicates that it is the primary key of this table. The matching CustomerID field in tblOrders is not bold, because it is a foreign key field in that table. To create the join, drag the CustomerID field in tblCustomers to the same-named field in tblOrders, as shown in Figure 1.31.

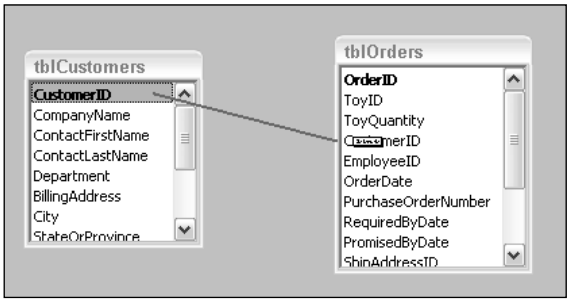

Figure 1.31

When you release the mouse, the Edit Relationships dialog opens. The Relationship Type box at the bottom of the screen displays the relationship that Access thinks is right; it is usually correct.

> *If the relationship type you intend to set up isn't shown in the Relationship Type box in the Edit Relationships dialog—for example, you want to set up a one-to-many relationship, and the box says Indeterminate—the most likely reason is that you have tried to link the wrong fields, or you linked the right fields, but they aren't of matching data types. Correct the problem and you should see the correct relationship type in the dialog.*

In this case, the relationship type is correctly identified as one-to-many, so all you have to do is check *Enforce Referential Integrity and Cascade Update Related Fields,* and click the Create button, as shown in Figure 1.32.

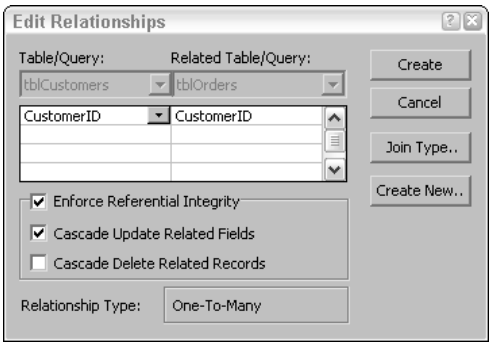

Figure 1.32

You can now see a line connecting the CustomerID field in tblCustomers to the matching field in tblOrders, as shown in Figure 1.33; note that it has a 1 on the left side (indicating that tblCustomers is the primary or "one" table), and an ∞ sign on the right side (the related or "many" table).

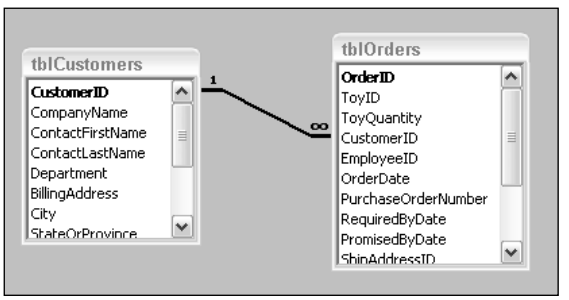

Figure 1.33

One-to-One Relationships

A one-to-one relationship (comparatively rare) is needed when a record in one table can only match a single record in another table. The linking field is the primary key field in both tables. Typically, such a relationship is created to limit access to certain data, such as confidential employee data. In the Toy Workshop sample database, there is a single one-to-one relationship, between tblEmployees and tblEmployeesConfidential. To set up this relationship, drag EmployeeID from tblEmployees to the same field in tblEmployeesConfidential; the Edit Relationships dialog will say One-to-One, as shown in Figure 1.34.

If the Edit Relationships dialog says One-to-Many instead of One-to-One, this indicates that the linking field is not the key field in both tables; change it to the key field in both, and you should be able to set up a one-to-one link.

In the Relationships window, the line representing a one-to-one relationship has a 1 at both ends, as you might expect.

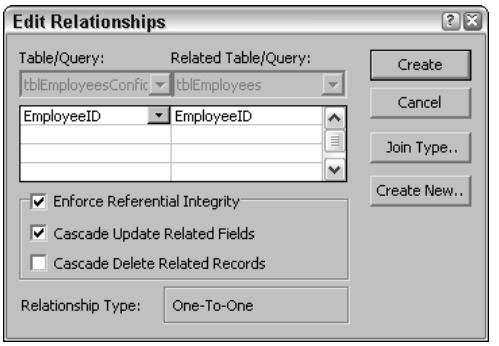

Figure 1.34

Many-to-Many Relationships

A many-to-many relationship is actually a set of two one-to-many relationships. There are two primary tables and a linking table; the linking table has two foreign key fields, one matching the primary key field of each of the primary tables. It may also (but usually doesn't) contain a few other fields that hold information related to that specific combination of records from the primary tables. In the Toy Workshop database, two many-to-many relationships are needed (the linking table is in the middle of each set):

- tblToys—tblToyMaterials—tblMaterials
- tblRepairs—tblRepairMaterials—tblMaterials

Once you have set up the two one-to-many relationships, you have a many-to-many relationship; Figure 1.35 shows the two many-to-many relationships in the Relationships window. You can see the two sets of primary tables with a linking table in between; tblMaterials serves as the primary table in two many-to-many relationships.

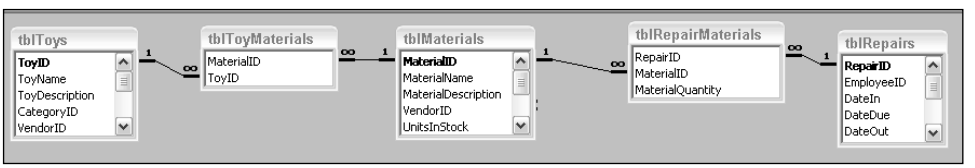

Figure 1.35

If you use a convention of naming all primary and foreign key fields with a suffix of ID, you can easily identify the fields that need to be linked to other tables in the Relationships window. However, not all key fields need to be linked to other tables—MailingListID in tblMailingList doesn't need any links because there are no tables with multiple records matching one record in tblMailingList.

Summary

Now that we have created the necessary tables for the Toy Workshop database, with appropriate relationships set up between them, we can proceed to create forms for entering and editing data, and queries for sorting and filtering.