Chapter 4

Database Design – The Brokerage System

Chapter Outline

4.1 Database Design – what does it gain you?

The design of your database determines how useful it will be – done correctly, your design will provide useful reports, answers to interesting questions, enforcement of business rules and can even assist with data entry by validating data as it gets entered, giving your data a standard, consistent look. Your design can actually provide Access with a set of rules that help keep "bad" data out of your system. Whether bad data gets in or not depends, in part, on your abilities as a database designer. Even as beginners, we can take advantage of several tools provided by Access that watch data as it arrives and prevent obvious errors. Some of these tools include

Input Masks

Validation Rules

Table Lookups

Defaults

Referential Integrity

Cascading Deletes and Updates.

Each of these are fairly simple additions to our design and well worth the few minutes it takes to get familiar with them.

In addition to introducing these new design features, this chapter also begins creating a system that will be used throughout the database section of this book. We start with the design of our tables and follow that with some refinements – specifically input masks, validation rules and referential integrity. These three features will watch over our data entry to help keep it clean. Cascading deletes and updates will help keep the database clean once we have data in there. What's cool about this is that Access does all the work… once we establish a few rules. So, follow along as you and a few friends build a database for a small brokerage firm.

4.2 Overview of the Brokerage database

Well, it's been ten years since you and a few friends graduated with degrees in financial services. Each of you have passed rigorous certifications in stocks, mutual funds and insurance and have received licenses in Massachusetts, Rhode Island and Connecticut. Your work for a large financial services firm has given you quite a bit of insider knowledge about the investment business. Last Friday, after work, the three of you met for dinner and began to plan the start of your own firm.

Harry has been charged with finding the costs of the various financial research systems you will need. He needs to line up the required purchases. These services provide current stock prices, graphs of price and "return on investment" data as well as comments by analysts who are specialists in each stock. The cost of these systems is fairly high but you can't provide investment advice without them – and we have the added benefit of getting up-to-the-minute stock prices.

After bragging about your background with Microsoft Access, your buddies have challenged you to build the customer database. That database will store data about each customer and the stocks they own. The brokers will need a simple system for locating a customer's records and reviewing them on screen. There will also be reports of both customer activity and broker activity. Initially, customers will be mailed their reports. Our goal over the next few months is to enhance the system with a web interface that provides customers with an online view of their accounts. Not only will our customers be pleased with the convenience, but it will reduce our considerable mailing costs.

Not much can happen until the database tables have been designed and related to each other – so that's where we'll start.

4.3 The database design process

There are four major steps in designing the database.

- Identify your tables
- Decide what fields go in which table
- Locate a primary Key for each table
- Find a way to relate the tables to each other

4.3.1 Identifying the Tables

Professional database developers follow a fairly involved process to help organize their data into tables – they spend a lot of time at the design stage since a database with design problems will be very difficult to work with. One of their techniques is to create **entity relationship diagrams**. These "ER" diagrams attempt to identify the major objects (entities) in the business and then look to see how these entities are related to each other.

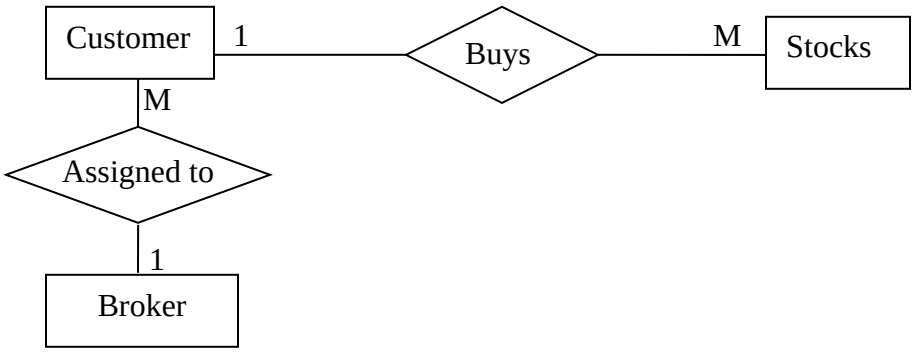
The database designer will meet with management and others involved in creating the new database. An understanding of the business is crucial to identifying the entities for

your ER diagram. In our case, the brokerage firm has three obvious entities. We have our employees (**Brokers**), we have **Customers** and we have **Stocks** to sell.

Customers
Brokers
Stocks

Once your entities are identified, you normally have just discovered your tables! In our case it looks like we'll need a **Customer** table, a **Stocks** table, a **Broker** table.

A simple ER diagram shows how each of these entities is related to the others. Notice how each entity is represented on the diagram along with a diamond shape identifying the relationship. We think that Brokers are assigned to Customers and that Customers buy Stocks. Further, we think that ONE Broker is assigned to MANY Customers. Reading that relationship in the other direction, we think that each customer is assigned to only ONE Broker. The diagram also suggests that a Customer can buy MANY stocks.

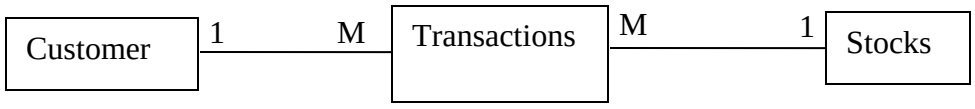


The process of building the "ER" diagram helps us focus on the big picture – before getting bogged-down on the details of naming fields and selecting their lengths. If you miss something at the broad "ER" view, the database is doomed to problems.

Have we missed anything? Let's think about the relationship between Customers and Stocks. Sure, a customer can buy many stocks – but isn't it true that a Stock can be owned by many customers? If that is true, then the relationship is not One-to-Many, but is actually Many-to-Many (often written as "**M:N**").

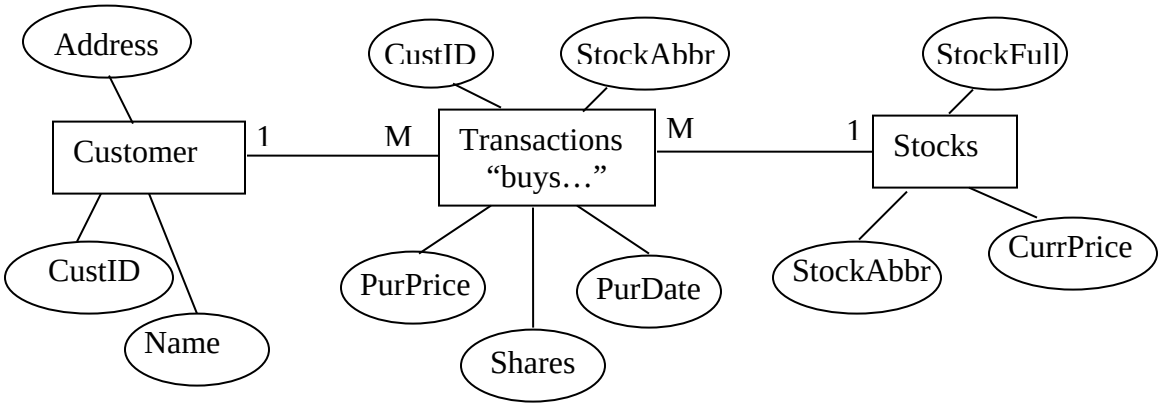
Knowing that we have a Many-to-Many relationship makes a big difference in our design. Relational databases typically deal with these M:N relationships by adding a connecting table – Access calls these **Junction Tables (**you'll also see the names ***Bridge*** and ***Linking*** tables used**)**. I'll create a junction table called "Transaction" that will represent the list of stocks owned by each customer (I guess, in a way, I'm representing the "buy" relationship). Customers can appear many times in the list of transactions and stocks can appear many times.

The junction table will contain the key fields from both the Customer and Stocks table. In this way, it represents a link between these two tables. Notice how this junction table breaks the Many-to-Many relationship into two One-to-Many relationships. Each Customer is mentioned once in the Customer table – but can appear in the Transaction table many times.



Without thinking our way through the ER diagram we might have missed the need for this extra table.

The final ER diagram might include the fields associated with each entity. Database people might refer to these as the ***attributes*** of the entities. To keep things manageable I've left off the Broker table.



4.3.2 Deciding What Fields Belong in the Tables

Now that we have tables it's time to decide what goes in them. Database professionals use their entity relationship diagrams and a process called normalization. For us, two simple rules will do nearly as well:
If an entity has a feature that occurs just once – then it probably goes in the table for that entity. For example, Customers have one First Name, one Last Name and one Social Security Number. You should also be careful to include only those fields that truly relate to that entity – customers have names, addresses and phone numbers. Yes, they have a broker, but should the broker's phone number be in the Customer table? Is that phone number truly a feature of a customer?
If an entity has a feature that repeats itself, that field normally goes in a separate table. In our case, each customer can purchase stocks. What would happen if we stored the customer's stock holdings in the Customer table? If you provided a field for Stock you

would be in trouble if they bought two stocks. Naturally, you could make Stock1 and Stock2 fields. What if they bought 20 stocks? Well, you could make 20 fields (Stock1 through Stock20). Could you ever predict the maximum number of stocks a person might buy? If you set up 20 fields named Stock, how much wasted space would there be if most people only owned one or two stocks? The varying number of Stock fields also makes calculations difficult since you are juggling 20 separate fields of data! If we wanted a list of customers who bought IBM, you would need to test every one of the 20 Stock fields to determine whether a customer owned that stock.

Thinking back to the entity relationship concept, a stock really isn't a feature of a customer – another reason to keep it out of the Customer table.

Take a look at what we've decided to place in each table – argue with your instructor if you think we've made a mistake. Notice how we've included CustID in both the Customer and Trans tables. We did this in order to link these two tables together. Study our table designs to anticipate other links we need to make.

Customer Table			
Field	***Data Type***	***Size***	***Comment***
CustID	Text	5	Unique identifier for each customer, our Primary Key – starts with a "C" followed by 4 digits
CFname	Text	10	Customer's given name
CLname	Text	15	Customer's family name
Street	Text	20	Street name including number
City	Text	20	
State	Text	2	Both characters should be uppercase
Zip	Text	5	We can require all 5 digits…
Phone	Text	13	13 characters if you count the parentheses and hyphen that a complete phone number would have
Photo	Ole Object		Customer photos are stored as Jpeg images – this field establishes links to the images. NOTE: getting images into your table can be tricky – see 'Geeks' for Chapter 5
BrokerID	Text	5	Relates to the Broker Table, where the broker's name can be found

Broker Table			
Field	***Data Type***	***Size***	***Comment***
BrokerID	Text	4	Unique identifier for Broker, Primary Key – starts with "B"
BFname	Text	10	Broker's first name
BLname	Text	15	Broker's last name

Stock Table			
Field	***Data Type***	***Size***	***Comment***
StockAbbrev	Text	5	Unique, our Primary Key
StockFull	Text	20	Full name of the company
CurrPrice	Currency		Current Price of the stock

Trans Table (short for 'Transaction" table)			
Field	***Data Type***	***Size***	***Comment***
TransID	AutoNum		Unique for each transaction, Primary Key
CustID	Text	5	Links to Customer table where we can find the full customer name
StockAbbrev	Text	5	Links to the Stocks table where we can find the full name and current price of the stock
PurPrice	Currency		Purchase Price of the stock
PurDate	Date/Time		Date stock was purchased
Shares	Number		Number of shares purchased

4.3.3 Selecting Primary Keys

Recall that primary keys are used as unique identifiers for our data. They provide a way to distinguish among the various "Bob Smith" customers we have. Do your tables absolutely need primary keys? Well, no…but if you don't have them it will be impossible for you to create relationships between tables – and that is one of the really powerful features of a relational database. Our Customer, Broker and Stock tables have fairly obvious choices for primary keys.

Customers are normally given CustomerIDs that are unique to each customer, so that becomes a very natural primary key. You could have selected Social Security numbers for the keys but companies are increasingly reluctant to expose those numbers. Be very careful when you select the key – values that seem unique to a customer might not be. For example, could you have used telephone number? A person might have several numbers, but they will provide their main number when we set up their account. The difficulty comes when there are several people in a household who have accounts with us. One of our customers has their son living with them. After graduation he decided to live at home and save for a down payment on a house. Both the parents and the son have accounts with us – but share a telephone number. Using telephone number would not allow us to distinguish between the parent and son accounts.

The Broker table is easy. Each of our brokers is assigned a BrokerID by the state of Rhode Island. They are licensed in other states, but RI is our home base. Rhode Island issues unique IDs to each licensed broker which makes that license ID a natural for the primary key.

In the Stocks table we are using the abbreviation of each stock as our key. The abbreviations are assigned by the stock exchange and are guaranteed to be unique.

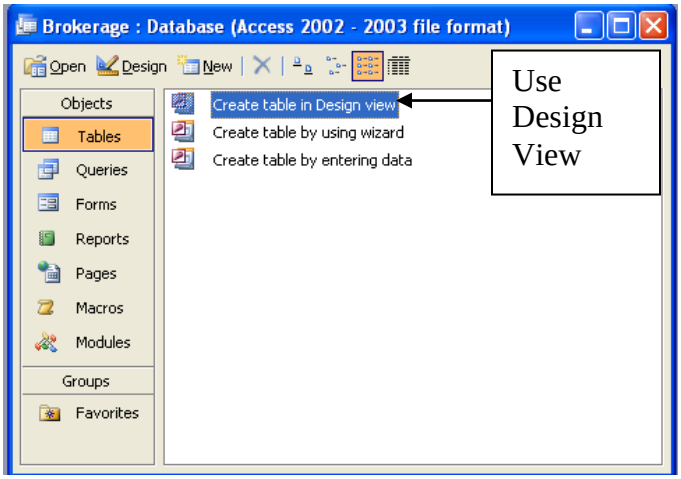
The only difficult key is the one we need for the Trans table. That table contains CustomerID – however while CustomerIDs are unique values in the Customer table, these values may appear in the Trans table a number of times (if you own four stocks, your CustomerID will be in the Trans table four times). So while a Customer is mentioned only once in the Customer table, they can own many stocks. As you scan the list of fields in the Trans table you'll find that none of the data fields are unique – for example, many people can buy a stock on the same day and same price. Our solution has been to number each transaction and use that number as the primary key. The ***autoNumber*** data type is perfect for this – it will start at "1" and increment each transaction from that point.

4.3.4 Relating the Tables

Our final step in the design process is deciding how the tables relate to each other. This involves finding common fields, shared by two or more tables. You are looking for a primary key in one table that appears in a second table as a regular field. A primary key that appears in another table (not as a primary key) is called a ***foreign key***. In our case, CustomerID plays the role of primary key in the Customer table, but is a foreign key in the Trans table. The same thinking holds for StockAbbrev; StockAbbrev appears once in the Stock table but may appear many times in the Trans table.

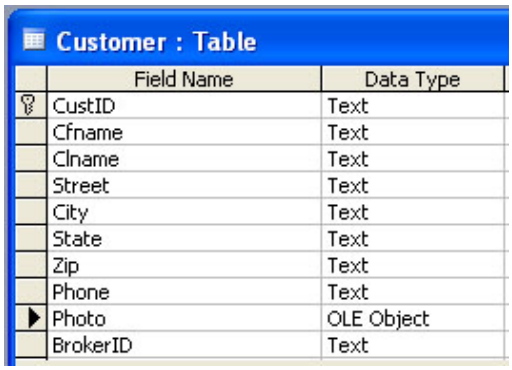
4.4 Building our tables

Let's get into Access and begin building the tables for our new Brokerage Firm. Open Access and ask for a Blank Database. Give the filename as **Brokerage.mdb**
On the Table tab, create your table in Design View.



We'll start with the Customer table.

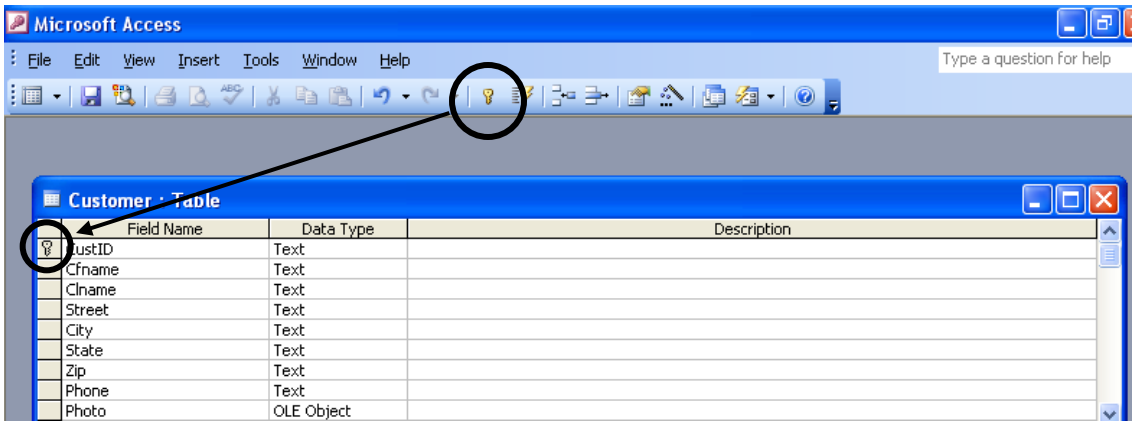
Start by entering the names of our fields and decide on the data type for each. I know it doesn't make sense to define zip code and phone as Text when we are accustomed to thinking of these as numbers – but we aren't doing any math with them, so defining these fields as numbers doesn't gain us a lot.



In fact, leaving them as Text will let us use "Input Masks" that will dress the data up a bit as we enter it. For example, phone numbers have parentheses around the area code and a dash between the exchange number and the final part of the phone number. An input mask will put those extra symbols in for us. The zip code doesn't need a mask (unless we decide to implement the new zip+4 system. Our rationale for leaving zip as text lies with the leading zero that so many zip codes have. If we had defined zip code as a number, the zip code 02806 would display as 2,806! Don't worry about the masks yet – just make sure you define each field with a ***specific length*** (refer to the data definitions for the Customer Table).

Selecting the Primary Key

Before we start working with Input Masks and Validation Rules, let's define CUSTID as the primary key for the Customer Table. Click on CustID to select that field, then click on the Key symbol on your toolbar. You should now have an image of a small key sitting next to CustID.

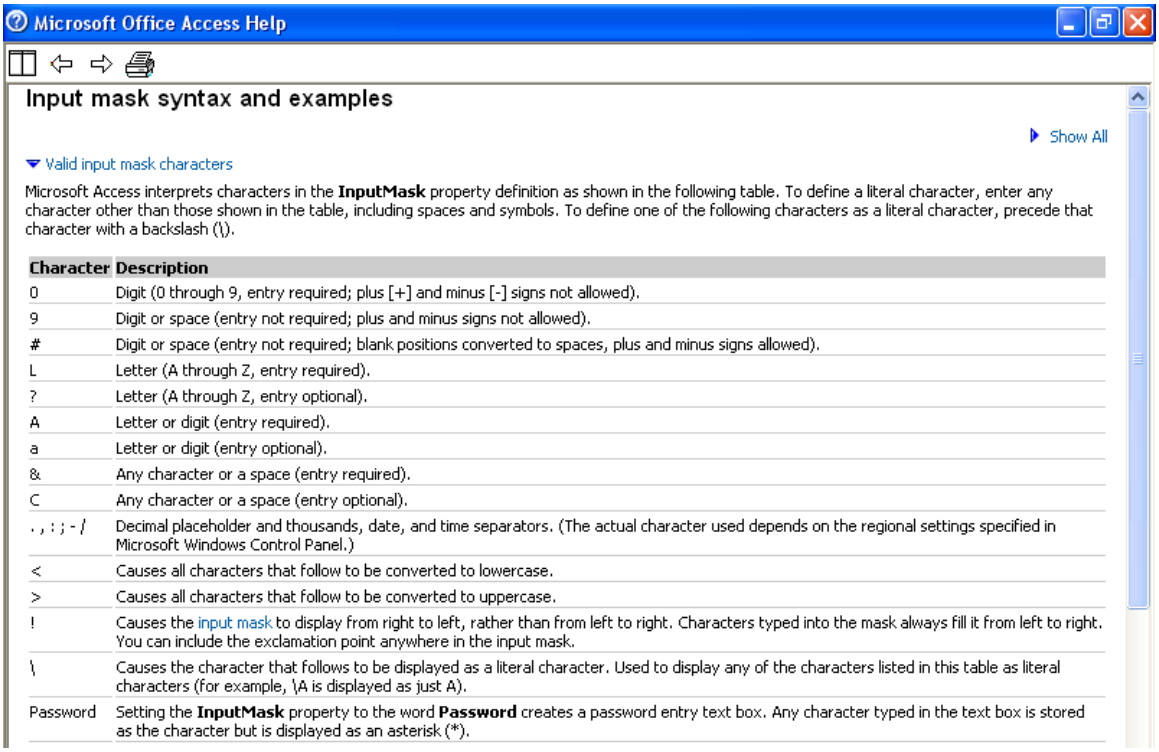


4.5 Saving the Table

This might be a good time to save our work on this Customer table. Access will prompt you to save the table and give it a name once you close this table by clicking the "X" in the corner of the design window. Be careful not to click the "X" that shuts Access down! When you're prompted for a name, call the table **tblCustomer**. Once it closes, Access automatically saves it. We have more work to do, so click on the Customer table ONCE and move back into design mode.

4.6 Input Masks

Most database developers are concerned with the consistency of data – and we can do something about that for some of our fields by using Input Masks. These are codes that define the look of our data. In effect, they represent a template for data entry. The figure below has a listing of the input mask symbols. In addition to providing consistency to your data, they also help enforce your business rules. For example, when your brokers set up a new account, we need a way to require a full phone number, one that includes the area code. That's a very simple rule, but the same strategy can be applied to more complex ones.



Don't get overwhelmed by the subtle differences between some of the input mask symbols. For example a "0" requires a digit (but it won't accept plus or minus signs) while a "9" represents a digit – but this time it isn't required that you enter one. It also allows a space (but still no plus or minus signs).

4.6.1 Masks related to letters and other non-numeric characters

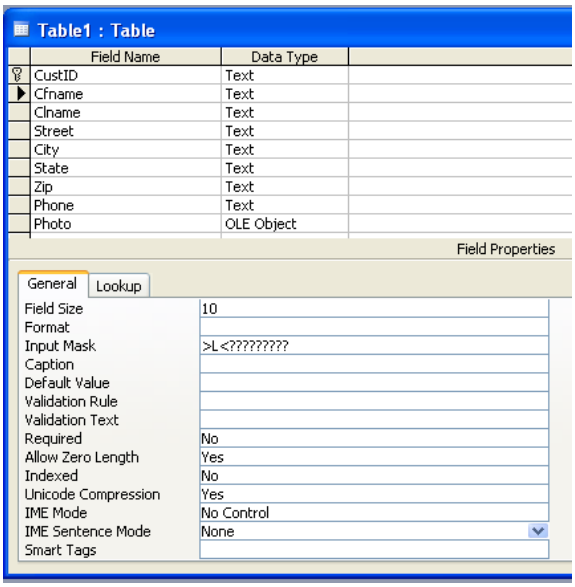
Let's try out a few simple input mask symbols.

Mask Symbol	Meaning
>	Make what follows be uppercase
<	Make what follows be lowercase
L	Require a letter at this location
?	This is a letter, but not required

For consistency we want first names to start with Uppercase characters and we want the remainder of the name lowercase, so we need to use both ">" and "<". Since all names must have at least one letter we'll required the first character – but the remaining letters will need to be optional since we don't know how long each name is. The mask would be:

In English it reads… "*Uppercase the first character (which is required) and then lowercase everything that follows*". The remaining letters are all optional. Notice how we used enough "L" and "?" symbols to fill our 10 character name field.

>L<????????????



You probably want to use a similar mask for City – I'll leave that one up to you.
Let's try another one – this time for State. We need to indicate that both letters in the state abbreviation are required and that they should be uppercased. So we will start with the uppercase symbol ">" and the use two "L" symbols to indicate that we are expecting alphabetic letters and that they are required.

>LL

4.6.2 Input Masks involving numbers

Now let's work with numbers. Our zip code is defined as a Text field – but we clearly want there to be numbers in the zip code. Defining Zip as Text just indicated that we weren't doing math with the field.

For a mask involving digits we have a choice of:

0	Digit with no spaces, plus or minus signs – and it is Required at that spot
9	Digit OR space, but no plus or minus signs – and it is Optional at this spot
#	Digit OR space and allows plus and minus signs – it is Optional at this spot

Zip codes require all 5 digits so we have little choice here, since only "0" requires the user to enter something. That will make our zip code mask be a group of 5 zeroes.

00000

The input mask is neat because it insists that all zip codes are 5 digits long – no one can accidentally enter a partial zip code since our mask has 5 required digits.
Phone number introduces two slight complications. The first is that area code is something we can control – if we want to insist that all of our phone numbers include the area code, then we should use the "0" symbol for it. If area codes are optional, then we need to indicate that by using "9". I think it's far too chancy for a business to omit the area code, so we'll require it. In a sense, the mask is helping to enforce one of our business rules – that we demand an area code from each customer.

Required Area Code: 000 000 0000 **Optional Area Code: 999 000 0000**

4.6.3 Input masks with "literals"

The second issue is that we want our brokers to be able to enter the phone number without having to enter the parentheses around the area code and the hyphen that separates the three digit Exchange from the last four digits. Not only does it make things easier for our brokers, but it also cuts down on data entry problems. If you check out our list of mask symbols you'll find that any "***literal characters***" need a backslash before them. So if you wanted an "L" to be included as the user entered the data, there needs to be a way to distinguish the "L" that you want displayed from the "L" that is our normal mask symbol – and the backslash does that. Since none of the symbols in the phone number can be confused with any current mask symbols, I suppose we can get away without using the backslash. In fact, try entering the mask without the backslashes and see what happens!

Literal means "*for real*". We often hear people say that something literally happened. With computers a "literal" means "we want that actual character". So if we see the letter "L" in a mask, is it a mask symbol that stands for a "letter" or does someone literally want the letter "L" to be in this spot?

You can enter either…
(000)000-0000
Or
\(000\)000\-0000

Finally, CustID is defined in our data dictionary as starting with an uppercase "C" followed by 4 digits. We can use a mask to automatically get the "C" in front of each number. That way, our brokers can just enter the customer's number and we'll automatically put the "C" in front of it. As with the phone number, we need to use the backslash to be sure our "C" doesn't get confused with the mask character.

\C0000

Input Mask Tricks and Complications!

Access would prefer to save room where it can. Telephone numbers are a great example – do we really need to store the two parentheses and the hyphen for every telephone number? Access would prefer to use the mask for display and data entry purposes, but 'wants' to store the telephone numbers without all the extra stuff.

In the case of our Customer and Broker IDs, we had hoped the "C" and "B" would get entered for us. If those characters don't actually make it into the tables, we're left with IDs that are just the digits! If you searched for customer "C1234" you would never find them if Access stored the ID as "1234".

We need a way to have the input mask "literals" stored in the table when we decide that is best for us…

The solution is to follow the mask with a ***semicolon and the number zero*** – Access takes that as an instruction that it must store the mask literals.

You should correct both the CustID and BrokerID masks…

\C0000;0
\B0000;0

4.7 Validation Rules

In addition to input masks, there are other ways to monitor data as it gets entered. A ***validation rule*** can make sure numbers fall within specified ranges or that words come from an approved list. For example, our brokers are each licensed to deal with customers in Massachusetts, Connecticut or Rhode Island…so our customers have to be from one of those three states. If a broker tries to sell to someone in another state we can issue a warning message and refuse to accept that customer.

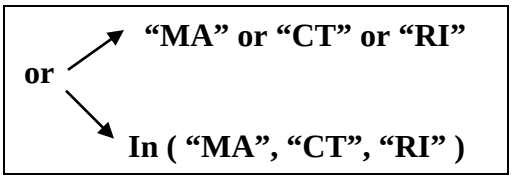
Three entries on the table design are involved in the validation:

1) *Validation Rules*

2) *validation text* and

3) the *field's description*.

4.7.1 Validation Rules describe what valid data looks like. For our "State" validation, click on the State field in your table design and then on Validation Rule. We can specify the rule in one of two ways.



The use of "IN" says that the entry must be "IN" the following list of items. The quotes around each item follow common computer practice that says words need quotes, whereas numbers don't get quotes.

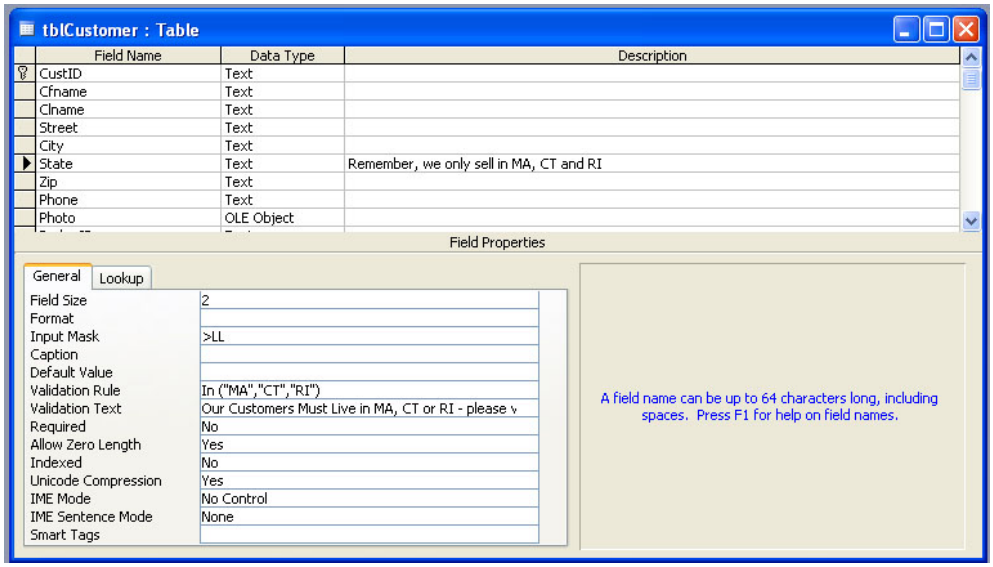
Check out this list of other validation rule examples.

Handy ways to represents ideas using math symbols	
A number can't exceed 20	<= 20
A number must be *at least* 20	>= 20
A number must be *between* 20 and 40	>= 20 and <= 40
Same idea… but using the word "between"	Between 20 and 40
The word must end with a "Y" – the asterisk is a wildcard	Like "*Y"

4.7.2 Validation Text describes the error message to display when someone violates the rule. You can leave it blank, but that doesn't make much sense. Why would you prevent data entry without explaining why the data was rejected? The message is simple text that you want displayed. In our case the message would be something like:

Our Customers Must Live in MA, CT or RI – please verify customer's location

4.7.3 Description is the area to the right of each field definition. Any text you place there will show on the ***status bar*** at the very bottom of the screen. We can use it to guide our brokers through the data entry process – a quick reminder that we only sell in three states. Consider it a warning that they can see before entering a wrong state!



Collecting Our Thoughts

Let's take a pause and review what we've done. Much of the table design is a review of what you did while setting up the "Computer_Survey" database in Chapter 3. Our Brokerage system has a few more tables, but the design process has been very similar. The new features are centered on making data entry a bit easier and at the same time a bit cleaner. We've used both input masks and validation rules to attempt to keep our data consistent. We don't want some people entering "ri", others "Ri" and some "RI". The input mask helps to keep the look the same. Our use of validation rules has helped enforce some of our simple business rules. Those rules apply whether you are entering data directly into the tables – or using forms (you learn more about forms in Chapter 5).

4.8 Building our other tables…

Since we have the data dictionary for each of the remaining forms, I'll move through those fairly quickly and leave most of the work to you. The use of ***defaults*** and ***lookups*** are the only new feature we'll introduce.

4.8.1 Building the Broker Table

There are only a few things to think about. Each brokerID is 5 characters long, starting with a "B". It's just like CustID, so we can use a similar input mask.

The mask says "we start with the letter "B" then require 4 digits. Remember to use the "semicolon zero" at the end (we want in insist that Access stores the B as part of the BrokerID).

\B0000 ;0

The broker's first name can use a mask that automatically applies the correct case – uppercasing the first character and lowercasing the remainder. Since we don't know how many letters are in the name, only the first letter will be required.

>L<????????

Finally, the key needs to be set to BrokerID. That completes the Broker table!

4.8.2 The **Stock table** is also fairly simple. Each stock has a unique abbreviation that we use as our primary key. We also need to know the full name of the stock and its current price. Nothing special needs to happen.

4.8.3 The Trans table is a very active place!

This is where we record the stocks a customer currently owns. We've already discussed the problem of finding a primary key for this table. I mentioned that a professional database designer will often look for combinations of fields that make a record unique. In this table CustID and PurDate might be a good combination. Since the date can include the time of purchase, it would be impossible for a customer to buy two stocks at exactly the same moment in time. These ***composite keys*** are very useful and might be a good approach for our Trans table. We have, however, decided on the other common approach to recording transactions – that of sequentially numbering the transactions. Using an AutoNumber as our data type will take care of the details for us; each transaction will be assigned a number one higher than the previous one.

You should have no trouble setting up this table since we've already dealt with CustID and StockAbbrev. Price is defined as currency – one of those fields that don't give us an option to specify size.

4.9 Setting Defaults

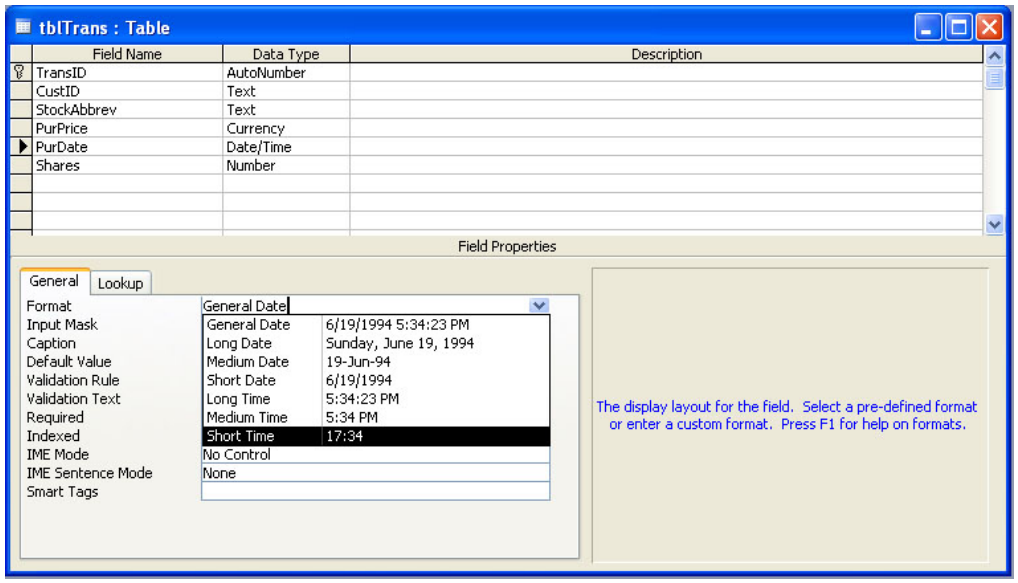
PurDate is defined as Date/Time which is another data type that doesn't let us specify size. The one interesting thing we *can* do is provide a ***default date***. Unless the broker enters a different date (and why would they?), the date of the transaction should match today's date. We can tell Access to enter the current date automatically – by **default**. The current date can be found by calling on a function in Access called **Date( )**. Setting the default is very simple. When you're defining the PurDate field, click on ***Default Value*** (two down from Input Mask) and type **Date( ).** Be careful not to put a space between the two parentheses. While you're setting the date field notice that various formats available to you. ***General date*** will display both the date and time – which might be helpful if you need to display exactly when a customer bought or sold a stock.

Sometimes arguments are useful!

Every function name ends with a set of parentheses – that's how you know the thing is a function. Sometimes the parentheses enclose ***arguments*** – basically one or more values that the function needs to work with. A good example is the TRIM function that is used when you need to chop extra spaces off a field; it needs to know what field needs to get trimmed.

e.g. *Trim (Lname)*

The Date() function doesn't need any additional arguments, so the parentheses are empty.

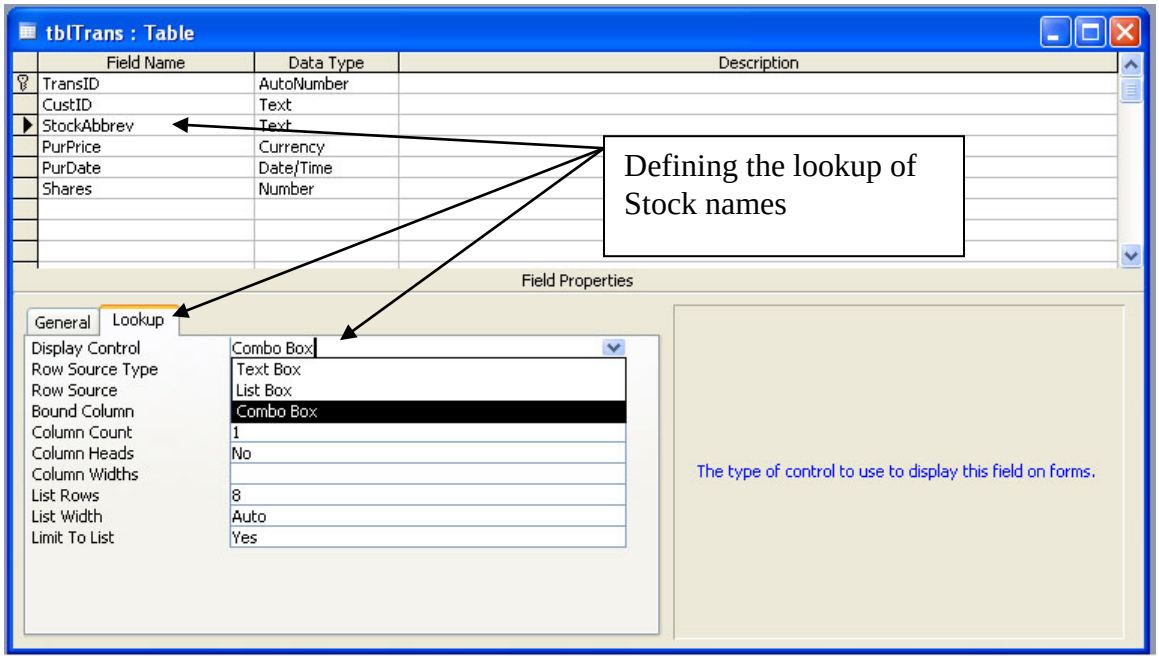


4.10 Lookup Tables

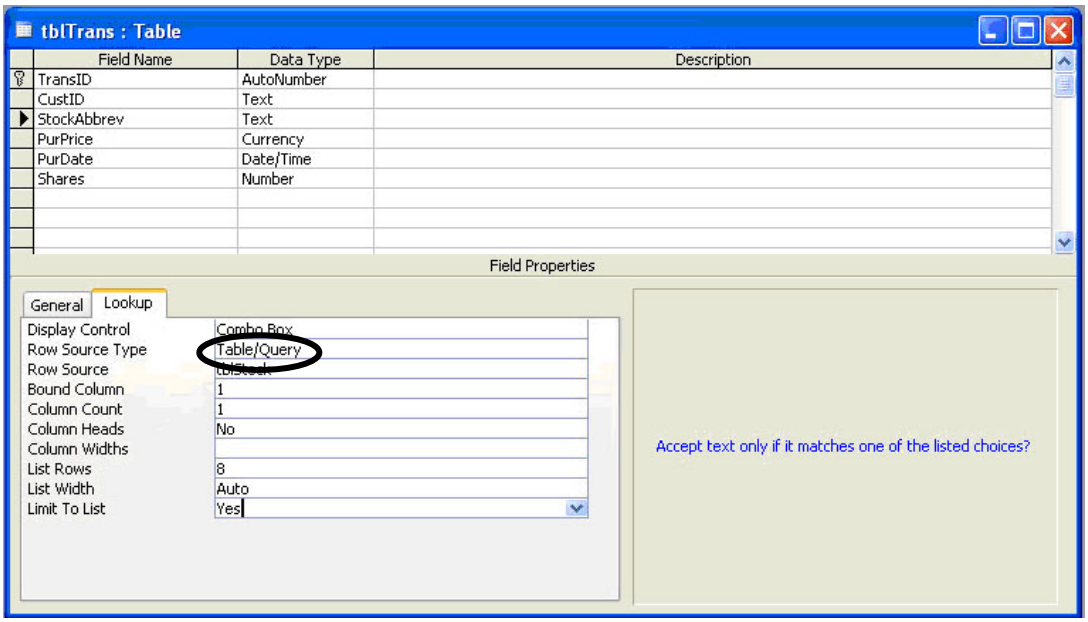
Let's check out one final approach to keeping our data clean. When a broker completes a sales form indicating that a customer has purchased a stock, we want to be sure the name of the stock is correct – let's not leave it up to the broker to get the abbreviation correct. Each of the valid abbreviations exists in the STOCK table. Access provides a fairly simple method of making the values in one table available to another table during data entry – it's called a ***Lookup***. As the broker goes to enter the name of the stock, a drop-down list will display a list of the valid abbreviations from the Stock table. Well, the technical name for the list is either a ***list box*** or a ***combo box***. Each of these is a list that drops into place as the user clicks an arrow. The difference is that a *list box* limits you to using an item from the list – while a combo box provides a spot up top for the user to enter some other value (well, you actually can limit a ComboBox to the existing list by setting the value of the ***Limit to List property*** of the Combo to the "YES" setting).

The text you enter in a ComboBox also provides a pretty neat added functionality to the Lookup – if you enter an "S", for example, you will be taken directly to the stocks that start with "S".

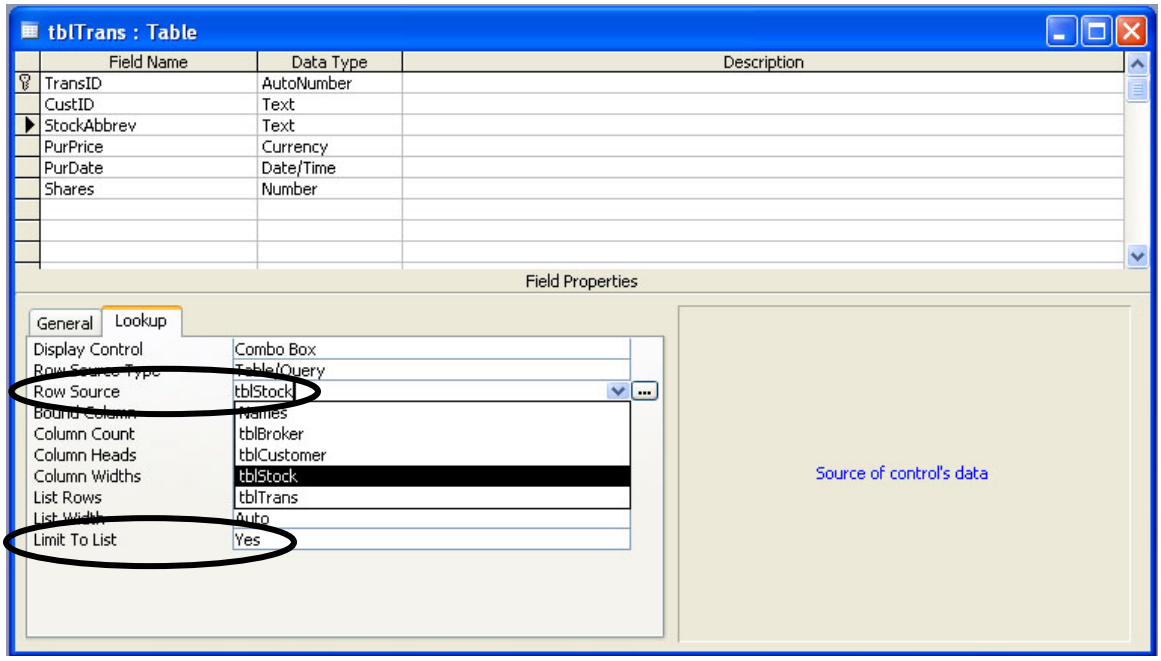
Start the process by clicking on the StockAbbrev field in the Trans table (design view). From there, click on the Lookup tab and select **ComboBox** as the ***Display Control*** (that is, how would you like the lookup displayed).



Once Access knows how the data should be displayed, it needs to know where the data is coming from. In our case, the data is sitting in the Stock table…so move to ***Row Source Type*** and select *Table / Query*. Basically, Access needs to know what kind of source will fill the rows of the list.

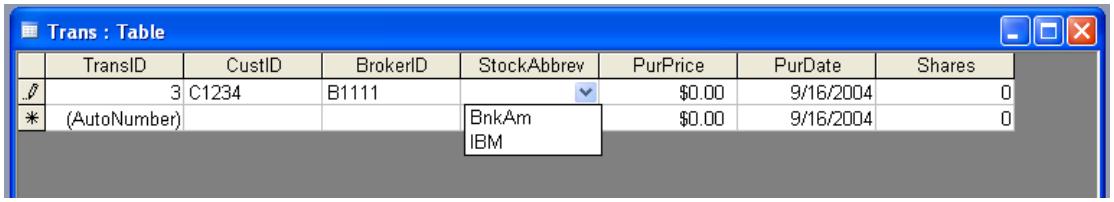


Now that Access knows the data will come from a table, we need to name that table. Click on ***Row Source*** and use the drop down list to select Stock as the name of the table.



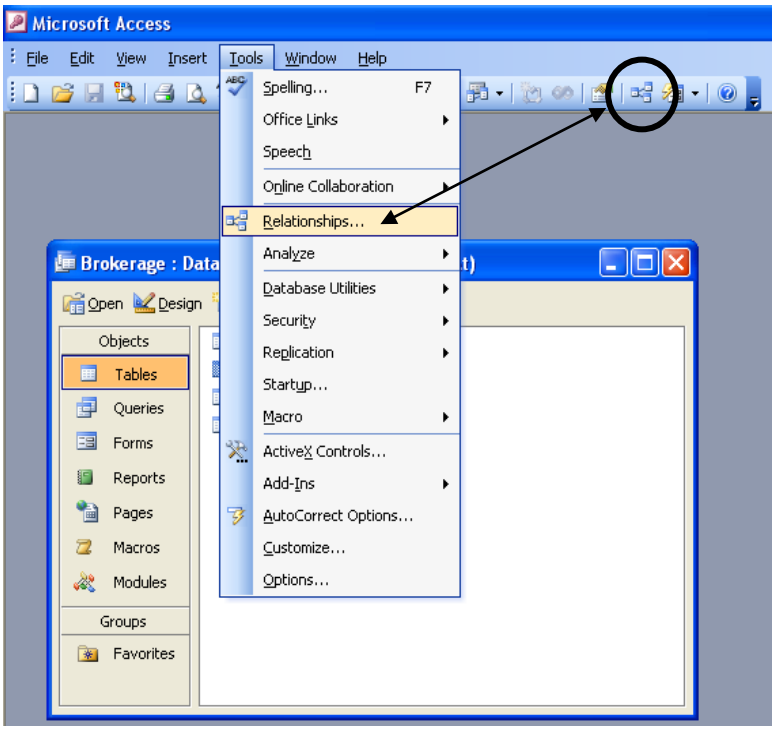
One of the problems with a ComboBox is that you are letting the user type something – in our case, the name of a stock. The name they type might not be a real stock abbreviation and we need a way to protect against that. One solution is to set "***Limit To List***" to "***yes***". Now the user can take advantage of the ComboBox's ability to scroll to the names that match the starting character the user might have entered – and we can still protect ourselves from a user who might enter an incorrect name.

At that point we're done! You might be asking how Access knows what we want displayed from the Stock table (given that three fields are in that table). Actually, it knows we want StockAbbrev – since that is the related field in the Stock table. When you get around to entering data, a list becomes available – and you need to make your selection from that list.

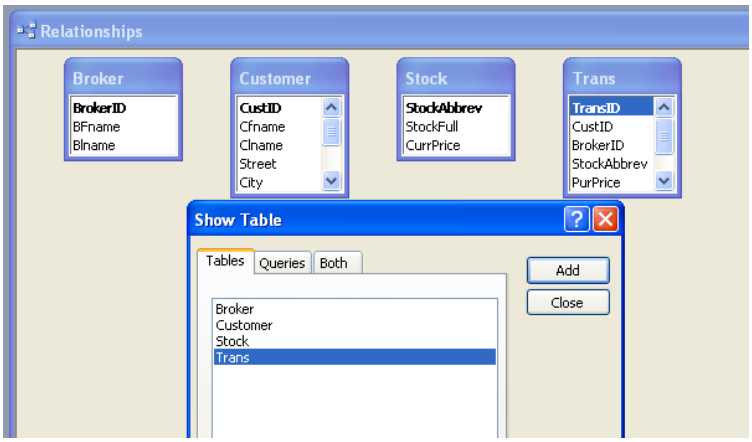


4.11 Establishing the Relationships

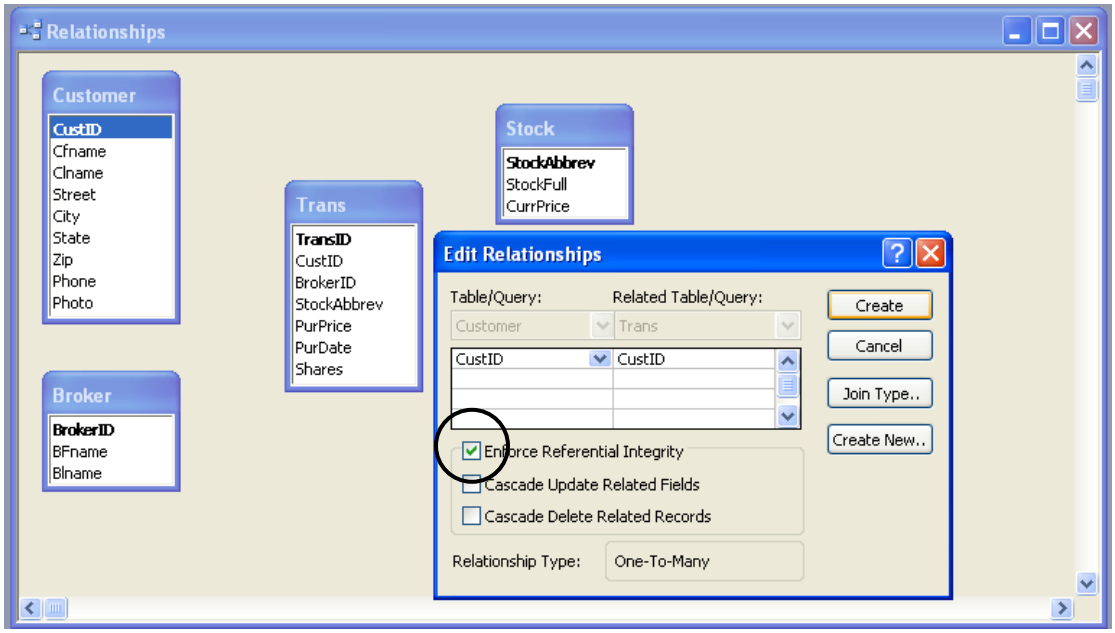
Now that each table is ready to go, let's spend a moment tying them together. There are two ways to reach the relationships screen – you can select it from the Tools menu or click the tool on the main toolbar.



Once you have the relationship screen, click each table and the ADD button until all four tables have been added to the Relationship Diagram. You can also add the tables by simply double-clicking each table (now you don't need to hit the ADD button for each table!)

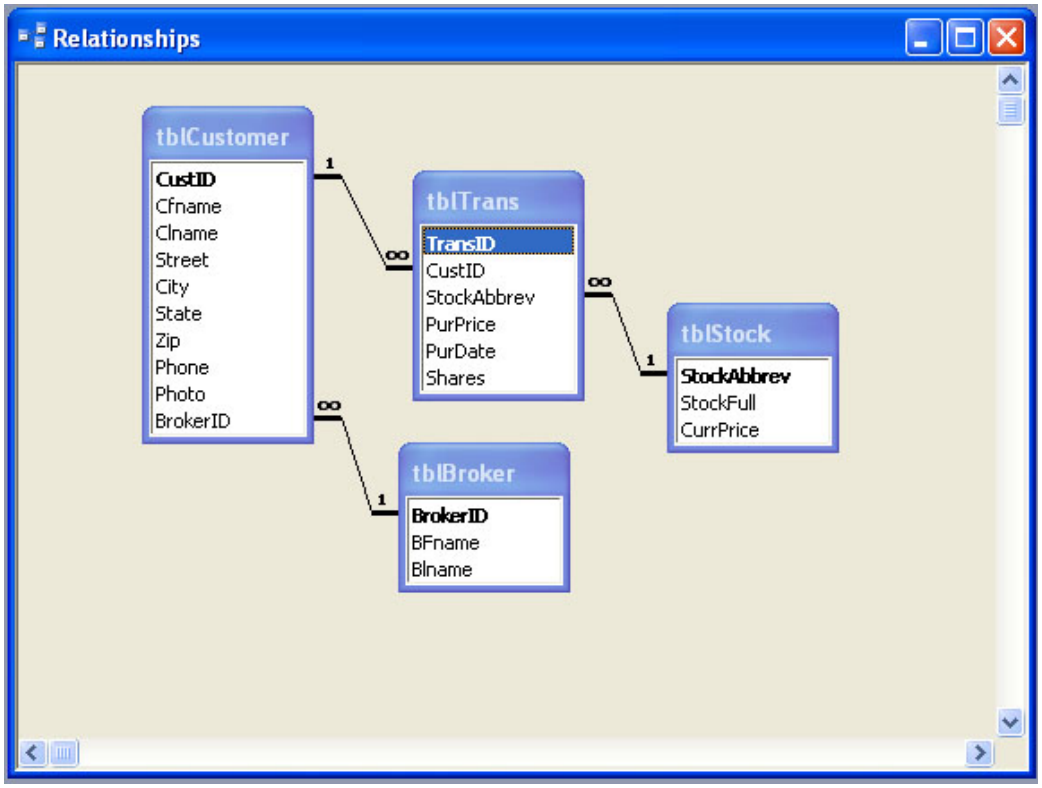


Note how the primary key for each table is highlighted in bold. You'll want to find the common fields in each table and drag the mouse from one of those fields to the other to make the connections. Most of the relationships will be heading to the Trans table. As you drag from CustID in the Customer table to the CustID in the Trans table a dialog box pops up asking you to Create the relationship. You can also decide to enforce Referential Integrity.



4.12 Referential Integrity

Your database would work without referential integrity but it's a great safety net. Basically it says that you can't have a record on the many-side of a relationship if a corresponding record doesn't exist on the one-side. In our example, you can't have a transaction in the Trans table if that customer doesn't exist in the Cust table. In a similar fashion, the broker and stock mentioned in the Trans table better exist as well! The details are all handled by Access – so if someone's willing to be sure your database doesn't have loose ends everywhere, why not take advantage of it? While you're at this Relationship dialog, notice the Cascading Deletes and Updates. These help with database maintenance. If you delete a customer from the customer table, it might be nice if all reference to that customer disappears from any tables that mention them. Any updates will ripple through the database as well – provided you checked the box. I know this sounds pretty neat – but it's also somewhat dangerous, so be careful… we won't be using *Cascades* in this project.



Your final Relationship Diagram should resemble this figure. Move the tables around until the lines aren't crossing each other. If your tables don't display every field, hold your mouse on the bottom edge of a table and drag downward (stretching) until all fields are showing.

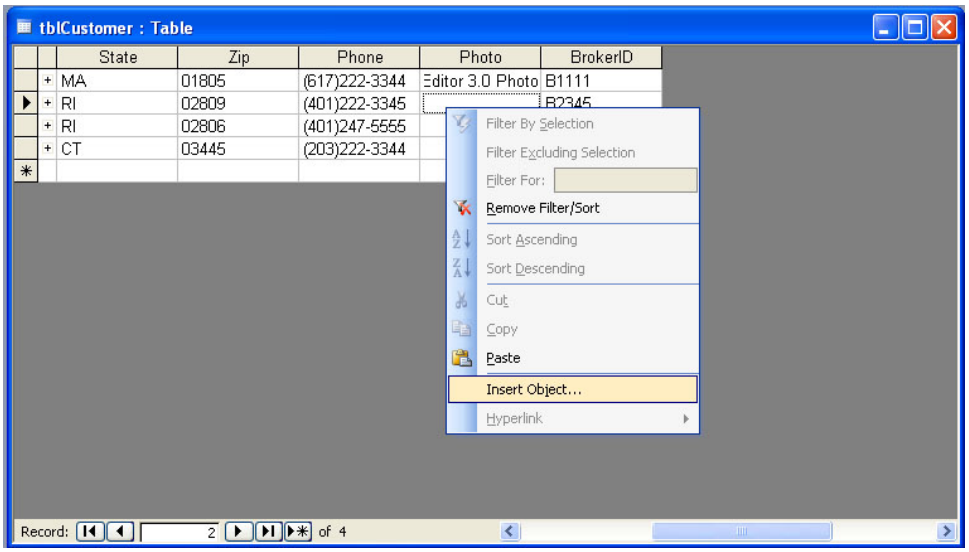
4.13 Entering Some Sample Data

Let's enter at least one record to see how our input masks, validation rules and defaults are working. We will use the actual tables for our data entry – but, keep in mind that entering data directly into tables is very rare. In "real" databases, the database developer will provide data entry forms (some kind of user interface) rather than allowing users to play with the actual tables! Later in this book (Chapter 10 and 11) you'll learn how to build web forms that let people on the internet enter data into your tables.

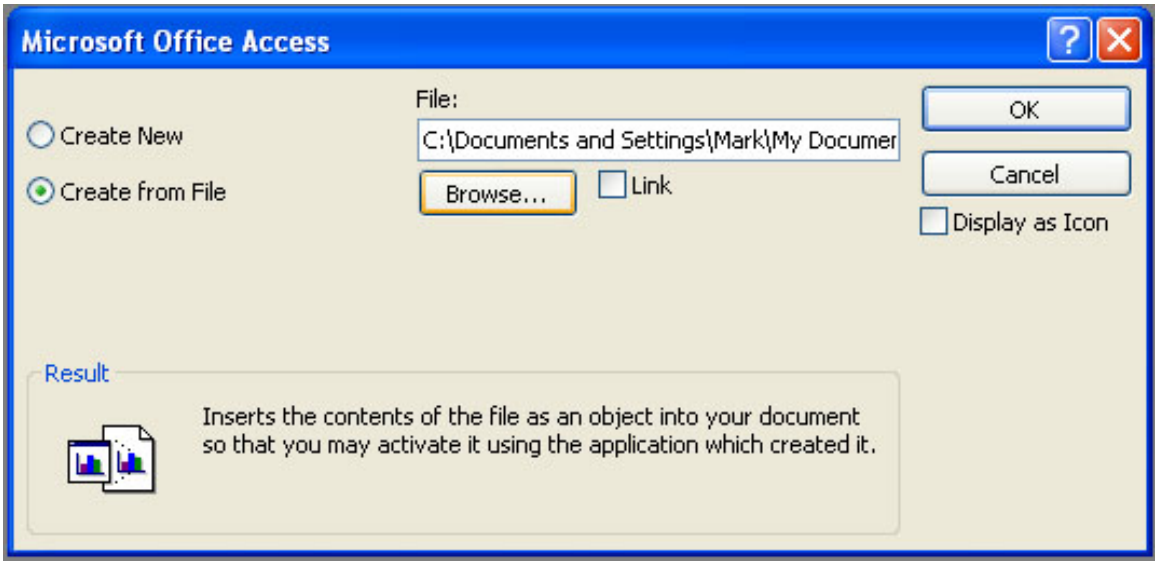
As we start entering data, keep the referential integrity rules in mind. You can't start entering transactions until you have a customer, a broker and a valid stock abbreviation. As you enter data into the Customer and Broker tables, notice how First Names are automatically capitalized correctly, notice how both the Customer ID and Broker ID have the proper starting letter. As you enter a transaction, try to violate our validation rules – for example, try selling to a guy from Vermont.

4.14 Linking to Our Customer Photos

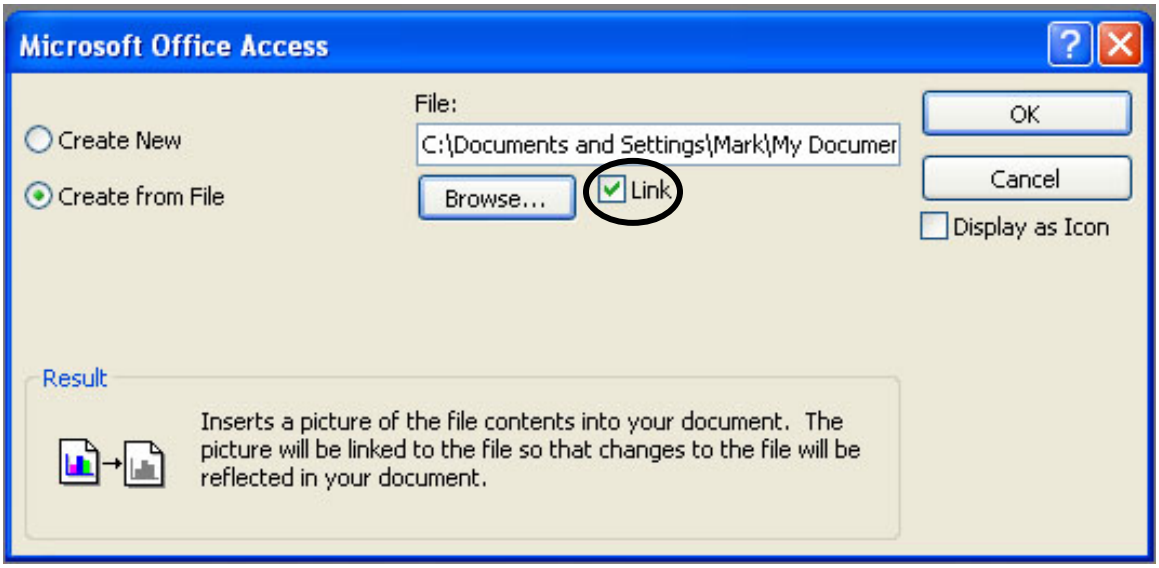
How are we going to enter a photo? You can't exactly enter it the way you would a name or number. I assume the images are located somewhere on your hard drive. Begin by opening the tblCustomer table and right clicking on the Photo field for one of our customers. That should open a context menu with ***Insert Object*** as one of our choices.



We now have a choice – we can create a new object or use one that already exists as a file. In our case, the images are already stored on our hard drive, so we will select **Create from File** and then **Browse** for the photo of that customer.



Finally, we need to establish a Link to that image, so check the Link box.



Is this the ideal approach to using photos in the database? It is for us…because it is very simple and pretty much foolproof. There is, however, a very heavy toll for that simplicity – every image comes with components that help Access display the image. This is going to have a big impact on the size of our database. Professional database developers would store the images on the hard drive, but the database would only have references to them (not Ole Links). This approach keeps the database from growing as each photo is added – but you need to understand quite a bit about computer code to get this more efficient approach to work.

If you have an interest in writing computer code, check Microsoft's web site for suggestions on writing the necessary code. The Knowledge Base Article is # 285820 – ***How to display images from a folder in a form, a report, or data access page***

NOTE: Images are discussed in the Geeks for Chapter 5. For now, realize that images can be stored in a number of file formats. Files with "BMP" extensions are huge, but work well as ole objects. Files with "JPG" extensions are in a compressed format and much smaller than bmp files – **unfortunately, you must have Microsoft Photo Editor installed on your computer in order for the ole links to work.** Photo Editor was included with Windows until Windows XP Service Pack 2, when Microsoft ceased including it. If you can install Photo Editor, you will have no problems including images as part of your database!

4.15 Summary

Could the system use a few more refinements? I guess so, but we've made quite a bit of progress and have put some safeguards in place to help ensure the quality of our data.

We've made a pretty good start on the Brokerage database, starting with a review of Entity Relationship Diagrams – at least the concept of ERDs and how it relates to table design. Entity Relationship Diagrams were introduced in the "Geeks" of Chapter 2. View the ERDs as a way to focus your thinking about your database at a fairly high level. Too many people create poorly designed databases, simply because they move immediately into entering field names and lengths into their tables – before they even know what tables they might need!

Much of the work in this chapter has built upon simpler databases developed in Chapters 2 and 3. In chapter 4 we began to refine our table design by creating validation rules that monitored the quality of data being entered. These rules provided a way to make sure customers can only come from states where we are licensed to sell stocks. The use of Input Masks helped us provide a consistency to data that might not have been there without the masks. Those masks also provided a mechanism to enforce simple business rules – we require an area code for all phone numbers. Finally, we used lookups as one more means of reducing errors in our data. Choosing a stock abbreviation from a list of valid abbreviations makes a lot more sense than letting the brokers take a stab at the abbreviation on their own! Finally, referential integrity provided a way to ensure that a customer and stock existed before we recorded a transaction.

The quality of our data has benefited from a few simple techniques:

Input Masks	Provide the user with a guide to how we want the data to look – forcing capitalization of names, requiring full telephone numbers and insisting that Customer and Broker IDs start with the appropriate letter
Validation Rules	Once the user enters data, we can have Access look it over to determine whether it follows the rules we've set up. In our case, the State must be MA, CT or RI
Lookup Values	We can make sure there aren't data entry problems by getting values from another table. In our case, Stock Abbreviations could be a major problem if we left it up to our Brokers to remember each of the thousands of stocks we can select from.
Referential Integrity	It would be a disaster if we recorded a stock purchase and had listed an invalid Customer ID – someone who didn't exist. Referential Integrity insists that the "one side" of each relationship actually exist.

Just for Geeks
Refining our Database Design Skills
(an Introduction to Normalization)

Many of us design databases by the seat of our pants – hoping the decisions we make for tables, fields and keys will create a functional database. Sometimes our intuitive design works well and sometimes we find ourselves facing a huge mess. The database might not handle the queries we had planned on, and might not create the forms and reports our boss wanted. Mistakes with the design can also lead to real hassles when we try to add, modify or delete data.

Unfortunately, it's sometimes hard to catch design errors until the database is populated with some sample data – once real data starts arriving, a re-design of the database can be expensive and disruptive to the business. It could also be an opportunity for the designers to sharpen their resumes… Luckily database researchers have considered the process of database design and have some suggestions that can help.

The design process is called ***normalization***. It takes you through a series of steps that gradually move your data into a design that reduces data redundancy (repeated data) and that organizes data into tables with related data. The intuitive 'rules' we established in Chapter 4 are very similar to the rules of normalization. We argued that a table should

contain data related to a single entity. For example, the Customer table should only contain data related to a customer – their name, address and phone. The Customer table is certainly not a place to be listing stock purchases. We also argued that a table should not include items that repeat. If a CD has many songs on it, the songs should land in another table.

The common sense rules of Chapter 4 have been formalized by database researchers as a series of ***normalization*** steps. Each stage in the process is called a 'normal form'. The first stage is "First Normal Form" and the last stage is often "Third Normal Form". Yes, you can go to higher forms – but it isn't usually required. Since the stages build upon each other, a database that is in 2nd Normal Form must already be in 1st Normal Form.

First Normal Form

Let's use our Brokerage database as we move through the normalization process. Our first job is getting the data into First Normal Form (often written 1NF). In 1NF, there are two goals:

There should be a primary key that uniquely identifies an entity (entity in the sense of an object such as a customer, a department or a line on sales receipt). Equally important, every column of data (fields) should relate to that primary key. If I know the key, can I tell you who that customer is, where they live and who their broker is?

A second goal of First Normal Form is to achieve ***atomicity*** – the term *atomic* refers to something that can't be divided. In database terms, atomic means that a field should contain data that can't be further subdivided and that there should be only one instance of that data on each row.

Take a look at this early attempt to arrange data in the Brokerage database. I've got a CustID, the customer's name and fields to store the name of each stock they own.

CustID	CustName	Stock1	Stock2	Stock3
111	Bob Smith	IBM,100 shares $30, $32	APPLE, 200 shares $45,$39	APC, 100 shares $60,$63
222	Alan Davis	DELL, 200 shares $65, $67	INTEL, 100 shares $80, $86	APPLE, 500 shares $40, $39
333	Duke Buckley	APPLE, 100 shares $37, $39	IBM, 500 shares $35, $32	INTEL, 200 shares $82, $86
444	Susan Pride	DELL, 100 shares $66, $67	APC, 100 shares $59, $63	IBM, 1000 shares $27, $32

The rules for First Normal Form tell us to watch for fields that aren't atomic. Looking at the CustName field you can see that it holds both first and last names. This could be a problem if we ever need to sort the data… how would you sort by last name? Since each field should hold only one piece of data, let's divide the name field into two fields Fname and Lname.

CustID	FName	LName	Stock1	Stock2	Stock3
111	Bob	Smith	IBM,100 shares $30, $32	APPLE, 200 shares $45,$39	APC, 100 shares $60,$63
222	Alan	Davis	DELL, 200 shares $65, $67	INTEL, 100 shares $80, $86	APPLE, 500 shares $40, $39
333	Duke	Buckley	APPLE, 100 shares $37, $39	IBM, 500 shares $35, $32	INTEL, 200 shares $82, $86
444	Susan	Pride	DELL, 100 shares $66, $67	APC, 100 shares $59, $63	IBM, 1000 shares $27, $32

The other atomicity issue relates to the repeating Stock field. Clearly we have the same problem we had with the Name field. The Stock field should not contain anything more than the name of the stock. Our attempt to record the number of shares, the purchase price and current price might make sense – but it violates the atomicity rule.

The other issue is that we have a different Stock field for each stock a customer buys – sure each field has a slightly different name, but we know the data is really the same. This arrangement is exactly what I might do if this were an Excel spreadsheet. In fact, I do have a spreadsheet to store my grades and I have columns labeled Exam1, Exam2 and Exam3. I also have columns for each assignment. It works for simple spreadsheets but makes life difficult when the data is in a database. Imagine searching for people who own IBM stock. Where would you look? Should your query search Stock1, Stock2 or Stock3? The arrangement in this early version of the table also makes poor use of space – some people fill the three stock fields while other people have only one or two stocks. What happens when our best customer decides to buy two more stocks? Do we need to modify the database by creating two new fields (Stock4 and Stock5)? The answer is that these repeating fields are bad news and need to be eliminated. Atomicity says we should only have ONE field named Stock and that the Stock field should contain only ONE piece of data.

Here's what the table looks like so far. Notice that I've created separate fields for Shares, Purchase Date, Purchase Price and Current Price. I've also beefed it up a bit with more details about the customer.

CustID	FName	Lname	Street	City	State	Zip	Stock	Shares	PDate	PPrice	CPrice

Before we leave First Normal Form, recall that there are two requirements.

1. Each row must have a primary key that uniquely identifies that row of data
2. Each field must contain a single piece of data and the field must not repeat

Selecting a Primary Key

We've taken care of the second requirement, but our table doesn't have a primary key yet. You might be tempted to use CustID as the key – but does that help us determine the stock they bought? Our Brokerage firm hopes each customer will buy many stocks, so simply knowing the CustID will not help us identify the stock. Customer 1234 will have purchased IBM, APPLE and other stocks – knowing just the CustID doesn't determine which stock they bought this time.

If we knew the Customer AND the Purchase Date we still don't have enough to determine what they bought since they might have bought several stocks that day (actually the Date that Access records includes the time down to a fraction of a second – so Date would be a great timestamp for the transaction – but for the sake of our discussion, assume it is just the date!).

If we know the CustID, the Date AND the stock they bought, then we would be able to determine the Customer, how many shares they bought, the purchase price and (knowing the Stock) the Current Price. So, let's go with a composite key (for now) that includes CustID, Stock and PurchaseDate.

That satisfies the requirements for First Normal Form – but leaves our database a long way from being a perfect design.

Second Normal Form (2NF)

Second normal form ONLY APPLIES TO TABLES WITH COMPOSITE KEYS!!! If your table has a single field acting as the primary key, then you are OK at this point. Since our table has three fields combining to form the primary key, we need to follow the rules for Second Normal Form.

The 2NF Rule requires that each field be dependent on the ENTIRE primary key. If you only have one field as your primary key, relax… For the rest of us, we need to examine each field and ask ourselves whether we need the entire primary key to know the value of the field we are examining – or could we determine the value of the field by knowing just a piece of the primary key. Dependencies that require only a piece of the key are called ***Partial Dependencies***.

Let's check out a few fields to see if we have any partial dependencies. I've shaded the primary key for you.

CustID	PDate	Stock	FName	LName	Street	City	State	Zip	Shares	PPrice	CPrice

Let's start with FName… could you know the person's First Name is you knew the CustID? Well, yes…if we have the CustID, we should be able to identify a customer's name and address. Could we learn the name of the customer if we knew the Purchase Date? How about learning the customer's name and address if we had the name of the Stock? In each of these cases, the customer's name and address are dependent only upon the CustID – and not dependent upon the other components of the primary key. You can see the same partial dependencies when you test Purchase Price, Shares and Current Price. These fields have nothing to do with the CustID.

How do we eliminate the partial dependencies? The standard approach is to list the parts of the key as well as the key in its "composite" state. In our case, we would get:

CustID
PDate
Stock
CustID/PDate/Stock

The second step is to find all the fields that depend solely upon CustID, and those fields that depend on PDate, Stock and finally those fields that require the entire composite key.

We might end up with something like this:

CustID ➔ FName, LName, Street, City, State, Zip

PDate ➔ ?????

Stock ➔ CPrice

CustID + PDate + Stock ➔ Shares, PPrice

Each element of the key that was able to "collect" dependent fields will become a separate table. PDate is crucial to identifying a stock transaction, but nothing depends solely upon the Purchase Date – so PDate will NOT become its own table.

Once the partial dependencies have been eliminated, we have our data in Second Normal Form!

Notice how we are forced to create additional tables as the database becomes more normalized. These tables will also need to be joined together at some point. Working with multiple tables and the processing required to join tables (in order to generate forms, reports and queries) will tend to slow things down. Because of the increase in tables and the overhead of joins, some database systems stay somewhat un-normalized in places in order to enhance performance.

Third Normal Form (3NF)

Third Normal Form is focused on finding "***transitive dependencies***". Transitive Dependency sounds pretty fancy, but is actually very simple. It is similar to partial dependencies, but this time, the question is "do any fields depend on any other fields? We aren't talking about fields that depend on keys.

Let me create a situation to illustrate these transitive dependencies. Let's say that our brokerage has three classes of customer. Our "Premier" customers pay a yearly fee to have access to more timely stock quotes, enhanced stock research reports and they also get a substantial break on the transaction fees. Our "Classic" customers pay less than the Premier customers. Their yearly fee requires a higher transaction fee, delays stock prices by 15 minutes and provides brief stock research reports from our analysts. Classic customers must pay an additional fee for each in-depth report they want. Our third category of customers are the "Visitors". We charge a fairly substantial transaction fee and provide very basic reports on each stock. The advanced research reports are unavailable at any fee.

Our Customer table now appears as:

CustID	FName	LName	City	State	Zip	CustClass	TransFee

Let's examine the dependencies. CustClass (Customer classification…Premier, Classic, Visitor) depends on the CustID. If I know the CustID, I can tell you the Customer Class they are in. The problem comes with Transaction Fee (TransFee). The TransFee depends on the CustClass – not the CustID. That means that the TransFee has a "transitive dependency" on CustClass. To get our data into Third Normal Form, this transitive dependency must be eliminated by creating a new table – with the ***determining field*** (CustClass) as the primary key.

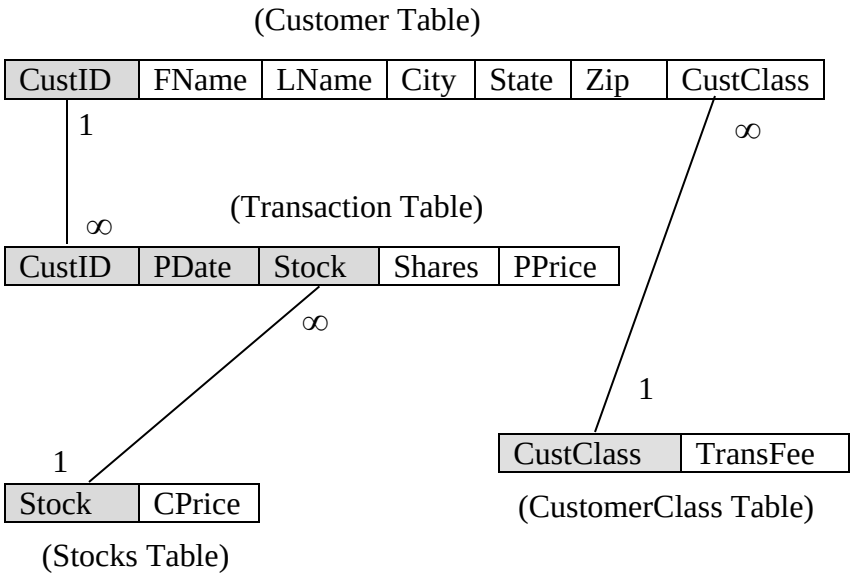
We could call the new table *tblClasses*. It would include both CustClass (the Primary key) and TransFee.

CustClass	TransFee

In order for us to link a customer to the tblClasses table they must share a common field. That requires us to leave CustClass in the Customer table.

CustID	FName	LName	City	State	Zip	CustClass

Before we leave this Geeks issue, let's draw each table and show how they relate to each other. It may seem odd to see Stock on the many side of the relationship for the table with CustID + PDate + Stock as the composite key – but remember, the combination of the three will be unique; the individual elements of the key CAN repeat.



The process of normalizing tables can take a while to get the hang of, but is well worth the effort. Normalization helps move your data into an organization that leads to fewer anomalies – that is, fewer weird things can go wrong when adding, deleting or modifying data. It also provides a design that is more likely to generate the forms, reports and queries your database is expected to handle.

Key Terms

Primary Key
Foreign Key
Composite Key
Autonumber
Referential Integrity
Cascading Deletes and Updates
Input Masks
Validation Rules
Validation Text
Default
Status bar
List Box
Combo Box
Junction Table
Bridge table
Linking Table
Functions
Arguments
Object Linking and Embedding (OLE)

Review Questions:

1. Imagine a table that holds the names of individual stores within a chain of stores. The Stores table has a primary key of StoreID. A second table lists the employees. Each employee only works for a single store. The Employee table includes both the EmpID and the StoreID. In the Employee table, StoreID is playing the role of…
   a) Primary key
   b) Foreign key
   c) Composite key
   d) Junction key

2. For the Stores/Employees database (question #1), Referential Integrity for the relationship between Employees and Stores would say
   a) that a store had to exist in order for an employee to be assigned to it
   b) that each employee had to be assigned to only one store
   c) that a store could have many employees
   d) that an employee could be assigned to many stores

3. Validation Rules are used to
   a) verify that your database name is valid
   b) check field lengths for valid sizes
   c) verify that field names are not longer than 64 characters
   d) test input data for specific values

4. When defining your tables, messages written in the Description area will
   a) show up on the Status bar during data entry
   b) appear only when data entry errors are made
   c) display only in design mode
   d) are strictly for documentation purposes

5. The Status bar is located
   a) at the very bottom of the screen
   b) at the top of the screen
   c) next to every field name when you are in design mode
   d) just under the toolbar

6. A company has a database with employees mentioned in many tables. When an employee leaves the firm, the database administrator wants that employee to disappear from each of the other tables, once they have been deleted from the Employee table. The easiest approach would be to use the
   a) *'related deletes'* feature
   b) *'cascading deletes'* feature
   c) *'associated deletes'* feature
   d) *'correlated deletes'* feature

7. Which input mask requires four characters and converts them all to uppercase?
   a) >LLLL
   b) <LLLL
   c) >????
   d) <????

8. Gatehouse Manufacturing requires all part numbers manufactured in-house to start with a "G" followed by up to 4 characters. The appropriate input mask would be
   a) G????
   b) /G????
   c) \G????
   d) !G????

9. If Gatehouse Manufacturing wanted that "G" to land in their tables, the mask would need to end with
   a) REQ
   b) ;
   c) ;0
   d) :R

10. Database designers are often faced with many-to-many relationships. For example, a baseball player can play at many positions and each position can be played by many players. These many-to-many relationships can be eliminated by
    a) adding a junction table that breaks the M:N into two 1:M relationships
    b) saying "no" when asked to engage *referential integrity*
    c) creating a validation rule stating which positions a player can play
    d) establishing a lookup table that limits a player to one position

Design Exercises

Draw an Entity-Relationship Diagram for the following two databases. Begin by identifying the entities (objects) in each system. Remember, these entities typically become your tables. Be sure to look for ways to relate the entities to each other. Label the related tables as **one-to-many** or **many-to-many**. Indicate the primary keys and use these to relate your tables. Once you've settled on the specific tables, you can add attributes to them (that is, what fields should be in which tables). You may need to add tables (that is, you might need a ***junction table*** for one or both examples).

1. A local college wants to establish a database to store data about each **student** and data about each **section** of a course. They also need a way of listing the students who are registered for each section.

2. Barton Community College is trying to help their commuter students establish friendships with fellow students. Their thinking is that a student is more likely to stay in school if the student feels connected to the school in some way. They have decided to offer $20 bookstore coupons to students who are actively involved in club activities. To monitor club activity, the Director of Activities has decided to establish a small database to store data about each student who has joined a club as well as storing data about each club. Once they have the list of students and the list of clubs, the director wants a way of tracking the club activities each student participates in. So, the Drama club runs a fund raising car wash, the club will be required to record who helped out.

Exercises

1. Create a database that tracks airline fights as well as the passengers having tickets on each flight. You should name the database **Airline** – be sure to keep it on your disk, as it will be used again in Chapter 5 to create Forms, Chapter 6 to create Reports, Chapter 7 to play with Queries and then in Chapter 8 to practice with SQL statements. Once you have a web site built for the airline, we will create a Web Form to let passengers search for flights and another form that will let airline personnel get passenger lists for particular flights.

   In this chapter you will establish the tables and their relationships. As you build the tables, there will be opportunities to create input masks, validation rules and lookup lists.

   There will be two tables in this simple database. The Flights table will include only data regarding the flights. A flight number is unique – once used, we never use that number again. Here's a data dictionary to help you create the table:

Table: *tblFlights*			
Field	***Data Type***	***Length***	***Mask/Validation Rules***
Fnum	Text	5	Let it start with an "F" followed by 4 digits
Depart	Text	15	Should be ALL CAPS
Destin	Text	15	Should be ALL CAPS
DateDep	Date/Time		*Defaults* to today's Date on data entry
Capacity	Number		None of our planes can carry more than 100 people

   The Passenger table lists the passenger's name, the flight they are on, the price of their ticket (we don't all pay the same price for a seat on a plane – airlines call it ***yield management***). Each passenger is assigned a unique PassengerID. Our company advertises a commitment to low prices – we say that tickets are always in the $75 to $200 range. Be sure the TicketPrice gets checked for any input errors; the data should always be in our price range. The ticket agent needs to receive an error message if they enter an invalid price.

Table: *tblPasseng*			
Field	***Data Type***	***Length***	***Mask/Validation Rules***
PassID	Autonumber		
Fnum	Text	5	Look this value up in the Flights table
Pfname	Text	15	Start with a Cap, then shift to lowercase
Plname	Text	15	
TicketPrice	Currency		Needs to be within our price range.

Issues with the Lookup: (Before testing the Lookup, you should enter a few flights into the Flights table.) Lookup lists normally include a single field. In our case, the first field in the Flights table is Fnum. That's actually the value we want to look up – but it will be useless unless we can also see the Departure and Destination fields and the date of the flight. As you establish the lookup, notice the entry for number of columns. Since you want to see more than one field, this value will need to be set to something other than 1. You may need to save the table design and try entering some data so you aren't making changes to the lookup without seeing how they impact the appearance of the lookup. Other properties of the lookup may also need adjustment; in particular, the Field widths and the List width. Start with the Field Widths. Now that you have 4 columns displaying, you can set their widths in inches – with a semicolon between each one. It might look something like this: .3 ; .5 ; .5 ; .3

The list may still not display fully. If it doesn't, try changing the List Width from its current setting "Auto" to some inch measurement.

Once the tables are established, define the relationships – look for common fields. When prompted to create the relationship, be sure to check Referential Integrity…we don't want people listed for flight that don't exist!

Finally – **enter some data**. I've included some test data for you. Most of this is clean data, but some will test your validation rules. For Flight F2222, correct the Capacity to 90 after you check the validation rule.

Sample Data for tblFlights

Test Data for *tblFlights*				
Fnum	***Depart***	***Destin***	***DateDep***	***Capacity***
F1111	Boston	Chicago	9/20/2013	100
F1234	Boston	Chicago	10/15/2013	100
F2222	Providence	Denver	10/15/2013	125 (validation check)
F3333	Boston	Chicago	10/6/2013	80
F4444	Boston	Miami	10/5/2013	75
F5555	Boston	Miami	10/4/2013	100
F6666	Providence	Denver	9/23/2013	100

Sample Data for tblPasseng

Once you've checked your validation rules, correct the prices for passenger 2 to $130 and passenger 3 to $150.

Test Data for *tblPasseng*				
PassID	***Fnum***	***Pfname***	***Plname***	***TicketPrice***
1	F1111	Al	Jones	$90
2	F3333	Sally	Smith	$300
3	F2222	Jack	Adams	$50
4	F1111	Alice	Smayda	$120
5	F3333	Patrick	Woods	$150
6	F2222	Michael	Harland	$175
7	F3333	Johanna	Cuttem	$125
8	F2222	Dick	Sargent	$100
9	F1234	Kathy	McMaster	$125
10	F4444	Dick	Frasier	$140
11	F4444	Chris	Gavin	$120
12	F5555	Sue	Williams	$135

Once you have the tables ready, Print the table designs, the relationship diagram and the data for each table and pass these in to your instructor. A simple way to do this is to display the table design, table data or the relationship diagram in Access, hit your "Print Screen" key (copies the image to the clipboard) and then paste the image into a Word document.