Topic	Background and theory	Core lessons	Advanced material
	1.Preface		
	2.Scenario		
	3.Data modeling		
		4.Introduction to ACCESS	
		5.Tables	
		6.Foreign keys	
		7.Relationships	
		8.Importing and linking	
			9. ODBC
		10.Basic QBE	
		11.Calculated fields	
		12.Basic SQL	
		13.Forms	
		14.Subforms	
		15.Bound controls	
		16.Parameter queries	
		17.Action queries	
		18.Visual Basic	
		19.Event-driven programming	
			20.Shipments
			21.Backorders

Topic	Background and theory	Core lessons	Advanced material
		22.Data warehousing	
		23.OLAP	
		24.HTML	
		25.HTML forms	
		26.Server-side scripting	
		27.Sessions	
		28.ADO	
		29.Business objects	

Lesson 16: Parameter queries

Lesson 17: Action queries

Lesson 18: An introduction to Visual Basic

Lesson 1: Getting started

1.1 Introduction: Why work through these lessons?

The lessons contained in this tutorial are designed to teach you about building information systems to solve practical business problems. Although much of the content relates to MICROSOFT products—and in particular, MICROSOFT ACCESS—the overall pedagogical objectives are much broader than simply teaching you ACCESS. Specifically, the lessons have been designed to:

1. Demystify important information technology concepts that arise in databases, programming, and the Internet.
2. Provide practical guidance on how to model real-world business problems and implement solutions using database technology.
3. Give the you confidence to build simple information systems to solve your own problems.

As a by-product of working through the lessons, you will certainly learn a great deal about using the MICROSOFT ACCESS database package. ACCESS is used because it is cheap (relative to other database systems), powerful enough to be a good teaching tool, and ubiquitous. However, much of what you learn will apply to any relational database system.

Will these tutorials help be become a MICROSOFT CERTIFIED PROFESSIONAL?[1] The short answer is “yes and no”. The long answer is in Section 1.1.3.

1.1.1 Betwixt dummies and developers

These tutorials have emerged from my experience as a teacher of information systems in a business school and also as a day-to-day user of information technology to solve my own problems. Thus, these tutorials address what I think “business people” (broadly defined to include you) need to know about the nuts-and-bolts of technology. The target audience of these lessons is people who are not information technology professionals, but who nonetheless

[1] Those unfamiliar with MICROSOFT’s certification programs may wish to visit the MICROSOFT web site and search under “certification”. The basic concept is that MICROSOFT has successfully incubated a massive third-party training market for its products. To ensure a minimum standard of quality across these programs, MICROSOFT has established curricula and an certification examination process.

need to use information technology to solve non-trivial, real-world problems.

Unfortunately, a conspicuous gap exists between the resources available for "dummies" and those available for "developers". The target audience for the "dummies" books tends to be end-users of a technology—for example, clerical staff who use ACCESS-based applications. Although the dummy books have enormous appeal and are an excellent way to get started, the material tends to focus on mechanical skills, such as formatting, using wizards, and so on. Unfortunately, knowing how to create simple tables and put bold headings on reports will only take you so far.

At the other end of the spectrum are resources targeted at developers—people who make their living writing software and building information systems. The titles of these books often contain terms such as "developer's bible" or "secrets unleashed" and the books themselves run to a thousand pages or so (not including material on the inevitable companion CD-ROM). The problem with these resources is that *we* are all not full-time developers. We have neither the time nor the inclination to (say) use the undocumented features of WINDOWS API to write a control system for a nuclear submarine.

These tutorials are meant fill the gap betwixt dummies and developers. The lessons begin slowly to accommodate people who know nothing about information systems or MICROSOFT ACCESS. As the lessons progress, however, theories and techniques are introduced that enable one to go far beyond a simple user's view of databases.

If you are not being modest and you really know *nothing* about computers, a dummy book may be a prerequisite investment. The dummy books are good at teaching you the basics of using the mouse, resizing and moving windows, and so on. In the lessons that follow, basic computer skills are assumed.

1.1.2 Make versus buy

By the time you finish the final lesson in these tutorials, you will have accomplished the following:

1. **Designed** a relational database for storing information about a small wholesale business.
2. **Built** a graphical application on top of the database to simplify order entry and minimize redundancy and errors.
3. **Transformed** your raw data into a more useful format to support decision-making.

4. **Web-enabled** the business application so customers can place their own orders over the Internet.

In short, these tutorials will teach you how to build a functional business system from scratch. Keep in mind, however, that there are people who devote their entire careers to the art and science of building functioning business systems from scratch. The knowledge you gain here should not be considered a substitute for hiring a professional when it comes time to implement a mission-critical business systems. Instead, the knowledge you gain by working through these tutorials is more appropriately used in one or more of the following ways:

1. **Enable you to build *non*-mission critical systems** — If you want to build a quick-and-dirty system to accomplish Task *X* and the failure to accomplish Task *X* does not endanger your ability to feed your family, by all means, take a crack at it. We learn by doing. If Task *X* is mission critical, however, you should hire a professional.
2. **Permit you to communicate with software professionals** — Good software professionals are hard to find. Good software professionals who also understand the complexities of *your* business problems are virtually impossible to find. Thus, the ability to communicate with your consultants or internal IS staff *in their own language* can yield a massive payoff in the long run. You will get the system you require without being condescended to or exploited to by techno-bullies.
3. **Provide a foundation for becoming an IT professional**—Given the shortage of good software professionals referred to above, there is ample opportunity to become one. These tutorials are a good place to start.

1.1.3 Certification

A deliberate decision has been made to avoid tying these tutorials in with the MICROSOFT certification curricula. There are two reasons for this. First, as stated above, the goal of these tutorials is to teach broad information systems and technology concepts that transcend a single vendor. As such, we are more interested in solving business problems than memorizing keystroke sequences and the intricacies of MICROSOFT's "component object model".

Second, as professional educators, we reserve the right to speak freely and editorialize whenever we feel the urge. That is, if we encounter a particular feature of ACCESS that is dumb, we want the freedom to write: "This is a dumb feature." In other words, we wish to be as unbiased and truthful as we can.[1]

[1] If for some reason Redmond threatens to litigate, we will, of course, cave-in without hesitation.

Naturally, there overlap between the material in this tutorial and any MICROSOFT approved training program (especially the MICROSOFT OFFICE USER SPECIALIST—or MOUS—program). In addition, the concepts we are addressing (e.g., relational databases, programming, web development) are standardized (more or less) so nothing here contradicts the official MICROSOFT training program (if it does, please let us know and we will instruct MICROSOFT to make the necessary corrections). Thus, although these tutorials are not designed to help you cram for a certification program, they certainly will not hinder your preparation.

1.2 Structure of the lessons

The tutorials are organized in the following manner:

1. **Scenario** — You are provided with a business scenario in Lesson 2 to work through. The scenario provides a common thread through all of the lessons and thus the lessons should be considered cumulative (i.e., you should finish Lesson 5 before starting Lesson 6, and so on).
2. **Lessons** — Each lesson consists of different sections:
   a) **Introductory comments and learning objectives** — The opening section of each lesson is meant to give you some idea of what you will accomplish in the lesson and why it is important.
   b) **Tutorial exercises** — The exercise are the meat of the lesson since you learn by doing. All the steps that require action on your part are marked with an arrow (❷).

   If you are an extremely action-oriented learner, you can scan the text for the arrows, follow the instructions, and read the ancillary chit-chat on an as-needed basis.

   c) **Discussion** — During the course of the lesson, technical and theoretical issues may arise. Rather than break of the flow of the exercises with long detailed explanations, you are encouraged to suppress your natural curiosity, work through the exercises, and then read the discussion at the end of the lesson. Hopefully the material in the discussion will fill in the important gaps in your understanding.
   d) **Application to the project** — The general problem that occurs when one works through the steps of a tutorial is that no real learning occurs. In other words, you may remember the sequence of keystrokes and mouse

jiggles required to complete a task, but you cannot generalize the skill to other tasks. In the "application to the project" sections, you are on your own. You are given tasks that you must accomplish before you can move on to the next lesson. If this sounds a lot like work, that is because it is work. No pain, no gain, right?

3. **Background lessons** — We are firm believers that there is nothing more practical than a good theory. Databases and programming languages are based on explicit theories, and unless you understand the critical theoretical principles that underlie these technologies, you are going to find it hard to generalize the skills you learn here to your own problems. Background lessons—such as this one—are intended to fortify your theoretical knowledge. Since there are no exercises in a background lessons, you are encouraged to take your hands off the keyboard and simply read.

1.3 Typographical conventions

The following is a brief list of symbols and typographical conventions used in the tutorials.

1.3.1 Warnings, tricks, and tips

Important warnings are marked with an exclamation mark. It is important that you heed these warnings to avoid common problems.

Other clarifications, recommendations, and trivia are marked with a question mark. These sections are typically time-saving tips or explanations of alternative ways of doing things.

HINT: In the "application to the project" sections, hints to help you complete the steps are indicated with the hint symbol.

1.3.2 Version differences

All the screen shots and videos in these tutorials are taken from Access version 8.0 (released as part of Microsoft Office 97). Although there are some important differences between the various versions of Access (i.e., version 2.0, version 7.0, version 8.0 and Access 2000) the underlying concepts remain the same in all versions.

Whenever the instructions given in the tutorial differ significantly from version 8.0 (Access 97), a warning box such as this is used.

Similarly, the "2" marker indicates that the procedure for version 2.0 differs from the procedure described in the lesson.

This set of tutorials does not cover ACCESS 2000[1]. However, with a few notable exceptions, the differences between ACCESS 8.0 and ACCESS 2000 are cosmetic. As such, it is possible to complete the lessons using ACCESS 2000.

1.3.3 Exercises

1 As discussed above, tutorial exercises are indented and marked with an arrow.

1.3.4 Important terms and hyperlinks

If a term is important, it is marked in **bold text**. If a word in the electronic version of this lesson is a hyperlink to another location, it appears in blue. Clicking on a hyperlink takes to you directly to the new location.

1.3.5 Menu commands

In some cases, you are asked to use the mouse to execute a series of menu commands (e.g., **File** → **Save As**). What this means is that you select **File** from the main menu, followed by **Save As**.

1.3.6 Programming code

In some exercises, you are asked to type in programming code, such as VISUAL BASIC (VB) or STRUCTURED QUERY LANGUAGE (SQL). The code you enter is shown in a `monospaced font`.

1.3.6.1 The "new line" marker

When multiple lines of programming code are shown, the "new line" marker (NL) is used to indicate the start of a new line (you start a new line by pressing the **Enter** key). For example, each of the `set` statements below should be typed on a single line regardless of the line breaks that appear in this document:

```
NL Set dbCurr =
   DBEngine.Workspaces(0).Databases(0)
NL Set rsBackOrders =
   dbCurr.OpenRecordset("BackOrders",
   dbOpenDynaset)
```

Unlike most modern languages, the main programming language you will use to develop your application, VISUAL BASIC FOR APPLICATIONS (VBA), requires each statement to be on its own line. However, the narrow columns used in the lessons result in line wrapping. If you type VBA code in its line-wrapped form, you will

[1] The ACCESS 2000 tutorials are under construction. See 2np.org for details.

encounter errors. Hence the use of the NL marker.

1.3.6.2 New code

When code is added to an existing program, the old code is shown in a lighter typeface whereas the new code is bolded. For example, the third statement below is new:

```
NL  Set dbCurr =
    DBEngine.Workspaces(0).Databases(0)
NL  Set rsBackOrders =
    dbCurr.OpenRecordset("BackOrders",
    dbOpenDynaset)
NL  Set qdf =
    dbCurr.QueryDefs!pqryBackOrderChang
    es
```

Making a distinction between new code and existing code makes it easier for you to find the location of the changes in your own programs much faster.

1.4 Questions, queries comments

Much of the material in these tutorials emerged from student questions. If you find something difficult to understand, or if you encounter a problem in your own work that is not covered in the tutorials, please let us know. We will do our best to continuously grow and upgrade the material.

2.1 Introduction: scenario-based learning

This tutorial package uses a single business problem—order management at a kitchen supply company—to illustrate many different information systems issues. We recognize that it is unlikely that the business problems you face have much in common with wholesaling kitchen gadgets. However, there are important advantages to introducing and sticking with a single scenario.

1. You get to work through a problem from start to finish. You start with the business problem, build a conceptual model, create a physical design that satisfies the requirements, and implement the design in software.
2. By sticking with a single scenario, we do not incur the setup cost of constantly introducing and describing different business environments.
3. Even though the kitchen supply scenario used here is simple, it contains enough complexity to highlight interesting technical and process design issues. Such issues do not generally emerge without looking at a problem in depth.

The purpose of this lesson is to present the business scenario in detail and set the stage for the remainder of the lessons in this tutorial package.

2.2 Business-to-business for little guys

You run a small-but-growing company that imports and distributes kitchen gadgets. Your product line consists of stainless steel utensils, ceramic serving dishes, small electrical appliances, and so on.

Your suppliers are manufacturing firms from around the world and your customers are primarily small retailers such as kitchen specialty stores and hardware stores. In other words, you are a middleman (or middleperson, if you prefer): you aggregate products from a global network of suppliers and market them to other small businesses in your region.

2.2.1 Your "value proposition"

The middleman is generally thought to be an endangered species in the information age. There is a risk that your customers (the mom-and-pop kitchen and hardware stores) will somehow get connected directly to the manufacturers and disintermediate you. That is, they will use the Internet to find your suppliers and submit their orders directly, without the costs and benefits of your participation.

To stay in business, it is important that you provide your customers with something more than simply another layer of cost. Your speciality might be finding unique products. Or you may have nurtured strong relationships with certain suppliers that provide you with preferential access to the newest and hottest products. Either way, one thing is certain: changes in the wholesaling environment mean that you must operate at a very high level of efficiency in order to remain viable.

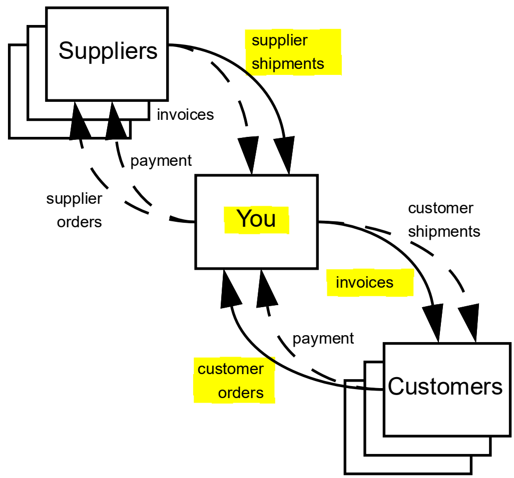
2.2.2 Scope of the project

The essential flows of information between you and your suppliers and customers are shown in Figure 2.1. Basically, you buy goods from your suppliers and keep them in inventory until they are ordered by customers.

To keep the scope of the scenario manageable, we are going to focus exclusively on the information flows shown by solid arrows and ignore everything else. The “everything else” includes:

- information flows indicated by dashed arrows;
- physical flows, such as the goods themselves; and,
- information flows to organizations that are not shown in Figure 2.1, such as banks, trucking companies, and so on.

FIGURE 2.1: The primary flows of information in the kitchen supply environment.



As you will hopefully discover, developers of information systems use the same tricks over and over. Thus, if you have the skills and knowledge to create a simple order entry system, you can use the same skills and knowledge to add additional functionality, such as tracking accounts receivable, managing backorders, reconciling invoices with shipping notices, and so on. The objective here is not to create a realistic order management system; rather, it is to introduce just enough complexity to keep the lessons interesting.

2.2.3 Business processes

In the following sections the critical business process and information flows in Figure 2.1 are discussed in greater detail.

2.2.3.1 Customer orders

From time to time, a customer (or a member of your small sales force) faxes you an order for products. For each item in the order, you check inventory (to determine the quantity you have in stock), decide how many of each item to ship (you cannot ship what you do not have), and verify the price of the product (the person who submitted the order might be using out-of-date prices).

Once you have determined the correct price and quantity to ship for each item in the order, you create an invoice. The people in your warehouse use the invoice to determine what to ship to the customer. The final invoice is included in the physical shipment to the customer and the customer is expected to pay the amount on the invoice in accordance with your payment terms.

2.2.3.2 Backorders

A backorder occurs whenever a customer orders a product that you currently do not have in stock. At this early stage of the tutorials, we are simply going to ignore backorders. Later on, in Lesson 21, you will add the capability to manage backorders to your application.

2.2.3.3 Supplier shipments

Since you are a middleman, it is natural that the fulfillment processes on the outbound (customer) side are mirrored by similar business processes on the inbound (supplier) side. In order to replenish your own inventory of products, you submit orders to your suppliers, and receive shipments and invoices in response. At this point, we are going to ignore the inbound side of the business and focus on customer orders. In Lesson 20 you will add the capability to track incoming supplier shipments to your application.

2.2.4 Your existing information system

Your current system for managing the information flows in Figure 2.1 is largely based on spreadsheets, paper records, and manual effort.

For example, you currently keep track of all your customers, orders, and inventory using different spreadsheets. Although this approach works reasonably well, there are at least three major problems:

1. You create all your invoices by cutting and pasting from your orders spreadsheet into a specialized invoice spreadsheet. This is error prone and tedious.
2. You and your employees sometimes make errors entering data and thus the values in your inventory spreadsheet are suspect. You spend a great deal of time verifying inventory levels by physically counting items.
3. You find it very difficult to make business sense of all the data in your spreadsheets. For example, calculating the sales-per-customer for your hottest selling products typically takes you most of an afternoon. And you have to repeat the whole process every quarter.
4. Words that are *not* currently in your vocabulary include: backup, transaction atomicity, auditability, and referential integrity. In short, you have a nagging suspicion that your current system is less sophisticated than it should be given the important role it plays in your business.

2.2.5 A wish list

You are interested in building a small transaction processing system that will provide the following core functionality:

- allow orders to be entered into the computer as they are received from customers and salespeople,
- track inventory levels of each SKU[1] as product is shipped to your customers
- automatically generate customer invoices.

There are other functions—such as recording shipments from your suppliers, managing back orders, and flagging items to reorder—that you want to add later. In addition, you are investigating the possibility of providing direct ordering via the Internet for certain customers. However, you prefer to move slowly since you do not have a great deal of experience with database software or information systems generally.

---

[1] Firms typically assign a number to each product they keep in stock and thus you may hear the term SKU (pronounced “skew”) to refer the “stock keeping units.” In the scenario used here, you are dealing with a very small number of SKUs.

You have decided to use MICROSOFT ACCESS to develop your application. Although you examined other packages, you chose ACCESS because it was reasonably priced (it was bundled in an office suite) and because the reviews you read were favorable.

Ideally, selection of a software package would have come after an in-depth requirements analysis. However, at this point, your objective is to build a prototype system to learn more about the benefits of a database system. Down the road, your plan is either to hire a professional developer to clean-up and extend your prototype application or buy an off-the-shelf package. You are certain that whichever path you follow, the experience you gain implementing the prototype system will enable you to make much better decisions regarding the use of technology to support the order management process in your company.

2.3 Project package

To help you build a prototype system, you are provided with the following information and documents:

1. **A complete inventory** — The current quantity on hand information for each product you carry has been stored in a plain text file. The file can be found in the project package as package\inventor.txt.
2. **Sales orders** — Your customers and sales representatives fax you sales orders every couple of days (orders are normally filled and invoiced in the sequence that they arrive). Electronic copies of these orders available in project package as package\orders.pdf.
3. **Current backorders and shipments** — The project package also contains a list of backordered products and details of a small number of shipments from your suppliers. You will not require these files until you extend the functionality of your system in Lesson 20 and Lesson 21.

2.4 Discussion: automate and informate

A distinction is sometimes made between the use of technology to automate and the use of technology to informate.[1] The distinction between the two concepts can be applied to the kitchen supply scenario to refine our thinking about the role of a database in the context of the business problem.

1 The automate versus informate distinction was introduced by Shoshana Zuboff in her 1988 book: *In the age of the smart machine: the future of work and power*.

2.4.1 Automating the business

Looking up inventory levels and generating invoices are both menial tasks that should be automated. Your value to your customers lies in your ability to maintain relationships with your suppliers and find interesting products that people want to buy. Every minute that you spend cobbling together an invoice or tracking down a backorder is a lost opportunity to do what you are supposed to be doing.

The concept of automating a process using a computer is straightforward enough. The difficulty lies in implementation—making the automated process work the way it is supposed to work. As discussed in Lesson 3, we are going to start with a data-centric view of the operation and add automation incrementally. A useful way to think about automation is:

1. What needs to be done? What are the steps in the process?
2. What data is required? For example, to know how much of a particular product to ship to a customer, you need to know how much the customer has ordered and how much you have on hand. Other processes have more complex data requirements. A simple rule of thumb is: if you do not have the data, you cannot automate the process.
3. When should the automated process occur? Is the process triggered automatically or manually? Does the process occur in response to the passage of time (such as a monthly closing) or in response to some business event (such as a stockout)?

The critical issues in automation are not which programming language you use or what type of computer you are running. Instead, what is important is an understanding of what needs to be done and when from a business perspective.

2.4.2 Informating the business process

Two things happen when you process a customer's order:

1. you execute the transaction requested by the customer, and
2. you learn something about what the customer wants.

Many organizations focus on execution—that is, on automation of business processes. However, it is important to recognize that every time you process a transaction in your day-to-day operations, you create information about your business.

For example, if you process a large number of customer orders and store the information about the orders in an appropriate manner, it is possible to learn new and important things from this data: Who buys what? When do they buy it? And so on. This information can enable you to operate more efficiently and provide a better level of customer service.

The problem is that although many people talk about the benefits of informating the organization, it is not always clear how one should go about achieving these benefits. This is especially true in small organizations.

Throughout the lessons that follow, you should step back from the nitty-gritty details of implementation from time to time and situate your progress with respect to the underlying business problem. Are you automating a tedious task? Are you storing data that can be used to support decision-making?

2.5 Application to the project

1. Ensure you understand the business scenario described in this lesson and the flows of information shown in Figure 2.1.

Lesson 3: An introduction to data modeling

3.1 Introduction: The importance of conceptual models

Before you sit down in front of the keyboard and start creating a database application, it is critical that you take a step back and consider your business problem—in this case, the kitchen supply scenario presented in Lesson 2— from a *conceptual* point of view. To facilitate this process, a number of **conceptual modeling** techniques have been developed by computer scientists, psychologists, and consultants.

For our purposes, we can think of a conceptual model as a *picture* of the information system we are going to build. To use an analogy, conceptual models are to information systems what blueprints are to buildings.

There are many different conceptual modeling techniques used in practice. Each technique uses a different set of symbols and may focus on a different part of the problem (e.g., data, processes, information flows, objects, and so on). Despite differences in notation and focus, however, the underlying rationale for conceptual modeling techniques is always the same: understand the problem before you start constructing a solution.

There are two important things to keep in mind when learning about and doing data modeling:

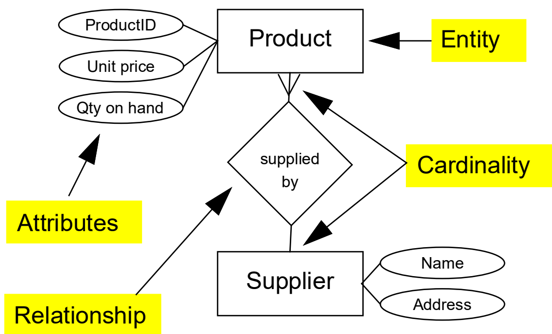
1. Data modeling is first and foremost *a tool for communication*.Their is no single "right" model. Instead, a valuable model highlights tricky issues, allows users, designers, and implementors to discuss the issues using the same vocabulary, and leads to better design decisions.
2. The modeling process is inherently iterative: you create a model, check its assumptions with users, make the necessary changes, and repeat the cycle until you are sure you understand the critical issues.

In this background lesson, you are going to use a **data modeling** technique—specifically, **Entity-Relationship Diagrams** (ERDs)—to model the business scenario from Lesson 2. The data model you create in this lesson will form the foundation of the database that you use throughout the remaining lessons.

3.1.1 What is data modeling?

A data model is a simply a diagram that describes the most important "things" in your business environment from a data-centric point of view. To illustrate, consider the simple ERD shown in Figure 3.1. The purpose of the diagram is to describe the relationship between the data stored about products and the data stored about the organizations that supply the products.

FIGURE 3.1: An ERD showing a relationship between products and suppliers.



3.1.1.1 Entities and attributes

The rectangles in Figure 3.1 are called **entity types** (typically shortened to "entities") and the ovals are called **attributes**. The entities are the "things" in the business environment about which we want to store data. The attributes provide us with a means of organizing and structuring the data. For example, we need to store certain information about the products that we sell, such as the typical selling price of the product ("Unit price") and the quantity of the product currently in inventory ("Qty on hand"). These pieces of data are attributes of the Product entity.

It is important to note that the precise manner in which data are *used* and *processed* within a particular business application is a separate issue from data modeling. For example, the data model says nothing about *how* the value of "Qty on hand" is changed over time. The focus in data modeling is on capturing data about the environment. You will learn how to change this data (e.g., process orders so that the inventory values are updated) once you have mastered the art of database design.

A data modeler assumes that if the right data is available, the other elements of the application will fall into place effortlessly and wonderfully. For now, this is a good working assumption.

3.1.1.2 Notation for relationships

In addition to entities and attributes, Figure 3.1 shows a **relationship** between the two entities using a line and a diamond. The relationship construct is used—not surprisingly—to indicate the existence or absence of a relationship between entities. A crow's foot at either end of a relationship line is used to denote the cardinality of the relationship.

For example, the crow's foot on the product side of the relationship in Figure 3.1 indicates that a particular supplier may provide your company with several different products, such as bowls, spatulas, wire whisks and so on. The *absence* of a crow's foot on the supplier side indicates that each product in your inventory is provided by a single supplier. Thus, the relationship in Figure 3.1 indicates that you always buy all your wire whisks from the same company.

3.1.1.3 Modeling assumptions

The relationship shown in Figure 3.1 is called one-to-many: each supplier supplies many products (where many means "any number including zero") but each product is supplied by one supplier (where "one" means "at most one").

The decision to use a one-to-many relationship reflects an assumption about the business environment in which your wholesale company operates. However, it is easy to imagine a different environment in which each product is supplied by *multiple* suppliers. For example, many suppliers may carry a particular brand of wire whisk. When you run out of whisks, it is up to you to decide where to place your order. In other words, it is possible that a **many-to-many** relationship exists between suppliers and products.

If multiple supplier exist, attributes of the product, such as its price and product number may vary from supplier to supplier. In this situation, the data requirements of a many-to-many environment are slightly more complex than those of the one-to-many environment. If you design and implement your database around the one-to-many assumption but then discover that certain goods are supplied by multiple suppliers, much effort is going to be required to fix the problem.

Herein lies the point of drawing an ERD: The diagram makes your assumptions about the relationships within a particular business environment explicit *before* you start building things.

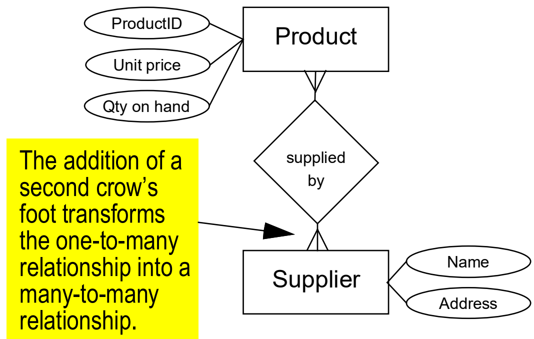
3.1.1.4 The role of the modeler

In the environment used in these tutorials, you are the user, the designer, and the implementor of the system. In a more realistic environment, however, these roles are played by different individuals (or groups) with different backgrounds and priorities. For example, a common stereotype of implementors (programmers, database specialists, and so on) is that they seldom leave their cubicles to communicate with end-users of the software they are writing. Similarly, it is generally safe to assume that users have no interest in, or understanding of, low-level technical details (such as the cardinality of relationships on ERDs, mechanisms to enforce referential integrity, and so on). Thus, it is up to the **business analyst** to bridge the communication gap between the different groups involved in the construction, use, and administration of an information system.

As a business analyst (or more generally, a designer), it is critical that you walk through your conceptual models with users and make sure that your modeling assumptions are appropriate. In some cases, you may have to examine sample data from the existing computer-based or manual system to determine whether (for instance) there are any products that are supplied by multiple suppliers.

At the modeling stage, making changes such as converting a one-to-many relationship to a many-to-many relationship is trivial—all that is required is the addition of a crow's foot to one end of the relationship, as shown in Figure 3.2. In contrast, making the same change once you have implemented tables, built a user interface, and written code is a time-consuming and frustrating chore.

FIGURE 3.2: An ERD for an environment in which there is a many-to-many relationship between products and suppliers.



Generally, you can count on the 10× rule of thumb when building software: the cost of making a change increases by an order of magnitude for each stage of the systems development lifecycle that you complete.

3.1.2 Core modeling constructs and notation

Data modelers typically adopt a set of notational conventions so that their diagrams are consistent. For example, large IT organizations and consultancies typically adopt a **methodology**[1]—a set of tools and procedures for applying the tools that specifies the notation used within the organization. Enforcing standardization in this way facilitates teamwork on large projects. Similarly, if a **computer-aided software engineering** (CASE) tool is used for conceptual modeling and design, notational conventions are often enforced by the software.

What follows is a brief summary of the notational conventions that I use when drawing ERDs. Keep in mind, however, that ERDs are first and foremost a tool for communication between humans. As such, the precise notation you use is not particularly important as long as people can read and understand the diagrams. With experience, you will come to realize that differences in the shapes of the boxes and lines have little effect on the core concepts of data modeling.

[1] It can be argued that the term "method" is grammatically preferable. In Europe, for example, the term "method" tends to be favored.

3.1.2.1 Entities

Entities are drawn as rectangular boxes containing a noun in singular form, as shown in Figure 3.3.

FIGURE 3.3: An entity named "Customer".

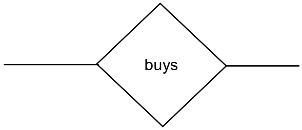
You will see later that each entity you draw ultimately becomes a table in your database. You might want to keep this transformation from entity to table in mind when selecting the names of your entities. For example, your entity names should be short but descriptive.

3.1.2.2 Relationships

A relationship between entities is drawn as a line bisected by a diamond. The diamond contains a verb (or short verb phrase) that describes the nature of the relationship between the entities, as shown in Figure 3.4.

Named relationships are used to make the ERDs more readable. However, unlike entity names, relationship names never show up in the final database. Consequently, it does not really matter how you label your relationships, as long

FIGURE 3.4: A relationship named “buys”.



as the labels make the diagram easier to interpret.

To illustrate, consider the relationship between products and suppliers shown in Figure 3.1. The relationship is described by the verb phrase “supplied by”. Although one could have opted for the shorter relationship name “has” instead, the resulting diagram (e.g., “Supplier has product”) would be more difficult for readers of the diagram to interpret.

3.1.2.3 Relationship direction

One issue that sometimes troubles neophyte data modelers is that the *direction* of the relationship is not made explicit on the diagram. Returning to Figure 3.1, it is obvious to me (since I drew the diagram) that the relationship should be read: “Product is *supplied by* supplier.” Reading the relationship in the other direction (“Supplier *is supplied* by product”) makes very little sense to anyone who is familiar with the particular problem domain.

Generally, ERDs make certain assumptions about the reader’s knowledge of the underlying business domain.

A notational convention supported by some CASE tools is to require two names for each relationship: one that makes sense in one direction (e.g., “is supplied by”), and another that makes sense in the opposite direction (e.g., “supplies”). Although double-naming may make the diagram easier to read, it also adds clutter (twice as many labels) and imposes an additional burden on the modeler.

3.1.2.4 Cardinality

As discussed in Section 3.1.1.2, the cardinality of a relationship constrains the number of instances of one entity type that can be associated with a single instance of the other entity type.

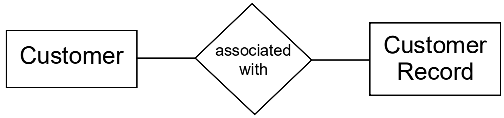
The cardinality of relationships has an important impact on number and structure of the tables in the database. Consequently, it is important to get the cardinality right on paper before starting the implementation.

There are three fundamental types of cardinality in ERDs:

- **One-to-many** — You have already seen an example of a one-to-many relationship in Figure 3.1. You will soon discover that one-to-many relationships are the bread and butter of relational databases.
- **One-to-one** — At this point in your data modeling career, you should avoid one-to-one relationships. To illustrate the basic issue, consider the ERD shown in Figure 3.5. Based on an existing paper-based system, the modeler has assumed that each customer is associated with one "customer record" (i.e., a paper form containing information about the customer, such as address, fax number, and so on). Clearly, each customer has only one customer record and each customer record belongs to a single customer. However, if we automate the system and get rid of the paper form, then there is no reason not to combine the Customer and Customer Record entities into a single entity called Customer.

In many cases, one-to-one relationships indicate a modeling error. When you have a one-to-one relationship such as the one shown in Figure 3.5, you should combine the two entities into a single entity.

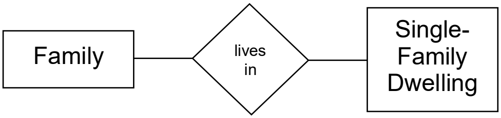
FIGURE 3.5: An incorrect one-to-one relationship



- **Many-to-many** — The world is full of many-to-many relationships. A well-used example is "Student takes course." Many-to-many relationships also arise when you consider the *history* of an entity. To illustrate, consider the ERD shown in Figure 3.6. At first glance, the relationship between Family and Single-Family Dwelling (SFD) might seem to be one-to-one since a particular family can only live in one SFD at a time and each SFD can (by definition) only contain a single family. However, it is possible for a family to live in different houses over time. Similarly, it is possible that many families inhabit a particular house over the years. Thus, if the concept of time is considered, the relationship becomes many-to-many.

We will discuss how you go about determining cardinality in subsequent sections. At this point, it is sufficient to recognize that there are two

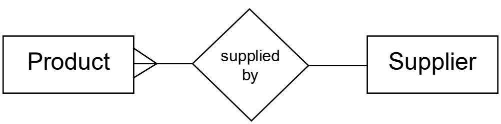
FIGURE 3.6: What is the cardinality of this relationship?



popular (and equivalent) approaches to denoting "one" and "many" on an ERD: the crow's foot notation you have already seen and the "1:N" notation.

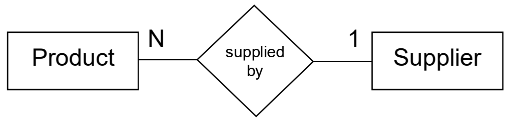
1. **Crow's foot notation** — In the crow's foot notation, three little lines (resembling a crow's foot) are used to indicate "many". Not surprisingly, the absence of a crow's foot indicates "one". Thus, the relationship in Figure 3.7 indicates that "each product is supplied by *at most* one supplier," whereas "each supplier may supply many products.".

FIGURE 3.7: A one-to-many relationship in crow's foot notation



2. **1:N notation** — In the 1:N notation, the symbol "N" (and/or "M") is used to indicate "many" whereas "1" is used to indicate "one". An example of the 1:N notation is shown in Figure 3.8.

FIGURE 3.8: A one-to-many relationship in 1:N notation



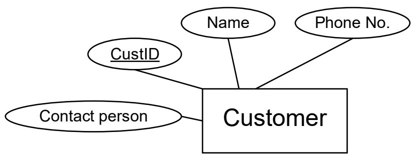
The ERD model also supports additional cardinality information in the form of cardinality constraints. To keep things simple, however, the discussion of cardinality constraints is deferred until Section 6.3.2 on Page 6-6.

3.1.2.5 Attributes

Attributes are properties or characteristics of a particularly entity about which we wish to collect and store data. In addition, there is typically one attribute that uniquely identifies particular instances of the entity. For example, each of your customers may have a unique customer ID. Such attributes are known as **key attributes**.

For ERDs that are drawn manually, attributes are traditionally shown as ovals containing the name of the attribute. If the attribute is a key, it is typically underlined. A number of attributes for the Customer entity are shown in Figure 3.9.

FIGURE 3.9: A number of attributes of the Customer entity are shown in ovals.



Adding attributes to ERDs can result in very cluttered diagrams. Some CASE tools list the attributes inside the entity rectangle (see the discussion of CASE tools in Section 3.4.5). Another way to reduce clutter is to only show a handful of critical attributes on the diagram.

3.2 Learning objectives

- understand the core constructs of the entity-relationship model
- create an ERD based on your understanding of a business scenario
- use associative entities to add attributes to relationships
- gain some familiarity with the role of data modeling and CASE tools in the development process

3.3 Exercises

In the sections that follow, we step through the construction of an ERD for the kitchen supply scenario. By following along, you should gain a better understanding of the basic techniques involved in data modeling as well as some of the design pitfalls that should be avoided.

If this lesson was on how to play golf, you would not read it and then assume that you are a good golfer. Golf is a skill that requires both theoretical knowledge and hours of practice. Thus, the only way to become a good golfer is to acquire a solid understanding of the fundamentals and then go out and hit thousands of balls. The same principles apply to data modeling.

3.3.1 Starting simple

Let us begin with the simplest and most essential statement one can make about the wholesaling environment: *customers buy products*. It is natural that you would want to both automate and informate (recall Section 2.4) this important business process.

3.3.1.1 Step one: identify the entities

Entities are physical things, organizations, roles and events about which we want to store information. In the wholesaling scenario, two entities are immediately obvious: Customer and Product. However, before we add the entities to our diagram, it is important that we have a firm understanding of what *exactly* these entities correspond to in the real world:

1. **Customers** — In the wholesaling environment, a customer is an organization, not a person. There may be a single person at the organization through whom we conduct our business. We will refer to this person as the "contact person" for the customer in order to maintain a clean distinction between people and organizations.
2. **Products** — It is not immediately clear whether the Product entity refers to a specific item or a class of similar items. For example, one of the products you sell is the "Fat Cat" mug. The Product ID of the mug is "88 4017" and it normally sells for $5.50. Note, however, that there are many *individual* "Fat Cat" mugs and each one is slightly different due to irregularities, variations in painting, and so on. In our case, there are advantages to ignoring the individuality of each mug and treating them all as a single group of interchangeable items. Thus, when we talk about "a product" or "a SKU", we are talking about an *entire class* of similar instances, not individual instances themselves.[1]

Having made these assumptions explicitly, we can now create our first ERD.

1 Take out a piece of paper and a pencil.

Unless you have a special-purpose CASE tool, it is seldom worth the effort to draw the early drafts of your conceptual models on a computer.

ERDs typically require many modifications so you should not invest much time making your diagrams look nice. In fact, the diagram you are about to begin will

[1] In some environments, it may be necessary to treat products as individual items. For example, in the aerospace industry, there is a requirement to track individual parts by serial number in case a part fails. The requirement for "unit effectivity" necessitates a different set of assumptions about the Product entity and thus leads to a different database design.

end up the in recycle bin by the end of the lesson.

2 Add the Customer and Product entities to your diagram, as shown in Figure 3.10.

FIGURE 3.10: Add the first two entities to your ERD.

3.3.1.2 Step two: specify a relationship between the entities

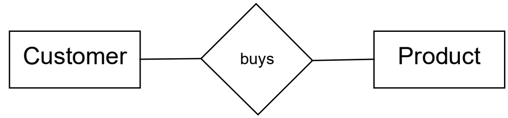
We know that customers buy products and that products are bought by customers. It is a simple matter to create a relationship to communicate this fact.

3 Add a relationship line between the Customer and Product entities.

4 Label the relationship "buys", as shown in Figure 3.11.

Unlike flow charts, the arrangement of boxes and the direction of lines in an ERD have no significance—any arrangement that fits on the page is valid. Similarly,

FIGURE 3.11: Add a relationship between the two entities.



the relationship line does not denote any type of sequence or flow of information.

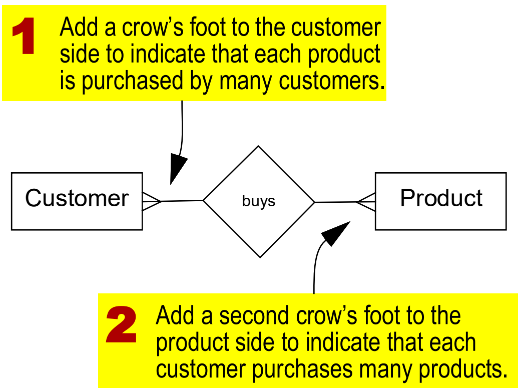
3.3.1.3 Step three: determine the cardinality of the relationship

Each customer can buy many products—indeed, that is the whole purpose of being in this line of business. To show this possibility on our ERD, we add a crow's foot to the Product side of the relationship line.

Similarly, each product can be purchased by many customers. For example, a number of our customers may chose to stock the "Fat Cat" mug (keeping in mind that the product refers to a style of mug, not an individual mug). As a result, a crow's foot is added to the Customer side of the relationship.

5 Designate the "buys" relationship as many-to-many using the crow's foot notation, as shown in Figure 3.12.

FIGURE 3.12: Indicate the many-to-many cardinality of the relationship.



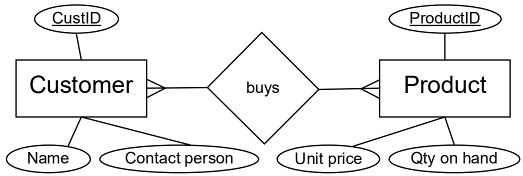
3.3.1.4 Step four: identify a few important attributes

There is a need to find a balance between the descriptiveness of the ERD and the ease with which others can decipher it. As such, I prefer to show only a handful of attributes on the ERDs.

6 Add a small number of important attributes to your diagram, as shown in Figure 3.13.

By adding attributes such as "Qty on hand" to the Product entity, it is clear that the entity refers to a class of products, not an individual product. In contrast, if "Serial number" were added to the diagram as an attribute instead of "Qty on hand", the reader of the ERD would come to a different conclusion about the meaning of the Product entity.

FIGURE 3.13: An initial ERD for the kitchen supply environment.



3.3.2 Dealing with many-to-many relationships

Although the diagram in Figure 3.13 is technically correct, it is missing a great deal of information about how a purchase transaction occurs in reality.

To illustrate, consider the attribute "Unit price" belonging to the Product entity. Unit price contains the *default* selling price of the product. To understand why it is the default price, consider the case of a "Fat Cat" mug that typically sells for $5.50. What if there is a particular mug that has a minor flaw? Although the mug can still be sold, the customer may expect a discounted price to compensate for the flaw. The question is therefore: Where on the diagram do we indicate the *actual* selling price of a particular mug?

A second example is the purchase quantity. What if the customer purchases a dozen "Fat Cat" mugs? Where is this information recorded? The "Qty ordered" attribute does not belong to the Customer entity because customers order many products besides mugs. Similarly, "Qty ordered" does not belong to the Product entity because different customers may order different quantities.

This is a problem that typically arises in many-to-many relationships: There are certain important attributes that do not seem to belong to either of the entities participating in the relationship. The solution is to assign the attributes to the relationship itself.

3.3.2.1 Attributes of relationships

The issues surrounding price and order quantity arise because the attributes belong to the *interaction* of the entities in the many-to-many relationship. Thus, the price of a flawed mug is an attribute of a particular product being sold to a particular customer on a particular day.

To summarize, there are a number of attributes that should be attached to the "buys" relationship in Figure 3.13:

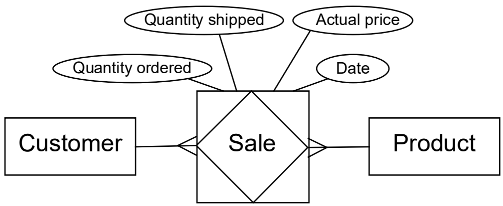
- **Date** — the date on which the purchase is made;
- **Actual price** — the price at which the item (or multiple items within the same class of products) are actually sold to the customer;
- **Quantity ordered** — the number of items with a certain product ID requested by the customer; and,
- **Quantity shipped** — the actual number of items shipped to the customer.

3.3.2.2 Associative entities

Given the number and importance of the attributes attached to the "buys" relationship, it makes sense to treat the relationship as an entity in its own right. To transform a relationship into an entity on an ERD, we use a special symbol called an associative entity. The notation for an associative entity is a relationship diamond nested inside of an entity rectangle, as shown in Figure 3.14.

To transform your many-to-many relationship (without attributes) into an associative entity (with attributes), do the following:

FIGURE 3.14: Transform a many-to-many relationship into an associative entity.



7 Draw a rectangle around the “buys” relationship.

8 Replace the relationship name “buys” with an appropriate noun, for example “Sale”.

Remember, entities—including associative entities—are named with nouns.

9 Decompose the many-to-many relationship into two one-to-many relationships.

10 Add the attributes to the associative entity, as shown in Figure 3.14.

Although Sale is now treated as an entity, there is no requirement to add relationship diamonds between Product and Sale or Customer and Sale. An associative entity serves as an entity *and* a relationship at the same time.

The meaning of the Sale associative entity in Figure 3.14 is the following: Each customer can be involved in many sales transactions, but each individual sales transaction involves only one customer. Similarly, each product can be involved in many sales transactions, but each sales transaction involves only one type of product.

3.3.2.3 Illustration

To better understand how an associative entity works, it is worthwhile to jump ahead a bit and consider what the data might look like. In Figure 3.15, sample data for the Customer, Product, and Sale entities are shown. There are a couple of interesting things to notice about the sample data:

1. Each entity contains information relevant to that entity only. For example, Customer only contains information about customers; Product only contains information about products, and so on.
2. Each row in the Sale entity shows the details of a single sales transaction. Each sales transaction consists of a particular product being sold to a particular customer. The transaction-specific information (such

FIGURE 3.15: Data showing the role of an associative entity

Customer

Customer ID	Customer name
1	Sam's Stock Pot
2	Loonie Mart #107
3	Rosch Dry Goods Inc.
4	Gadgets "R" Us
5	The Chef's Assistant

Each customer can participate in many sales transactions.

Product

Product ID	Description	Unit price	Quantity on hand
51 5012	Water jug, s.s. w/ice guar	$23.50	36
57 3826	Spatula, 6" "Cuisipro"	$4.00	65
57 3828	Spatula, 8" "Cuisipro"	$4.25	20
57 4966	Mixing bowl, 16 qt.	$12.50	7
57 551	S.S. salad server set	$3.15	32
71 12101	S.S. soup ladle	$5.25	49
71 12110	S.S. skimmer	$5.00	32
71 12111	S.S. sauce ladle	$5.25	9
71 12114	S.S. grave ladle with spou	$4.75	56
74 4042	Snail plate w/white handle	$3.15	36
74 4321	Pastry brush, 1"	$4.00	32
74 4539	Meat tenderizing hammer	$2.50	12
74 6083	Spring form pan, 9" non st	$7.50	8

Each product can participate in many sales transactions.

Sale

Customer ID	Product ID	Actual Price	Qty Ordered	Qty Shipped
3	51 5012	$23.50	12	10
1	51 5012	$22.80	1	1
3	57 3828	$4.25	48	48
2	71 12101	$5.25	12	12
3	71 12101	$5.25	24	0

as the actual selling price and quantity ordered) are attributes of the Sale entity.

3. By using the data for the Sale entity, it is possible to determine which products have been purchased by a particular customer. Similarly, it is possible to determine which customers have purchased a particular product.

4. Only the minimum amount of information required to identify the customer and product is included in the Sale associative entity. For example, neither the name of the customer or the description of the product appear in Sale since this information can easily be found elsewhere using the values of Customer ID and Product ID respectively. In the context of

the Sale entity, the Customer ID and Product ID attributes are called **foreign keys** (foreign keys are so important that Lesson 6 is devoted to the topic).

3.3.3 Revising the ERD

There is an important problem with the ERD as its now stands. The constraint that each sales transaction involves only a single product appears to be at odds with the reality of the business situation described in Lesson 2. For example, the “sales transactions” with which we are most familiar—the customer orders you receive by fax— typically request many different products: a dozen “Fat Cat” mugs, two dozen spatulas, some wire whisks, and so on. The mismatch between the diagram and the business environment means that the ERD must be revised.

3.3.3.1 Identifying the problem

The problem with the ERD in Figure 3.14 is that ignores the *technology* (broadly speaking) used by customers to place orders. Specifically, customers normally wait until they need enough stock to make an order worthwhile. In addition, factors such as minimum order values and shipping costs favor the *batching* of small, single-product orders into large, multi-product orders.

By taking the technology used for ordering into account, it becomes clear that we have failed to model an important **event entity**: the arrival of an order.

Be careful—not all pieces of paper in the existing business process are automatically event entities. For example, the invoices that we send to our customers are more properly thought of as *reports* (which can be generated from the information already contained in other entities). With practice, the distinction between entities and non-entities will become clear.

3.3.3.2 Adding the new entity

In this section, you are going to modify your ERD to include an Order entity.

11 Create a new ERD consisting of entities for customers, orders and products.

Here is where a CASE tool pays off: you can delete entities or move entities around the screen and the relationships and connector lines follow automatically.

12 Create relationships to reflect the fact that customers place orders and orders consist of products.

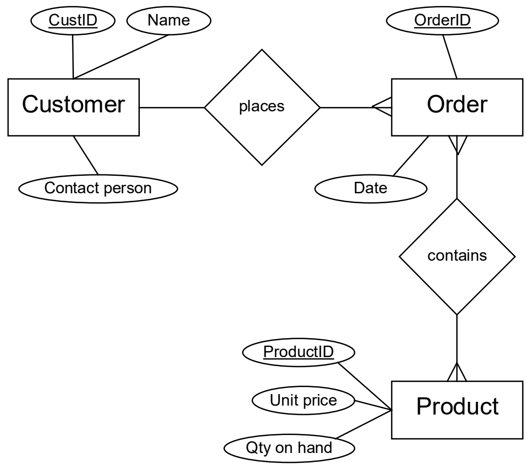
13 Add cardinality symbols to reflect the fact that each customer can place many orders, but each order belongs to a single customer.

14 Add cardinality symbols to reflect the fact that each order can contain many products and each product may be contained in many orders.

15 Add a handful of attributes to the diagram to help clarify the meaning of the entities.

The resulting ERD is shown in Figure 3.16.

FIGURE 3.16: Revise the ERD to include an Order entity.



3.3.3.3 Creating OrderDetails

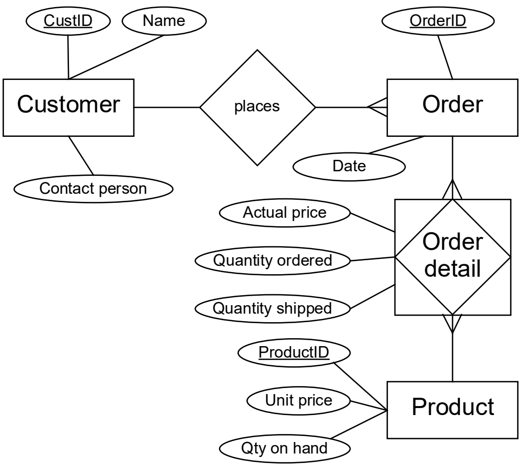
Although Figure 3.16 is a great improvement over our previous ERD, much of the same information missing from Figure 3.13—such as actual price and quantity ordered—is missing from the new ERD. As a consequence, we must transform the "contains" relationship into an associative entity with its own attributes.

16 Transform the "contains" relationship into an associative entity using the procedure described in Section 3.3.2.2.

To remain consistent with MICROSOFT's sample databases, I recommend using the name "Order Detail" for the associative entity. Alternatives include "Line Item" and "Order Item".

The resulting ERD is shown in Figure 3.17. To help understand the relationship between the Order entity and the Order Detail associative entity, look at the orders included in the project package. Each order has header information (such as customer, order date, and so on) and multiple order details. Each detail has information such as quantity ordered, quantity shipped, and price.

FIGURE 3.17: Create an associative entity to model individual order details.



3.4 Discussion

3.4.1 Logical versus physical models

A distinction is typically made between logical and physical data models:

- **Logical data models** — logical models capture general information about entities and relationships and are used for communication with business users.
- **Physical data models** — physical models serve as a precise specification for the implemented system. As a consequence, the models must take into account the technology used to store the data. For example, a given logical data model translates into very different physical data models depending on whether the target technology is a file-based system, a relational database, or an object-oriented database.

Normally, you start with a high-level logical model and refine with the help of users over several iterations. Once you are happy with the logical model, you transform it into a physical model and hand it to a database administrator (DBA) for implementation as a database.

For relational databases, the translation process from logical to physical is relatively straightforward and involves the following steps:

1. **Decompose all many-to-many relationships** — Since the relational database model does not support many-to-many relationships, you must replace all many-to-many relationships with

associative entities as described in Section 3.3.2.2.

2. **Add attributes** — The data model should be "fully attributed" before handing it over to a DBA.
3. **Identify primary keys** — Each entity requires an attribute that uniquely identifies instances.
4. **Add foreign keys** — In relational databases, relationships between entities are implemented using foreign keys.

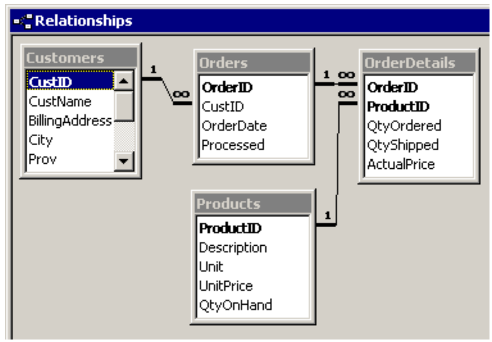
At this point, it is not critical that you understand each of these steps (you will get lots of practice in subsequent lessons). What is important is that you understand that there is a clear progression from high-level, graphical, conceptual models to low-level database schemas.

3.4.2 ERDs versus the ACCESS relationship window

If you have used the relationship window in MICROSOFT ACCESS, you know that the relationship diagrams used by ACCESS resemble ERDs. The primary differences between the two are

- The relationships between tables are not named in ACCESS.
- Many-to-many relationships are not supported. That is, the relationships window permits physical data models only.
- ACCESS uses the symbols "1" and "∞" instead of "1" and "N", as shown in Figure 3.18.

FIGURE 3.18: The relationship window in ACCESS uses a notation similar to that used in ERDs.



The ACCESS relationship window supports physical data modeling. One-to-many relationships are denoted using "1" and "∞".

3.4.3 Why do I need to know about data modeling?

Clearly, you do not *need* to know the intricacies of data modeling to start using a database package such as MICROSOFT ACCESS. In fact, MICROSOFT has gone through great lengths between the release of ACCESS version 2.0 and ACCESS 2000 to make the product more accessible to data modeling neophytes. For example, there is a table analyzer, a table design wizard, and all sorts of other aids intended to automate the database design process. In my view, there are three problems with MICROSOFT's "dumbing-down" strategy:

1. No wizard or add-in tool is going to change the fact that database management systems (DBMSs) are specialized software packages that presuppose an enormous amount of prior knowledge. Even the error messages generated by ACCESS (as you will soon discover) can only be understood if you have a firm grasp on the theoretical fundamentals of the relational database model. For example, what do you do when you accidentally violate a "referential integrity constraint"? What is a "primary key"? What is an "ambiguous outer join"?
2. Trends in application development increasingly emphasize data and de-emphasize programming. Thus, a solid understanding of your data is critical. Put another way, just about anyone can build a reasonably sophisticated system if the underlying database is well designed (indeed, by doing these tutorials, you will see just how far you can go without writing a single line of programming code). In contrast, if the database design is poor, you will have to be a programming wizard just to create the illusion that the application works.
3. CASE tools from vendors such as ORACLE, COMPUTER ASSOCIATES, VISIO (now part of MICROSOFT), and many others translate ERDs directly into database tables. Thus, if you know how to draw diagrams similar to the one shown in Figure 3.17, you know how to design databases.

In short, the last thing you want to do is rely on wizards to shelter you from the database design process. Instead, you want to get in at the nitty-gritty level and understand the trade-offs between various designs. Once the design is complete you can use wizards and shortcuts for everything else.

3.4.4 How do I learn about data modeling?

One important problem that I perceive as a university professor is that despite the importance of data modeling, it is very difficult to find good practical training as a data modeler.

In the standard computer science database course, we tend to focus on theoretical issues such as relational algebra, set theory, indexing, normalization, and so on. Although such knowledge is certainly important for DBAs and other technical professionals, it provides little guidance when we are faced with real-world modeling problems.

Conversely, the introductory information systems (IS) courses that we offer in business schools provide only cursory treatment of data modeling and database design. I suppose the rationale is that it should be possible to hire a computer science graduate to do the data modeling!

3.4.5 CASE tools and the design process

Computer-aided software engineering (CASE) tools are software packages that simplify the process of creating conceptual models. In addition, some CASE packages are more than just drawing tools: they translate the data models into database tables, programming code templates, and so on.

To illustrate, consider the diagram in Figure 3.19 which was drawn using the "database modeling tool" in VISIO ENTERPRISE. The database modeling tool allows a designer to start with a a physical ERD-like diagram and add implementation-level metadata (data about data). For example, in Figure 3.20, the physical-level properties of the `ActualPrice` attribute are specified in a dialog box.

Once the physical-level metadata has been added to the model, many CASE tools can generate table schemas for the target database. For example, ORACLE DESIGNER can generate structured query language (SQL) data definition commands for ORACLE databases. VISIO ENTERPRISE can generate SQL or create the tables in various database packages (including ACCESS) directly.

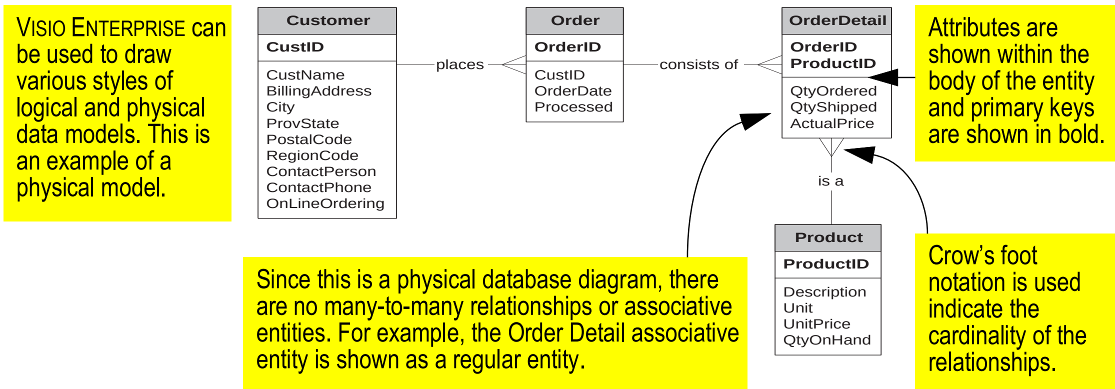
Given CASE tools with this type of functionality, it is clear that the most important skill for database designers is not a complete knowledge of SQL syntax. Instead, it is the ability to analyze a real-world problem and create a data model that accurately and elegantly captures the critical elements of the problem.

3.5 Application to the project

17 Complete your own ERD for the order entry scenario. Remember that at this point, the scope of the project is very limited. It will be expanded somewhat in subsequent lessons.

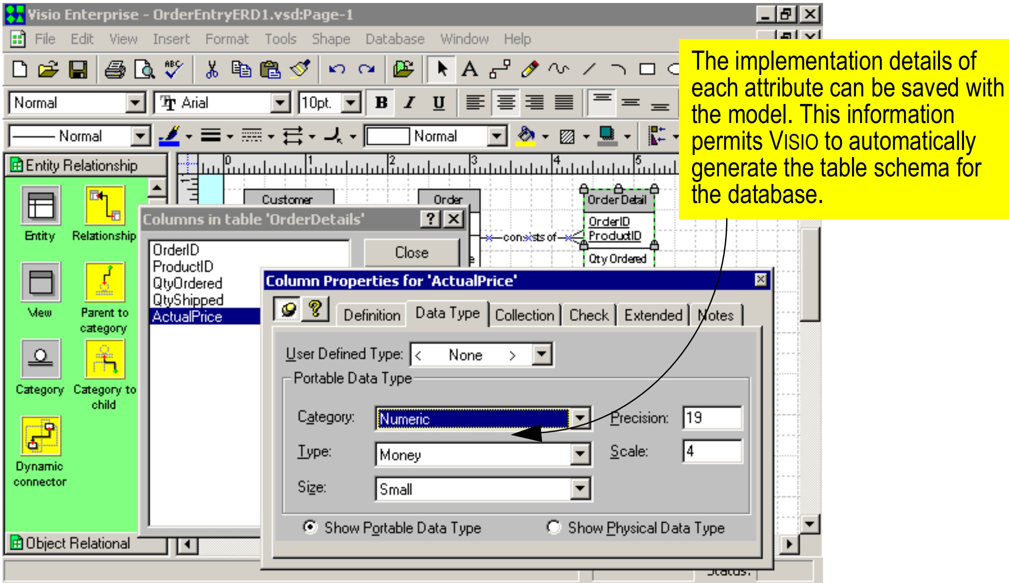
HINT: If you get stuck, you can refer to the VISIO diagram in Figure 3.19. Keep in mind, however, that Figure 3.19 is a *physical*

FIGURE 3.19: An physical database model for the kitchen supply environment created using a CASE tool.



ERD; you should be working with *logical* ERDs at this stage of the development process.

FIGURE 3.20: A CASE tool can be used to add implementation details to a graphical model.



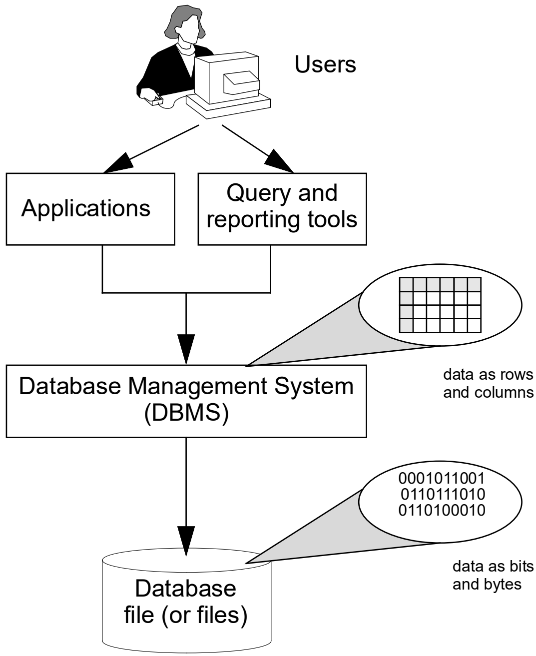
4.1 Introduction: What is Access?

MICROSOFT ACCESS is a **relational database management system** (DBMS). At the most basic level, a DBMS is a program that facilitates the storage and retrieval of structured information on a computer's hard drive.

4.1.1 The role of database management systems

The role of the DBMS within an information system or application is shown in Figure 4.1. As far as we are concerned, a **database** is an amorphous blob[1] of data stored in binary format (ones and zeros) on disk. To read and write to the database, a DBMS is required.

FIGURE 4.1: The role of the DBMS.



Despite the clear conceptual difference between a database (a blob of binary data) and a DBMS (software to manage one or more databases), we often refer to programs such as ACCESS and ORACLE as "databases". Although such usage is sloppy, the context in which the term is used is usually sufficient to eliminate any confusion.

[1] The term blob is used here in its informal sense (e.g., "a blob of gunk"). In database terminology, the acronym BLOB refers a special data type used to store Binary Large OBjects (like graphics files or spreadsheet). We discuss different data types in Lesson 5.

An important feature of the arrows in Figure 4.1 is that users do not interact with the DBMS directly. Instead, they use either a special-purpose application (e.g., a payroll program) or a query and reporting tool (e.g., CRYSTAL REPORTS) to view or modify the data. Similarly, the applications and query/reporting tools do not access the database directly. Instead, all requests for data are made to the DBMS and the DBMS software takes care of reading and writing data to and from the hard disk.

The primary advantage of this layered approach is that the DBMS is used to hide the complexities of low-level disk access from both the users and the application designers. In other words, the DBMS software provides an **abstraction**—instead of thinking about bits and bytes, end-users and designers can think in terms of **tables** of data consisting of **rows** and **columns**.

4.1.2 Inside an ACCESS database file

Although the term "database" typically refers to a collection of related data "tables", an ACCESS database file includes more than just tables. Indeed, an ACCESS ".mdb" file contains several different types of **database objects**:

- saved **queries** for organizing data;
- **forms** for users to interact with the data on screen;
- **reports** for organizing, summarizing, and printing data; and,
- **macros** and **VISUAL BASIC programs** for extending the functionality of database applications.

All these database objects are stored in a single file named `<file name>.mdb`.

When you are running ACCESS, a temporary "locking" file named `<file name>.ldb` is also created. You can safely ignore the *.ldb file; everything of value is in the *.mdb file.

4.2 Learning objectives

- identify the version of MICROSOFT ACCESS that you are using
- open and explore an existing database
- learn how to create a new database
- identify the database window and understand how the different database objects fit together
- get help from the on-line help system
- compact a database to save space

4.3 Exercises

4.3.1 Starting Access

1 To start ACCESS, you double click the ACCESS icon (for version 8.0 and 7.0 or for version 2.0) from within MICROSOFT WINDOWS.

If you are uncertain which version you are using, you can watch for the “splash” screen as the program loads. Alternatively, selecting **Help** → **About ACCESS** from the main menu will tell you everything you need to know.

4.3.2 Finding and using an existing database

In order to make the elements of Figure 4.1 more concrete, we will start by finding and opening a sample database application provided by MICROSOFT called “NORTHWIND TRADERS”. NORTHWIND TRADERS is a fictitious wholesaler of specialty foods to retailers around the world. Given its business environment, it is not surprising that the order entry system application built for NORTHWIND is similar to the one that you will build from scratch in these tutorials.

When you are stuck doing your own projects, it is sometimes convenient to take a look at the NORTHWIND TRADERS application to see “how MICROSOFT does it.” However, keep in mind that MICROSOFT’s way is only one of many possible ways of accomplishing tasks.

The problem with opening the sample file is that its location depends on the precise manner in which MICROSOFT OFFICE was installed on your computer. To find the file, you can use one of three approaches:

- Look around in the file system directory in which ACCESS/OFFICE is installed.
- Use the “advanced find” feature of the “open file” dialog within ACCESS. This approach is described in more detail below.
- Use the search feature of your operating system. For example, in Windows 95/98/NT/2000, use **Start** → **Search/Find** → **Files or Folders** from the task bar.

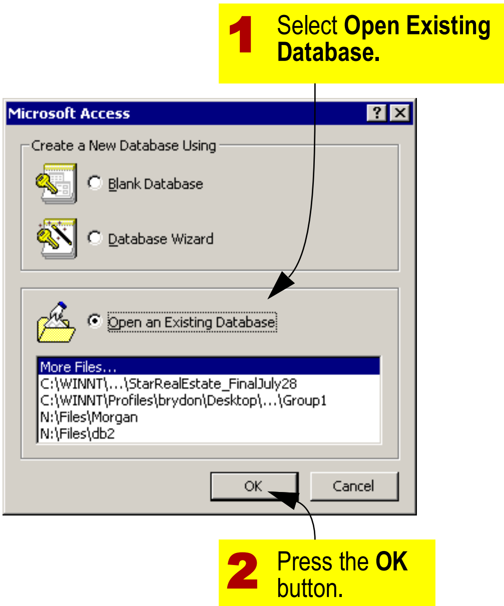
2 Because of the filename limitations in WINDOWS 3.x, the sample application that ships with ACCESS version 2.0 is called `Nwind.mdb`. In more recent versions, it is called `Northwind.mdb`.

Show me (lesson4-1.avi)

To find the file on your hard disk using the “advanced find” feature, do the following:

2 After starting ACCESS, you should see a screen similar to that shown in Figure 4.2. Select **Open an Existing Database** and press **OK**.

FIGURE 4.2: Open an existing file from within ACCESS.



If you do not get the dialog in Figure 4.2 or accidentally close the dialog window, you can select **File** → **Open** from the main menu at any time.

You should now have a dialog box with the title “Open”.

3 Type “Northwind” into the field labeled **File name** and press the **Advanced** button, as shown in Figure 4.3.

4 Select the drive on which ACCESS is installed, indicate that you would like to search in the subfolders of the current folder, and press **Find Now** (see Figure 4.3).

Hopefully, the “advanced find” utility finds the location of the sample file. If not, you can revert to one of the other approaches above.

When installing ACCESS, you are given the option whether to install the sample databases. If you cannot find the NORTHWIND TRADERS database on your computer it is probably because it was never copied to your hard disk during installation. This problem is easily solved, however: re-run `setup.exe` from the OFFICE CD-ROM and indicate that you want the sample databases installed.

5 When you have found the file, highlight it and press **Open**. Alternatively, you can

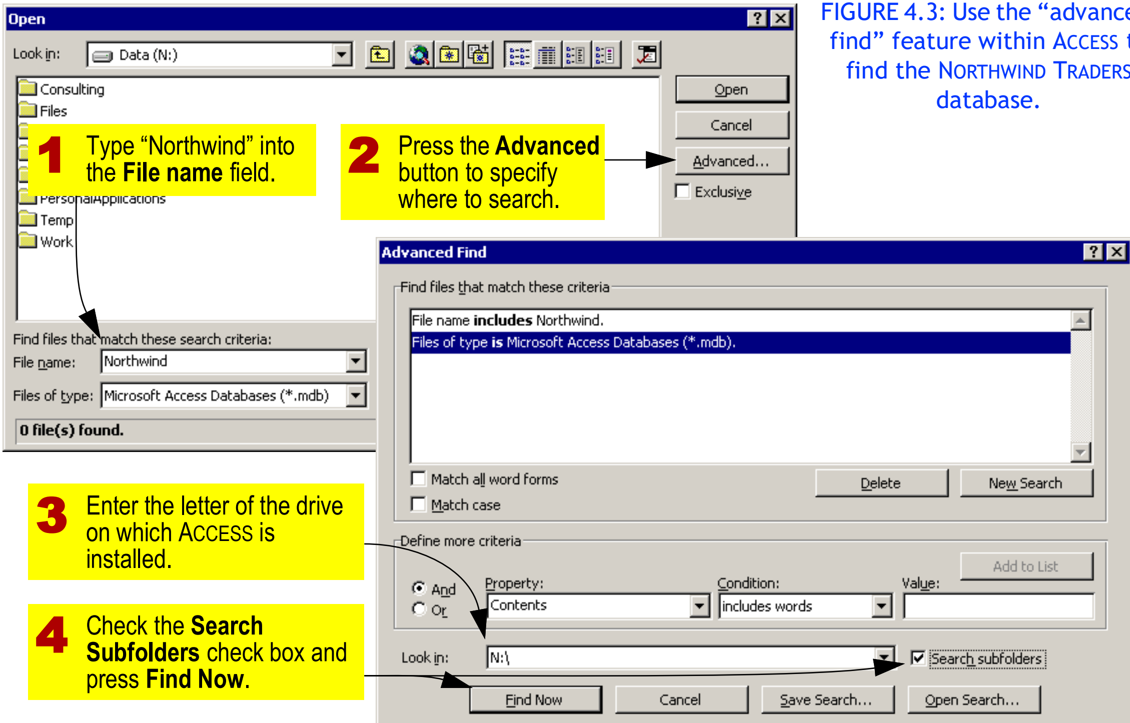


FIGURE 4.3: Use the "advanced find" feature within ACCESS to find the NORTHWIND TRADERS database.

simply double-click the file in the file dialog.

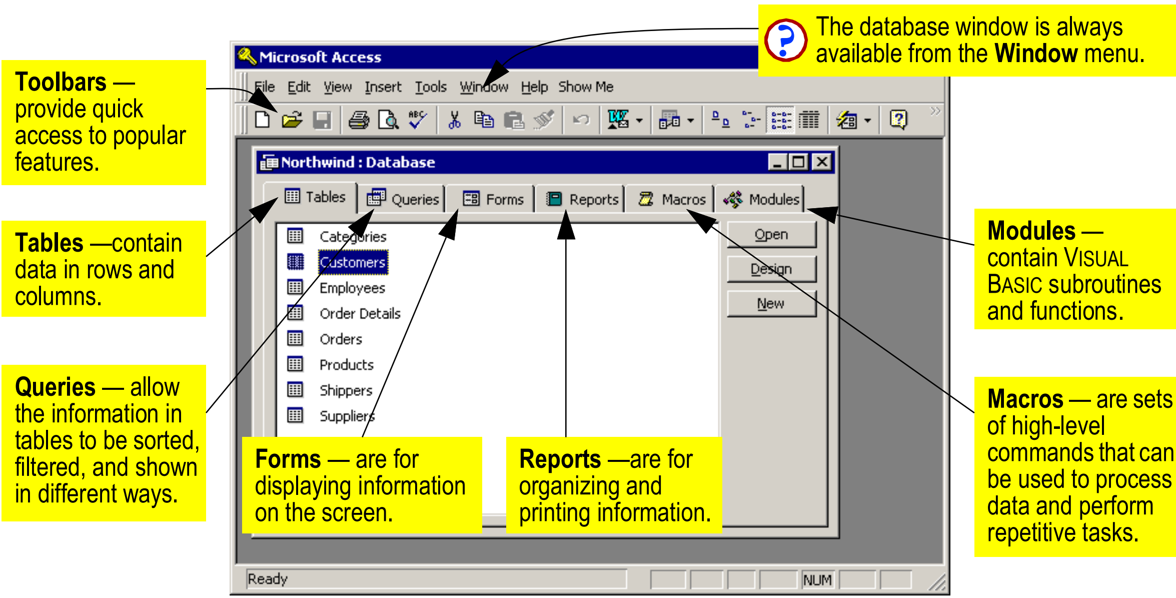
If it seems like we are working through these steps in excruciating detail, it is because we are. As the lessons progress, the pace quickens. Eventually, you will simply be told *what* to do, not *how* to do it. In these early stages, however, we are moving slowly.

4.3.3 Exploring the NORTHWIND TRADERS database

Depending on whether you have ever opened the NORTHWIND TRADERS file, an introductory screen may appear.

6 Close the introductory screen (if any). You should be left with the database window, as shown in Figure 4.4.

FIGURE 4.4: The database window contains all the database objects for a particular application.



4.3.3.1 Tables

In this section, you are going to take a brief look at the `Customers` table. Tables are where all the data in a database is stored. As such, understanding the structure of tables is critical.

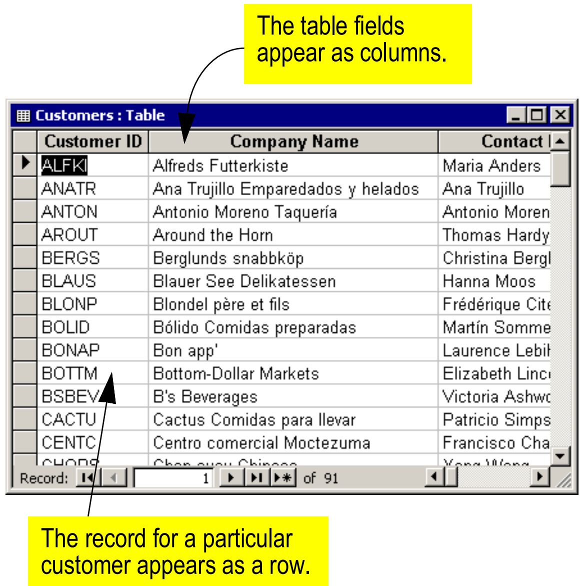
7 Ensure the **Table** tab is selected in the database window and double-click the `Customers` table.

The datasheet view shows the data that is stored in the table (see Figure 4.5). Each column (or field) corresponds to an attribute of the entity and each row (or record) corresponds to an instance of the entity.

In the pre-database era, the logical structure of a data file was described in terms of records and fields. Thus, each customer would have one record in the customer file and the customer's phone number would be stored in the `PhoneNo` field. Although the terms "row" and "column" are preferred in the relational database context, "record" and "field" and are still widely used (even by relational DBMSs such as ACCESS). In these lessons, we will use the terms row/record and column/field interchangeably.

8 Switch to the table's design view by selecting **View** → **Design View** from the main menu.

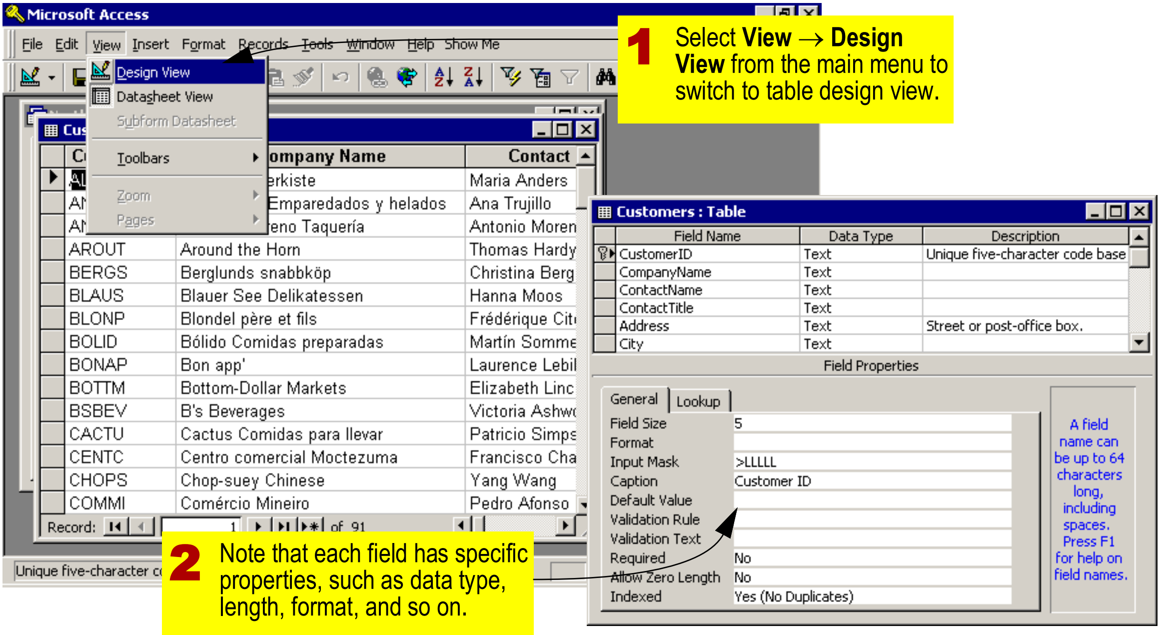
FIGURE 4.5: Open the *Customers* table.



9 Take a brief moment to observe the table's structure, as shown in Figure 4.6.

Although the datasheet view may resemble a spreadsheet, the information in a database is highly structured. For example, you cannot simply move to a

FIGURE 4.6: Switch to the table's design view to observe its structure.



column labeled **PhoneNo** and type in some arbitrary text. The **PhoneNo** column, like other columns, has a predefined structure and data type that is enforced by the DBMS.

10 Close the **Customers** table. You should be back at the database window.

Although it is possible to make changes to the data in datasheet view, this is not the way in which we will expect our users to interact with the data. Instead, we will use forms to create a

user interface that is friendlier and more functional.

4.3.3.2 The user interface

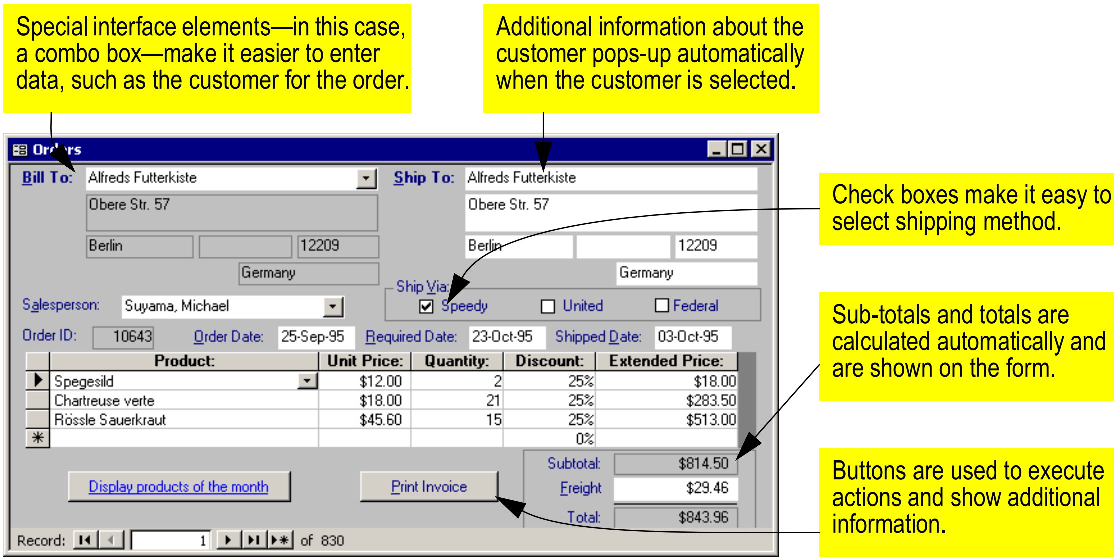
To get a better sense of what a **database application** looks like, and to see how data is actually entered into the database, we can look at the order entry form for NORTHWIND TRADERS.

Show me (lesson4-2.avi)

11 Click on the **Forms** tab along the top of the database window and double-click the form called `Orders`. You may have to resize the form to make it completely visible on your screen.

12 Note some of the features of the form, as shown in Figure 4.7.

FIGURE 4.7: Explore the order form for NORTHWIND TRADERS.



13 Close the order form.

4.3.3.3 Queries and reports

The final type of database object we are going to consider on this whirlwind tour is a report. Reports are generally created in two steps. In the first step, a query is used to organize and filter the information that is required for the report. In the second step, a report template is created to display the information in a format fit for human consumption.

14 Click on the Reports table along the top of the database window and double-click the report called `Products by Category`.

15 Click on the report to zoom in and zoom out. Note that the report breaks out NORTHWIND's products and inventory out by product category and formats the information into columns.

16 Close the report. Also, close the database window for NORTHWIND TRADERS.

Here ends our tour of the sample application. Hopefully, you now have a better idea of what a desktop database can do. Before we end this lesson, you will create a database file of your own and learn about some basic housekeeping issues.

4.3.4 Creating a new database

17 Select **File** → **New Database** from the main menu.

18 In the "New" dialog box, select the **General** tab and "Blank Database".

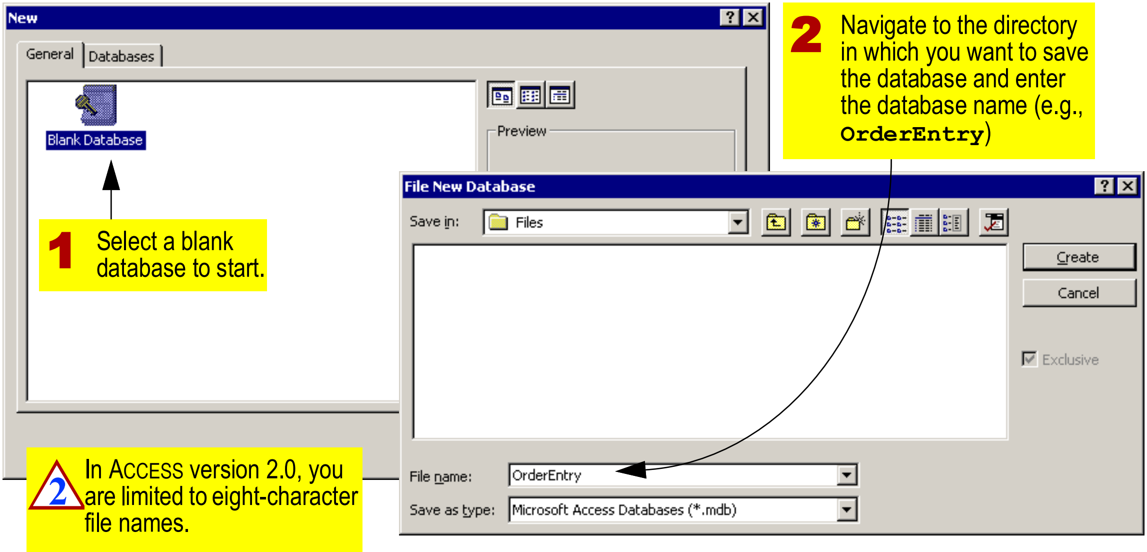
8 The "New" dialog does not exist in version of ACCESS prior to version 8. It permits you select the **Databases** tab and chose from a number of database design wizards. As indicated previously, a database design wizard may sound interesting, but its use contributes nothing to our pedagogical objectives.

19 Navigate to the directory in which you want your new database file to be saved and enter a name (e.g., `OrderEntry.mdb`) for the file, as shown in Figure 4.8.

4.3.5 Using the on-line help system

Commercial software relies increasingly on on-line help and documentation in lieu of printed manuals. As a consequence, experience navigating on-line help systems is essential for learning any new software. In this section, you will use ACCESS' on-line help system to learn how to perform a rather esoteric (but useful) task in ACCESS: compacting a database.

FIGURE 4.8: Create a new database and save it in a subfolder.



The term “on-line” is used here in the general sense to mean “computer-based”. The “connected to the Internet” connotation of the word is a relatively recent etymological development.

20 Press **Help → Contents and Index** to invoke the on-line help system. Use the index to find information on compacting a database.

21 Skim the contents of the entry.

4.3.6 Compacting your database

As the on-line help system points out, ACCESS database files can become highly fragmented and grow to become much larger than you might expect given the amount of data they contain. Compacting your database from time to time eliminates fragmentation and can dramatically reduce the disk space requirement for the database.

22 Follow the directions provided by the on-line help system to compact your database.

Since your database is completely empty at this point, compacting has little effect. The pedagogical objective of this section is to create awareness of the compacting feature so that you know what to do when your database balloons to a couple of megabytes for no apparent reason.

The compacting utility in MICROSOFT ACCESS helps use space within the ".mdb" file more efficiently. Compacting has nothing to do with general-purpose "compression" software such as WINZIP or PKZIP.

4.4 Discussion

4.4.1 Relationship between ACCESS and other databases

MICROSOFT ACCESS is a desktop DBMS. It is designed for use by a small number of users on a single machine and the design emphasis is on convenience rather than raw performance. Other notable desktop DBMSs include MICROSOFT FOXPRO, BORLAND/INPRISE dBASE and PARADOX, LOTUS APPROACH, and FILEMAKER PRO.[1]

Because of its emphasis on convenience, ACCESS brings all the components in Figure 4.1 into a single, integrated package. Specifically, ACCESS provides:

- a database engine that takes care of reading and writing data to disk, optimizing queries, managing security, and so on;
- query and reporting tools for extracting and formatting specific data; and,
- a complete environment for creating and executing custom applications.

In contrast, many industrial-strength client/server database such as ORACLE, MICROSOFT SQL SERVER, and IBM DB2 are strictly database management systems. Although these vendors also sell supplemental tools for every imaginable query and reporting requirement, the supplemental tools are best seen as distinct products. In most cases, industrial-strength

---

1 The multiple offerings from MICROSOFT and BORLAND/INPRISE are the results of acquisitions over time. Large and vocal "installed bases" (rather than any substantive technological differences) appear to be the primary barrier to product-line rationalization.

DBMSs are invisible to users and are accessed over a network by custom applications written in programing languages like C++ or VISUAL BASIC.

Another class of DBMS software that is worth noting is **open-source** client/server databases such as MYSQL, POSTGRES, and INTERBASE. These databases are freely available under open-source licenses and in some cases offer functionality and power that rival commercial products. INTERBASE, for example, was formerly a commercial product selling for about US$10K per copy. Now it is free.

All three of the open-source databases mentioned above run on LINUX, an open-source operating system. Thus, it is possible to run a sophisticated database server using software that costs nothing. The downside of open-source software in general is that it tends to be aimed at people who know what they are doing.

4.4.2 The many faces of ACCESS

MICROSOFT typically incorporates as many features as possible into its products. For example, the ACCESS product contains the following elements:

- a relational database system that supports two industry-standard query languages: Structured Query Language (SQL) and Query By Example (QBE);
- a full-featured procedural programming language—essentially a subset of VISUAL BASIC,
- a simplified procedural macro language that is unique to ACCESS;
- a rapid application development environment complete with visual form and report development tools;
- a sprinkling of objected-oriented extensions; and,
- various wizards and builders to make development easier.

For new users, these “multiple personalities” can be a source of enormous frustration. The problem is that each personality is based on a different set of assumptions and a different view of computing. For instance,

- the relational database personality expects you to view your application as sets of data;
- the procedural programming personality expects you to view your application as commands to be executed sequentially;
- the object-oriented personality expects you to view your application as a collection of independent objects with state, methods, and events.

MICROSOFT makes no effort to provide an overall logical integration of these personalities (indeed, it is unlikely that such an integration is possible). Instead, it is up to you as a developer to pick and choose the best approach to implementing your application.

Since there are often several vastly different ways to implement a particular feature in ACCESS, recognizing the different personalities and exploiting the best features (and avoiding the pitfalls) of each are important skills for ACCESS developers.

The advantage of these multiple personalities is that it is possible to use ACCESS to learn about an enormous range of information systems concepts without having to interact with a large number of "single-personality" tools, for example:

- ORACLE for relational databases
- POWERBUILDER for rapid applications development,
- SMALLTALK or JAVA for object-oriented programming.

Keep this advantage in mind as we switch back and forth between personalities and different computing paradigms.

4.4.3 Developing applications in Access

In general, there are two basic approaches to developing an information system:

- in-depth systems analysis, design, and implementation,
- rapid prototyping (in which analysis, design, and implementation are done quickly and iteratively).

ACCESS provides a number of features (such as graphical design tools, wizards, and a high-level macro language) that facilitate rapid prototyping. Since you are going to build a small system and since time is limited, you will use a rapid prototyping approach to build your application. That is, rather than undertake an elaborate requirements specification and design phase, you are going to sketch some data models and dive right into building a prototype. You will then test the prototype, make changes, and repeat the cycle until you have developed a good solution to the business problem.

During the iterative development process, prototypes often become full of junk and remnants of failed approaches. In addition, prototypes do not have sophisticated error-handling code or internal documentation. Thus, it is often good practice to throw the prototype away at the end of the development process and use what you have learned during prototyping to build a clean system from the bottom up.

As you will discover, starting over from scratch is not as bad as is sounds—once you know what you are doing, you can re-build a system very quickly.

4.5 Application to the project

23 Play with the NORTHWIND TRADERS application until you have a good idea of what it does.

If you are worried about damaging or corrupting the sample database, you can make a copy under a different name (e.g., `myNorthwind.mdb`) and work with the copy instead.

24 Ensure you have drawn an ERD (see Lesson 3) to model the data requirements of the system. The ERD will help you design tables in the next lesson.

If you take a closer look at the NORTHWIND application, you will notice that it contains more detail than the simple ERD we created in Lesson 3. For example, the `Products` table contains information about, reorder quantity, whether the product has been discontinued, the supplier of the product, an so on. Although you are free to add additional tables and attributes to your kitchen supply application, the skills and principles are the same regardless of whether you build tables that have five fields or 50 fields. As a consequence, I recommend that you accept the unrealistically narrow scope of the project in order to minimize the amount of time you waste repeating the same simple skills.

Lesson 5: Building basic tables

5.1 Introduction: The importance of good table design

The advantage of building a business application on top of a relational database is that the DBMS provides an abstraction that hides the ugly physical details of data storage. This allows you to ignore the bits and bytes and concentrate on higher-level constructs like tables, rows, and columns.

If you can create a good table structure, you will be able to use a tool like MICROSOFT ACCESS to put together a functional and robust application without resorting to sophisticated programming. If your table design is poor, however, you will have to be a black-belt programmer just to achieve a basic level of functionality.

Extra time spent thinking about table design can result in enormous time savings during later stages of the project. Non-trivial changes to tables and relationships become increasingly difficult as the application grows in size and complexity.

In addition to storing information about products, customers and so on, ACCESS uses **field properties** to store large amounts of information about the data itself (such as captions, default values, constraints, etc.). Such *data about data* is called **metadata**.

There are no standards for what types of metadata are supported by various relational DBMS vendors and thus the discussion here is necessarily ACCESS-centric. However, the fundamental motivations and mechanisms for using metadata are the same for all database systems.

5.2 Learning objectives

- create a new table from scratch
- set the primary key for a table
- specify field properties such as the input mask and caption
- gain some experience using the input mask wizard
- learn about the different data types supported by ACCESS
- understand why an autonumber field will not restart counting at one

5.3 Exercises

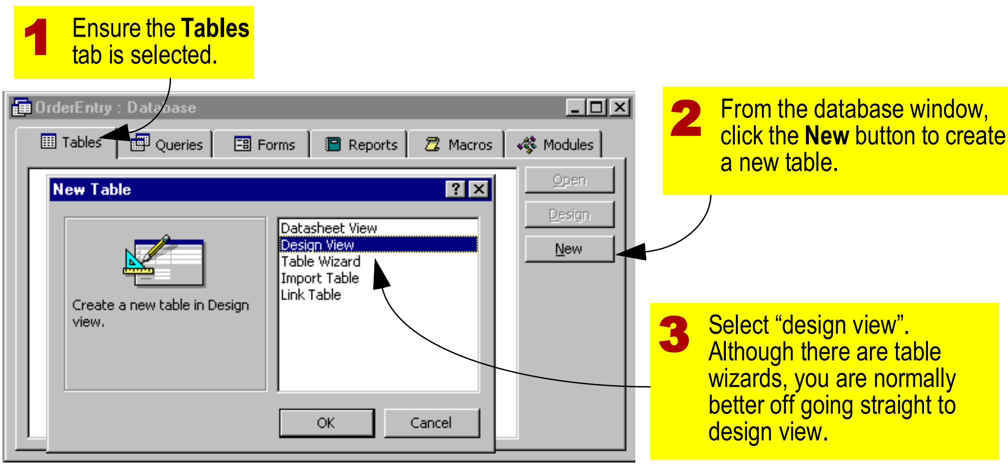
In this lesson, you will start to implement the entities in your ERD as tables in a relational database.

5.3.1 Creating a new table from scratch

1 If it is not already open, open the `OrderEntry` database you created in Lesson 4.

2 Ensure the **Tables** tab of the database window is selected and press the **New** button. This is shown in Figure 5.1.

FIGURE 5.1: Create a `Customers` table from scratch.



The table design window allows you to specify the structure of the data (databases are all about structured data). The key elements of the table design window are shown in Figure 5.2. For each field, you must specify a set of core field properties such as field name, data type, and length (if appropriate). You will learn more about these and other field properties in Section 5.4.5. For now, you can use the field properties that are provided below.

3 Create the `Customers` table using the field names and data types shown in Table 5.1.

Entities on an ERD are typically named using the singular form of a noun (e.g., "Customer") whereas tables are typically named using the plural form (e.g., "Customers").

5.3.2 Setting the primary key

Each table must have a **primary key** that uniquely identifies each row in the table. For example, every customer should have a unique `CustID`.

FIGURE 5.2: Use the table design window to enter the field properties for the `Customers` table.

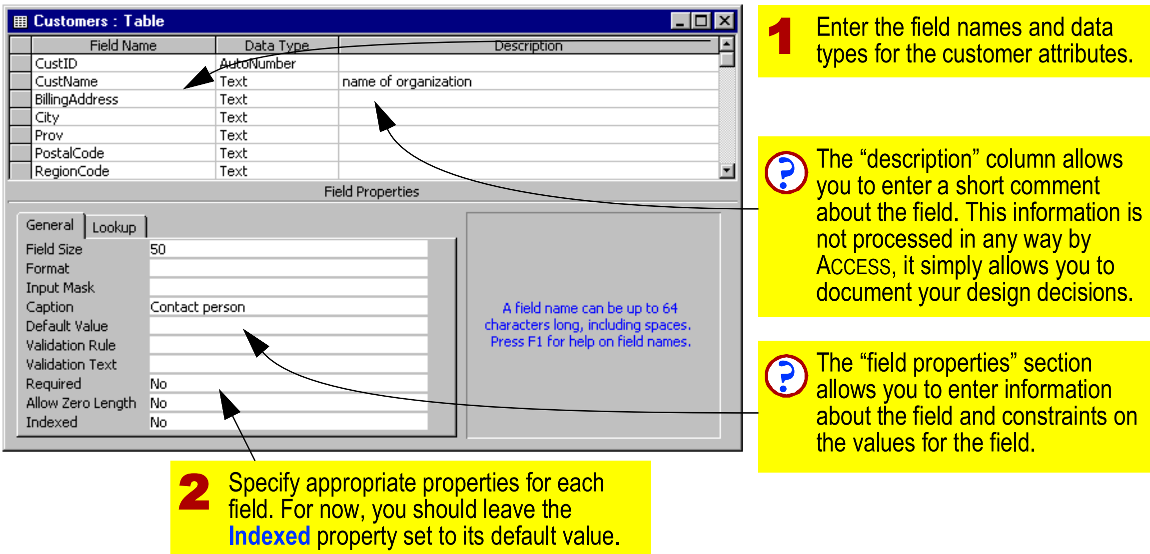


TABLE 5.1: Suggested field names and data types for the *Customers* table.

Field name	Data type (length)
CustID	autonumber/counter
CustName	text(30)
BillingAddress	text(100)
City	text(20)
Prov	text(2)
PostalCode	text(7)
RegionCode	text(1)
ContactPerson	text(50)
ContactPhone	text(15)
OnLineOrdering	yes/no

When you designate a field as the primary key, ACCESS will prevent you from entering duplicate values into the field. Thus, if you have a record with the primary key `CustID` = 3 in the table already, you will be prevented from adding another record with the same value of `CustID`.

4 Select the `CustID` field and use **Edit** → **Primary Key** (as shown in Figure 5.3) to set `CustID` as the table's primary key.

Show me (lesson5-1.avi)

5 Select **File** → **Save** from the main menu (or press **Ctrl-S**) to save the table under the name `Customers`

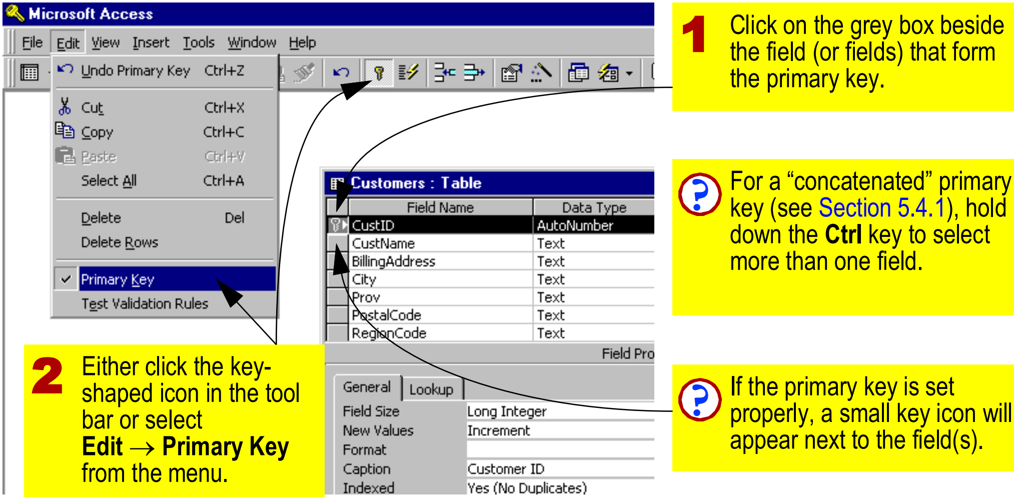
It always a good idea to save your work as you go. Saving using the **Ctrl-S** key combination works in any WINDOWS application and it is easy to get into the habit of pressing **Ctrl-S** often when designing database objects such as tables, queries, and forms.

5.3.3 Specifying optional field properties

In addition to core field properties such as name and data type, ACCESS allows you to specify the following when creating tables:

- formatting properties that control how the data looks when displayed;
- constraints that can be used to control the type of data that can be entered into the field; and,
- index information that makes searching and sorting more efficient.

FIGURE 5.3: Set the primary key for the **`Customers`** table.

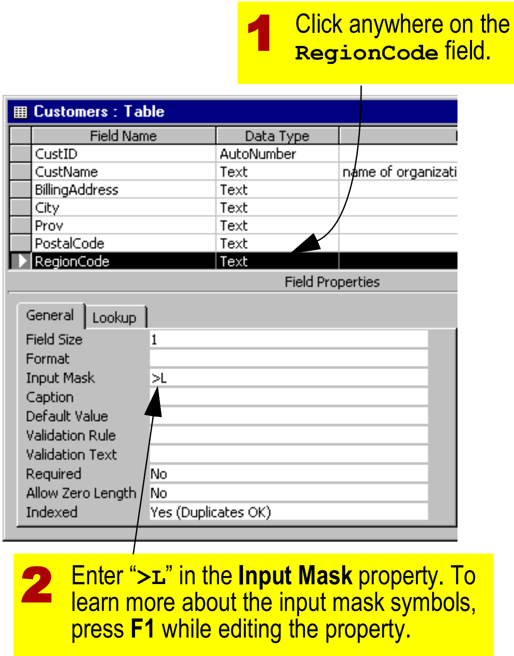


6 Set the **Caption** property for all fields that have "user *un*-friendly" names. For example, the caption for the **`CustName`** field should be set to something like "Customer name." Similarly, the caption "Province or state" can be used to provide a more general label for the **`Prov`** field.

If you specify a caption for a field, ACCESS uses the caption instead of the field's name when displaying your data. This feature allows you to use short, compact field names, but present end-users with more meaningful field name aliases.

7 Set the **Input Mask** property for the **`RegionCode`** field to "**`>L`**" as shown in Figure 5.4.

FIGURE 5.4: Set the input mask for the ***RegionCode*** field.



An input mask allows you to specify a "template" that controls the type of data that can be entered into the field. For example, the input mask you have just entered prevents users from entering anything other than a single capital letter from A to Z. More complex input masks are discussed in Section 5.4.6.

8 Save the `Customers` table.

5.3.4 The input mask wizard

In this section, you will use the input mask wizard to create a complex input mask for a standard text field.

Show me (lesson5-2.avi)

9 Bring up the `Customers` table in design view.

10 Select the `ContactPhone` field, move the cursor to the input mask property, and click the button with three small dots (...) to invoke the input mask wizard.

11 Follow the instructions provided by the wizard to create a phone number input mask.

12 Close the `Customers` table.

5.3.5 Creating a lookup table

To store each customer's region, a single-letter code (e.g., "E") is used rather than the full name of the region (e.g., “Eastern”). A list of the codes used in your company to represent sales regions is reproduced in Table 5.2.

TABLE 5.2: Single-letter sales region codes used within your company.

Region code	Region name
N	North
S	South
E	East
W	West
C	Central
K	Key

Not only does use of a simple code save time and space when entering data, it helps to avoid data entry errors and typos. For example, if a non-code field is used, different users may enter the same region in different ways (e.g., “East”, “Eastern”, “east”, and so on). Unlike humans, database software is not very good at recognizing that these variations are meant to represent the same region. When it comes time to query the data and calculate (for example) net sales by region, such inconsistencies in data entry can lead to incorrect results.[1]

The downside of short codes is that they are difficult to remember and interpret. For this reason, it is often worthwhile when implementing a database to use a **lookup table** to associate short codes and numeric IDs with more meaningful descriptions. A lookup table is simply a table that matches codes with descriptions in exactly the same manner as Table 5.2.

You will see in Lesson 15 that we can use lookup tables in ACCESS to create combo boxes and other convenient interface elements.

13 Create a new table in your database called `Regions`. The table should consist of two text fields: `RegionCode` and `RegionName`.

Since `RegionCode` is text with length = 1 in the `Customers` table, it should be the exact same data type and length in the `Regions` table.

14 Set the input mask for `RegionCode` to “`>L`” as you did for the `Customers` table. In

[1] It would be a bad thing, for instance, to single-out the sales representative assigned to the Eastern region for poor performance when the real problem is inconsistent data in the order entry system.

this way, the format of **`RegionCode`** data will be identical in both tables.

15 Do not specify a caption for **`RegionCode`**. We will use this field in a subsequent lesson to illustrate the extra work that is created by ignoring the caption property.

16 Save and close the **`Regions`** table.

5.3.6 Populating the Regions table

As you discovered in Lesson 4, tables in ACCESS have two views: **design view** and **datasheet view**. The datasheet view allows you to view and interact with the data that is stored in the table.

Although your **`Regions`** table is now defined, it contains no data. In this section, you are going to add some data (i.e., "populate" the table) in order to get a better understanding of datasheet basics.

In general, it is good practice to create all your tables and relationships before populating your tables. We are taking a shortcut here for pedagogical purposes.

17 Open the **`Regions`** table in datasheet mode by double-clicking the table name the database window The critical elements of the datasheet view are shown in Figure 5.5.

18 Using the values in Table 5.2, add the first region code ("N") to your **`Regions`** table.

19 Note the behavior of the **record selector** when you are adding, modifying, and saving records, as shown in Figure 5.6.

Show me (lesson5-3.avi)

20 Attempt to change **`RegionCode`** to some other value (e.g., "7", "kat", "#"). You will see that the input mask is at work ensuring that a capital letter A to Z is all that can be entered into the field.

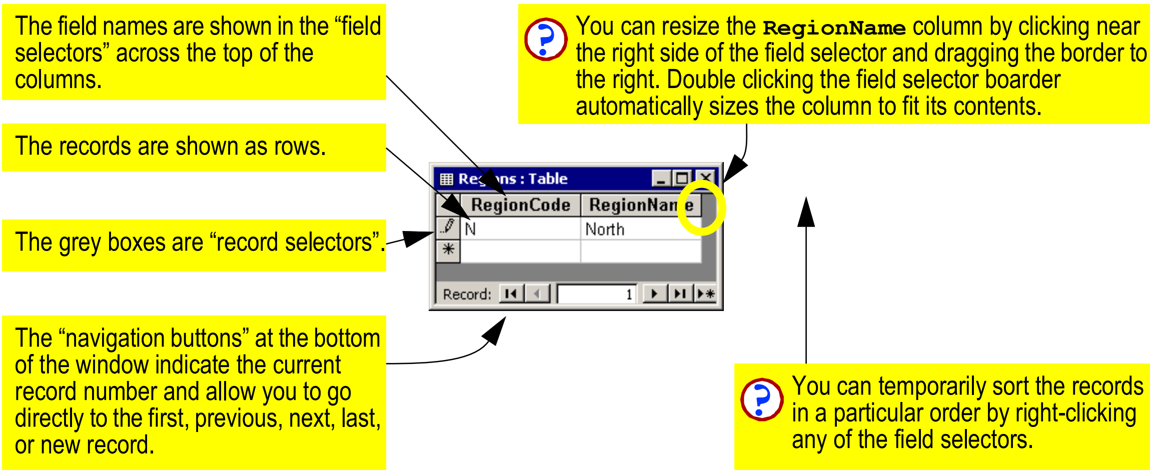
5.4 Discussion

5.4.1 Key terminology

A key is one or more fields that uniquely determines the identity of the real-world object that the record is meant to represent. For example, the customers in your **`Customers`** table are assigned sequential **`CustID`** numbers by ACCESS.

The advantage of automatically generating a unique **`CustID`** field instead of using an existing field—like the Customer's company name—is that there may be more than one organization

FIGURE 5.5: The critical elements of the datasheet view of a table.



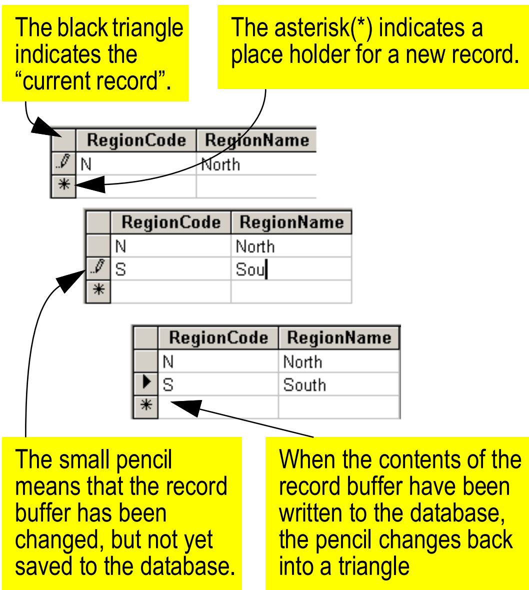
with the same name (if in doubt, search YAHOO for a common name like “IPC”).

Since the terminology of keys can be confusing, the important terms are summarized below.

1. **Primary key** — The terms “key” and “primary key” are often used interchangeably. Since there may be more than one **candidate key** for an application, the designer has to select one: this is the primary key.

2. **Concatenated key (or compound key)**: The verb “concatenate” means to join together into a chain. Hence, a concatenated key is made by joining together two or more fields. Course numbers at universities provide a good example of a concatenated key made by joining together the `DeptCode` and `CrsNum` fields. Department code alone cannot be the primary key since there are many courses in each department (e.g., BUS 492, BUS 905). Similarly, course number cannot be used as a key since there are

FIGURE 5.6: States of the record selector when adding and modifying records.



many courses with the same number in different departments (e.g., BUS 492, HIST 492, MATH 492). However, department and course number *together* form a concatenated key—there is only one BUS 492, for example.

3. **Foreign key**: In a one-to-many relationship, a foreign key is a field (or fields) in the “child” record that uniquely identifies the correct “parent” record. For example, `RegionCode` in the `Customers` table is a foreign key since it allows us to find the corresponding (unique) record in the `Regions` table. Foreign keys are described in additional detail in Lesson 6.
4. **Surrogate key**: A surrogate key has no meaning in the domain except as a convenient means for the computer to uniquely identify records. For example, an automatically-generated field called `CustID` is used in the `Customers` table in lieu of a cumbersome concatenated key such as `CustName` + `BillingAddress`.
5. **Secondary key**: A secondary key is a field (or combination of fields) that does not uniquely identify a record, but which is nonetheless useful in searching for a record. An employee’s `OfficePhoneNo` field is an example of a secondary key. Although there may be more than one employee in the organization with the same phone number (e.g., people who share an office), the subset of records with matching values on the secondary key is typically very small relative to the set of all records. Thus, a matching secondary key allows you to narrow your search considerably.

5.4.2 About data types

The field's data type tells ACCESS how to handle the contents of the field. Thus, if a field's data type is Date/Time, then the DBMS can perform date and time arithmetic on the values stored in the field. For example, ACCESS can automatically calculate the number of days between two dates (taking into account the number of days in each month, leap year, and so on). If the same date were stored in a plain text field, ACCESS would treat it just like any other string of characters and would be unable to do date-specific calculations.

Selecting the right data type for your fields is an important (and perhaps long-term) decision. The Year 2000 (Y2K) problem arose because programmers during the early days of building business systems stored the year portion of date fields using two digits (e.g., "79") instead of four digits ("1979"). A two-digit representation was an engineering decision based on two factors: the high cost of memory and disk space at the time and the expected life span of the systems being built. The decision to minimize memory requirements became problematic as the millennium drew to a close because many of these early systems were still in service with large firms such as banks and insurance companies. Unfortunately, computers have difficulty knowing whether the year value "01" corresponds to 1901 or 2001. When performing calculations such as the amount of interest payable on loans, such little details matter.

5.4.3 Data types supported by ACCESS

ACCESS' on-line help system provides detailed information on the data types it supports. The critical information from the help system is summarized in Table 5.3. Despite the large number of choices, the basic data types for all relational database systems can be broken down into four categories: text, numbers, abstract data types, and pointers.

5.4.3.1 Text or character

The text data type is self-explanatory—a text field may contain a string of letters, numbers, or special characters. In older database systems, if you specified 255 characters for a text field called `EmployeeName`, then all 255 characters would be allocated, even if only the first ten or so contained data (the remainder would consist of blank spaces). Like most modern database systems, however, ACCESS allocates space to text fields dynamically.[1] Thus, if you specify a field length of 255 characters, but have an employee with a three-letter name, you do not waste the other 252 characters. Accordingly, the size of the text

[1] In an ANSI-compliant DBMS, the "VarChar" data type is used for variable-length text fields.

TABLE 5.3: Major data types in ACCESS.

Data type	Typical use
Text	text or numbers that do not require calculations (up to 255 characters)
Memo	lengthy text and numbers, such as notes or descriptions (up to 64,000 characters)
Number	data to be used for mathematical calculations (length depends on subtype)
Date/ Time	dates and times
Currency	currency data type (15 decimal places of precision to prevent rounding errors during calculations)
Auto-Number	unique sequential integer automatically inserted when a record is added
Yes/No	Boolean (true/false) values (only one bit is used)
OLE Object	links to objects such as WORD documents, spreadsheets, pictures, etc.

field can be seen as an “upper bound” on the length of a the data to be stored in the field.

If a DBMS supports variable-length text fields, you should err—within reason—on the long side when deciding how long to make your text fields. For instance, if you know that product numbers are generally 10-15 characters long, you may want to set the length of the `ProductID` field to 20 characters. However, setting the length to the maximum number of characters (255) makes less sense.

A major advantage of database systems over **file systems** is **program-data independence**. If you set the length of a field to (say) 20 characters in a database and then discover that the field must accommodate longer values, you can generally modify the length of the field without creating any negative side effects elsewhere in your applications. The exception to this generalization is primary keys: Since primary keys may have corresponding foreign keys in other tables, you will have to manually propagate any changes to a primary key to all dependent foreign keys.

5.4.3.2 Numbers

Numeric data types are typically divided into two major subtypes: **integer** and **real** (or **floating point**) numbers. Recall that integers are whole numbers (e.g., 6, -21) whereas real numbers have fractional parts expressed using decimal notation (e.g., 2.389, -813.2).

Both integers and real numbers are further subdivided based on the capacity or precision of the data that they can contain. For example, the **Integer** data type in ACCESS uses two bytes and can therefore store $2^{2 \times 8}$ = 65,536 unique values (specifically, whole numbers from -32,768 to 32,767).

Obviously, it is important to know what kind of numbers a field can store before you roll out your application. To illustrate, assume that you have been asked to create an employee database for a large company. If you use an Integer data type for the `EmpID` field, but then try to add more than 32,767 employees records to the database, you will run out of unique IDs[1].

To address this problem, ACCESS provides a **Long Integer** data type. Since a Long Integer allocates twice as many bytes of storage as an Integer, it is capable of storing $2^{4 \times 8}$ unique values (numbers from -2,147,483,648 to 2,147,483,647).

[1] Remember that IDs should never be recycled. Thus, even if the company never has more than a few thousand employees at a given time, whenever someone retires or quits, their employee ID is gone forever. With this in mind, 32,767 is a much smaller number than you may initially think.

To be on the safe side, many ACCESS developers opt for Long Integers whenever they create an ID field. Indeed, the AutoNumber type in ACCESS (see Section 5.4.4) is implemented as a Long Integer.

The memory requirement issues for real numbers are similar, except that the problem is generally caused by very small fractional parts rather than very large numbers. The **Single** data type (short for single-precision floating point) allocates four bytes and can represent numbers as small as 10^{-45}. In contrast, the **Double** data type (short for double-precision floating point) allocates eight bytes and can represent numbers as small as 10^{-324}. If you are doing scientific calculations and want to minimize the impact of rounding errors, you should use a double-precision data type for your value.

ACCESS provides a special numeric data type—**Currency**—that is optimized to minimize rounding errors when manipulating fields that contain monetary values.

5.4.3.3 Abstract data types

The Date/Time data type provided by ACCESS is an example of an **abstract data type**. The data type is abstract in the sense that the details of how the dates and times are actually stored in ACCESS are hidden from the designer. Thus, if you use the Date/Time data type, you do not know (or care) whether the year is implemented using two characters or twenty characters—as long as the field is capable of storing time-dependent values and supports special-purpose transformations of the data.

As a point of trivia, the Date/Time data type in ACCESS does not use a fixed number of characters to store the year. Instead, all dates and times are implemented as real numbers in which everything to the left of the decimal refers to day, month, and year and everything to the right of the decimal refers to hours, minutes, and seconds. Of course, since special functions are provided for manipulating and formatting Date/Time fields, you do not need to know anything about the underlying representation.

5.4.3.4 Pointers

A pointer is a field that does not contain data, but instead contains the *address* of data somewhere else in the database or the computer's file system. The Memo and OLE object data types provided by ACCESS are both examples of pointers.

For example, if you want to save a large amount of text in your database (i.e., more than the 255 character maximum permitted by the Text data type), you can use the Memo data type. All that is stored in the actual table is the address of (i.e., a "pointer" to) a large block of text stored elsewhere in the ACCESS database file.

Note that OLE objects are similar to the Memo data type except for two important differences: First, there is no requirement for OLE objects to be ASCII text. Indeed, OLE objects are typically proprietary binary formats such as graphic files, spreadsheets, and so on. Second, the file referenced in the database field does not need to be stored in the database file. That is, the OLE object field can point to an existing file elsewhere on the hard disk.

You are not going to use the pointer data types much in this project. If you are interested in the Memo and OLE object data types, consult the on-line help system for more information.

5.4.4 Choosing a data type

At the most basic level, the choice of data type is straightforward—there is a trade-off between what you can represent and how much it costs (in terms of storage space). However, there are some subtle practical issues that should also be taken into account when selecting a data type for a field. The following are some generic guidelines that you might find helpful:

1. Do not use a numeric data type unless you are going to treat the field as a number (i.e., perform mathematical operations on it). For example, you might be tempted to store a person's employee number (e.g., "58938") as an integer. However, an employee number is really a sequence of numerical characters rather than a number. If the employee number contains dashes or leading zeros, then a numeric data type is clearly unsuitable. A notable exception to this guideline follows.
2. Like most database systems, ACCESS provides a special numeric sub-type called **AutoNumber**

2 The AutoNumber feature is called a **Counter** in ACCESS version 2.0.

An AutoNumber is a Long Integer that is automatically incremented by ACCESS every time a new record is added. Fields that increment automatically are convenient for use as primary keys when no other key is provided or is immediately obvious. That is, AutoNumbers can be used as surrogate keys (recall Section 5.4.1).

5.4.5 Other field properties

5.4.5.1 Field names

ACCESS places relatively few restrictions on field names and thus it is possible to create long, descriptive names for your fields. The problem is that you have to type these field names repeatedly when building queries, macros, and programs. As such, it is best to strike a balance between descriptiveness and ease of typing.

Below are a number of naming issues you should consider when naming your fields:

- Use short (but descriptive) field names *with no spaces*. Although names without spaces may look odd at first, names with spaces (e.g., `Cust Name`) create additional work for you in the long run and make it more difficult to migrate your data to other database systems.

? Some database designers recommend using the underscore character instead of spaces (e.g., `Cust_Name`).

- Like many databases, ACCESS ignores the capitalization of field names. As such, the names `CustName`, `CUSTNAME`, and `custname` are identical as far as the DBMS is concerned.

In ACCESS, I like to use capitalization instead of underscores to distinguish between words in a name, but this is a personal preference.

- Avoid all non-alphanumeric characters other than the underscore and perhaps the dash.

Although ACCESS will permit you to create field names such as `Cust#`, non-alphanumeric characters (such as #, /, $, %, ~, @, etc.) may cause subtle, undocumented problems later on.

- It is becoming easier to "upsize" ACCESS applications to client/server databases (such as ORACLE and SQL SERVER). If there is even a *remote possibility* of having to upsize your application, it is worthwhile to take the time to acquaint yourself with the naming constraints used in the target client/server DBMS.

5.4.5.2 Captions and aliases

In Section 5.3.1 you created a field with the name `CustName`, but used the caption property to provide a longer, more descriptive label (e.g., `Customer name`). The net result is a field name that is easy to type when programming and a field caption that is easy to interpret when the data is shown in datasheet mode.

5.4.5.3 Input masks

An input mask is a means of restricting what the user can type into the field. It provides a template that tells ACCESS what kind of information should be in each space. For example, the input mask `>LL` consists of two parts:

1. The right brace (>) ensures that every character the user types is converted into upper case. Thus, if the user types `bc`, it is automatically converted to `BC`.
2. The characters `LL` are placeholders for letters from A to Z with blank spaces not permitted. What this means is that the user has to type in exactly two letters. If she types in fewer than two or types a character that is not within the A to Z scope, ACCESS displays an error message.

There are many special symbols used for the input mask templates. Since the meaning of the symbols is seldom obvious, and the input mask "language" is ACCESS-specific, there is little value in memorizing the language. Instead, simply place the cursor on the input mask property and press **F1** to get on-line help.

5.4.6 Complex input masks

5.4.6.1 The importance of input masks

In addition to controlling what characters a user can enter, an input mask can automatically enter supplemental characters as the user types. For example, the input mask can be set to add the dash automatically to a North American phone number. This feature ensures that all phone numbers are formatted and stored the same way and is critical if you are using phone number as a secondary key (as far as the database is concerned, 555-8111 is not the same as 5558111 or 555.8111).

The choices provided by the input mask wizard depend on the "regional settings" stored by the operating system. If the wizard does not provide a template for common data types in your region (e.g., social insurance numbers in Canada) use the appropriate Control Panel applet in WINDOWS to change the regional settings.

5.4.6.2 Literal values

To have the input mask automatically insert a character into a field, use a slash to indicate that the character following it is a "literal value". For example, to create an input mask for the local telephone number 555-8111, use the following template:

`000\-0000;0`

The semicolon and zero at the end of this input mask are important because, as the on-line help system points out, an input mask value actually consists of three parts (or "arguments"), each separated by a semicolon:

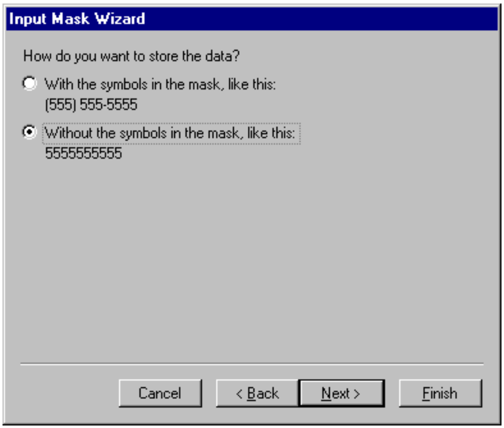
- the actual template (e.g., `000\-0000`),
- a value (0 or 1) that tells ACCESS how to deal with literal characters, and
- the character to use as a placeholder (showing the user how many characters to type).

When you use a literal character in an input mask, the second argument determines whether the literal value is simply displayed or displayed *and stored* in the table as part of the data.

For example, if you use the input mask `000\-0000;1`, ACCESS will not store the dash with the telephone number. Thus, although the input mask always displays the number as "555-8111", the number is actually stored in the database as "5558111". In contrast, if you use the input mask `000\-0000;0`, you are telling ACCESS to store the dash with the rest of the data.

If you use the wizard to create an input mask, it asks you a simple question about storing literal values and fills in the second argument accordingly (see Figure 5.7). However, if you create the input mask manually, you should be aware that by default, ACCESS does not store literal values. In other words, the input mask `000\-0000` is identical to the input mask `000\-0000;1`. This has important consequences if the field in question is subject to referential integrity constraints (the value 555-8111 is not the same as 5558111).

FIGURE 5.7: Deciding whether to store literal values from an input mask



5.4.7 "Disappearing" numbers in autonumber fields

If, during the process of testing your application, you add and delete records from a table with an autonumber field, you will notice that the deleted keys values are not "reclaimed".

For instance, if you add records to your **`Customer`** table (assuming that **`CustID`** is an autonumber), you will have a series of **`CustID`** values: 1, 2, 3, … If you later delete Customers 1 and 2, you will notice that your list of customers now starts at 3.

Although this may seem "untidy", think about what would happen if ACCESS renumbered all records to start at 1. What would happen, for instance, to all the printed invoices with **`CustID`** = 2 on them? Would they refer to the original customer 2 or the newly renumbered customer 2?

The bottom line is this: once a key is assigned, it should never be reused, even if the record to which it is assigned is subsequently deleted.

Thus, as far as you are concerned, there is no way to get your customers table to renumber from **`CustID`** = 1. Of course, there is a long and complicated way to do it. But since you used a surrogate key in the first place, you do not care about the actual value of the key—you just want it to be unique.

5.5 Application to the project

21 Add the remaining values from Table 5.2 to the `Regions` table.

You will finish populating the `Customers` table in Lesson 8.

6.1 Introduction: Extending the scope of your system

In Section 5.3.5, you created a new table called **`Regions`**, even though you do not have a Region entity on the entity relationship diagram (ERD) that you created in Lesson 3.

The decision to add a **`Regions`** table was based on user interface criteria rather than data modeling criteria. Specifically, we decided to use region codes rather than the full region names to minimize the possibility of input errors. However, in order to remember what the one-letter region codes mean, we identified the requirement for a look-up table called **`Regions`**.

You will probably create a lot of look-up tables when you implement your databases. Look-up tables always have the same format:

- a short code or numeric ID that is appropriate for use as a value in other tables; and
- a description or name for the code that can be "looked up" so that users do not have to interpret (or even see) the short codes.

Some designers like to include look-up tables as entities on their ERDs. Doing so maintains a consistent mapping between entities and tables. Personally, I find this practice clutters the ERDs and creates a lot of extra work for the person creating the diagram. Instead, I keep the "obvious" look-up tables off the ERDs, but add them to the database as required. Naturally, if you are using a CASE tool for physical design and to generate your database, you need to include all the entities on the diagram.

6.1.1 Death, taxes, and scope creep

Now that you have a table called **`Regions`** implemented, it might occur to someone (in this case, the someone is you wearing your user/manager hat) that it would be a good idea to store information about the sales representatives who are responsible for each region. In this way, you can use the order entry system to create reports showing performance-related information such as total sales for each sales rep, annual change in sales for each rep, and so on.

Adding new requirements and functionality to an information system once implementation is a common form of **scope creep**. In this case, you want to expand the scope of the order entry system by including information about sales people. This is not an unreasonable demand. Since we already have the `Regions` table in place, it is a simple matter of associating each region with its assigned representative.

The next link in the scope creep chain of reasoning is: “Since we have employee information, we might as well add payroll and benefits information.” At some point, scope creep leads to a project that is too large and too complex to ever be completed. As a consequence, it is usually a good idea to resist scope creep and get a small system up and running before expanding it to address other problems and include “nice-to-have” features.

6.1.2 Leveraging existing data assets

To make the addition of sales rep information to the order entry system interesting, we are going to assume that you already have all your employee information in electronic form in an off-the-shelf payroll system that you bought a few years ago. Furthermore, we are going to assume that you do not know much about the payroll system, except that it contains information about all your employees (including sales reps) and that a great deal of effort is routinely expended to ensure that the employee information in the payroll system is up to date. Given the turnover you experience with sales reps, you have no interest in storing and maintaining this data in two separate systems.

6.1.3 The relationship between customers, regions, and employees

Before we dive into implementation issues, it is worthwhile to return to our ERD and ensure we understand how salespeople fit into the grand scheme of operations in your kitchen supply company.

The current policy within the company is to assign each customer to one of your six sales regions (North, South, etc.). Each region can contain multiple customers, but customers never belong to more than one region.

Each sales region is the responsibility of a single sales rep. That is, if something is wrong in a region, there is a single employee who acts as the point of contact. In some cases, however, a single sales rep can be assigned responsibility for more than one region. You try to avoid this situation, but it arises from time to time when sales reps leave the company on short notice, go on maternity leave, and so on.

Later in this lesson, you will modify your ERD from Lesson 3 to include these entities and relationships.

6.1.4 The infrastructure for one-to-many relationships

As discussed in Section 3.1.2.4, one-to-many relationships are the bread and butter of relational databases. Not only do one-to-many relationships occur frequently in business contexts, but all many-to-many relationships must ultimately be decomposed into one-to-many relationships for implementation using a relational database system.

The relational database model cannot represent many-to-many relationships directly. Instead, you must transform each many-to-many relationship into an associative entity, as discussed in Section 3.3.2.2.

The procedure for implementing a one-to-many relationship in a relational database is always the same: you take the primary key from the table on the “one” side of the relationship and include it in the table on the “many” side of the relationship.

To illustrate, recall the **Customers** and **Regions** tables you created in Lesson 5: Look-up tables are always on the “one” side of a one-to-many relationship. In this case, each customer can belong to (at most) one region, but each region can contain many customers. To implement the one-to-many relationship in the tables, we do the following:

1. Identify the primary key from the “one” side of the relationship—in this case, it is the **RegionCode** field in the **Regions** table.
2. Add a field with the same data type as the primary key to the table on the many side of the relationship—in this case, the **Customers** table.

The field **Customers.RegionCode** is called a **foreign key**. It is “foreign” in the sense that it is the primary key of the **Regions** table, not the **Customers** table.

It is common practice to express table names and field names using “dot” notation: ***<table name>.<field name>***. In this way, it is possible to distinguish between fields with the same name in more than one table (e.g., **Regions.RegionCode** and **Customers.RegionCode**).

The operation of the foreign key is shown in Figure 6.1: If we know that SAM’S STOCK POT has been assigned to the region with **RegionCode** = “C”, then it is a simple matter to go to the **Regions** table and look up the corresponding record. The values of **RegionCode** are guaranteed to be unique in the **Regions** table so there can be no confusion or ambiguity regarding the region to which Sam's Stock Pot has been assigned.

FIGURE 6.1: The ***RegionCode*** field in the ***Customers*** table is a foreign key. It permits us to associate a region with every customer.

Customer ID	Customer name	RegionCode	Contact person
1	Sam's Stock Pot	C	Sam Wong
2	Loonie Mart #107	K	Bill Williams
3	Rosch Dry Goods Inc.	K	Alice McRorie
4	Gadgets "R" Us	C	Leslie Cranfield-Jones
5	The Chef's Assistant	N	Andre Oulett
(AutoNumber)			

RegionCode	Region
C	Central
E	East
K	Key Accounts
N	North
S	South
W	West

As you will see in the lessons on relationships (Lesson 7) and join queries (Lesson 10), the look-up procedure described above is automated by the DBMS. That is, you never have to manually make the connection between the foreign key in one table and the primary key in another. This is one of the core strengths of the relational database model.

6.2 Learning objectives

- understand how to implement one-to-many relationships using foreign keys
- update your ERD to include cardinality constraints
- modify an existing database to include foreign keys
- understand the concept of a weak entity

6.3 Exercises

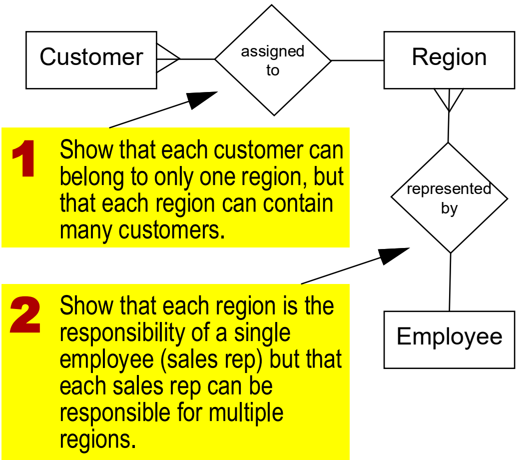
6.3.1 Adding new entities

The first step in adding employee information to the database is to update your ERD.

1 Add two new entities to your ERD: Region and Employee.

2 Create the one-to-many relationships shown in Figure 6.2.

FIGURE 6.2: Update your ERD to show the relationship between Customer, Region, and Employee.



The cardinality of the relationships in Figure 6.2 follow directly from the company policies outlined in Section 6.1.3. For example, the constraint that each region is represented by at most one sales rep is an idiosyncratic feature of this particular company, not an immutable law of the universe. Another company might organize its sales function differently and therefore require a different set of modeling assumptions.

6.3.2 Cardinality constraints

You may be wondering about the choice of the name “Employee” instead of “Sales Rep” in Figure 6.2. You know that your payroll system contains information about all your employees. As such, you can assume that the Employee entity has already been implemented within the organization. It makes sense to reuse this entity and retain any existing naming conventions rather than introduce an entirely new entity called “Sales Rep”.

Using the Employee entity leads to a different problem: not all Employee entities participate in a relationship with the Regions entity. That is, only employees belonging to the sales function of the firm are assigned sales regions. Unfortunately, the ERD in Figure 6.2 does make the distinction between salespeople and other employees explicit.

6.3.2.1 Mandatory versus optional participation

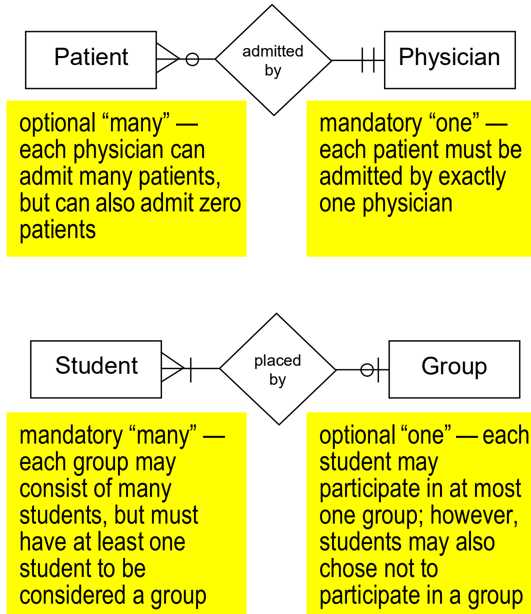
In the Entity-Relationship model, a construct called cardinality constraints can be used to differentiate between “mandatory” and “optional” participation in a relationship. To this point, all we have said about the relationship between the Employee and Region entities is that each employee can represent “many” regions. However, we may want to use ERD notation to indicate that “many” includes zero—that is, it is okay for an employee (e.g., the warehouse manager) to have no sales region.

There are other situations, however, in which we want to indicate mandatory participation in a relationship. For example, in a hospital environment, each patient is admitted by exactly one physician (that is, one physician takes initial responsibility for the patient). It is never the case the more than one physician admits a patient and it is certainly never the case that a patient is admitted without being assigned to a physician.

6.3.2.2 ERD notation

One notational convention for indicating cardinality constraints on ERDs is shown in Figure 6.3 (there are many other notations, but the underlying concept is always the same). Although including the little lines and circles on the relationships makes the ERDs more complex and harder to explain to users and managers, the designation of a relationship as “mandatory” or “optional” can have important consequences during implementation.

FIGURE 6.3: Examples of “optional” or “mandatory” participation in relationships.



When you give your physical ERDs to implementors to build, you should make sure that all your design decisions are explicit. For example, you do not want a contract database developer working on a hospital information system deciding on her own whether it is okay for a patient to be admitted without a doctor's order.

6.3.2.3 Adding cardinality constraints

3 Use the notation shown in Figure 6.3 to show that each region must have exactly one employee representing it, but that all employees do not necessarily participate in a sales rep role.

6.3.3 Adding foreign keys

It might not be immediately obvious, but the status of the Region entity has changed. Initially, information about regions was implemented as a simple look-up table and thus the Region entity was not included as a *real* entity on the ERD. But now that the link between employees and customers is being made, Region is a *bona fide* entity—that is, it corresponds to something identifiable and important in the domain we are modeling.

Fortunately, the foreign key infrastructure between the `Customers` and `Regions` tables is already in place because the `Customers` table already contains a the `RegionCode` foreign key. However, the same is not true of the foreign key infrastructure between the `Regions` and `Employees` table. Since `Regions` is on the many side of the relationship, it requires a foreign key. But what is the primary key of the `Employees` table?

To make things interesting, we are going to assume that the primary key of the `Employees` table in the payroll application is the combination of the employee's first names (a field we know is called `emp_fname`) and last name (a field we know is called `emp_lname`). Of course, we recognize that this particular combination of fields is a poor choice for a concatenated primary key because it is (a) not guaranteed to be unique, and is (b) long and unruly.

Despite the obvious shortcomings of our primary key, we will stick with it for now and make changes as we learn more about the structure of the payroll data in Lesson 8.

4 Open the `Regions` table in design mode and add two fields: `emp_fname` and `emp_lname`.

5 Set the data type of the fields to text and the length to some suitably large value (e.g., 100).

At this point, we do not know enough about the structure of the two fields in the `Employees` database to match the data type and lengths exactly. Thus, we will make educated guesses for now and make modifications as required.

6 Since the use of `emp_fname` and `emp_lname` as a foreign key is provisional, do not spend any time specifying optional field properties.

7 Save and close the `Regions` table.

You now have the foreign key infrastructure for creating relationships between customers, regions and employees. In Lesson 7, you will learn how to make the relationships explicit in ACCESS.

6.4 Discussion

6.4.1 Concatenated keys

In this lesson, you added a concatenated foreign key in the `Regions` table. A concatenated foreign key is identical to a single-field foreign key except that multiple fields are used to make the link between the "one" and "many" sides of the relationship.

To illustrate why a concatenated key is used, consider the Region and Employee entity: If the employee in charge of the East region is Bill Williams, then the field `emp_fname` is probably insufficient as a foreign key since there may be more than one employee named "Bill" in the organization. However, at this point in time, it turns out that the combination of `emp_fname` + `emp_lname` (e.g., "Bill Williams") is unique in the `Employees` table. Of course, if you hire Bill's son, Bill Jr. to work in the warehouse, your system will break and need to be fixed.

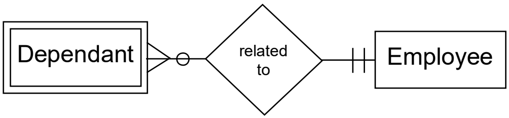
You should take some care when selecting the primary keys for you table. If existing fields cannot be guaranteed to be unique in the long term, you should opt for a surrogate key such as an AutoNumber.

6.4.2 Weak entities

Consider the relationship between the Dependant and Employee entities shown in Figure 6.4. Knowing who an employee's dependants are is important for benefits, such as heath and dental insurance. However, if a particular employee leaves the company, then the Dependant entities associated with that employee cease to be of interest to the organization (from an information system perspective).

In this example, Dependent is called a weak entity. It is *weak* in the sense that each instance of the entity relies on an instance of another entity (in this case, Employee) for existence in the database. If the *strong* instance is removed from the database, then all corresponding instances of the weak entity should also be removed from the database.

FIGURE 6.4: An ERD for employee benefits. Note that Dependant is a weak entity.



Weak entities are often designated with a double rectangle.

There is one feature of weak entities that often causes confusion when determining foreign keys: the foreign key of a weak entity is also *part of its primary key*.

To illustrate, assume the table schema for the **`Dependants`** table is as follows:

`Dependants(emp_id, index, name, relationship, date_of_birth, …)`

The standard way to write a table schema is ***Table name*(*key field*$_1$*, key field*$_2$*, …, field*$_{n-1}$*, field*$_n$)**. The field or fields that make up the primary key are typically underlined.

To make the table schema more concrete, assume that we have the following information about the dependents of two employees, Russell Plevy (**`emp_id`** = 8) and Vivian Peng (**`emp_id`** = 2):

TABLE 6.1:

emp_id	index	name	relationship
8	1	Alica Marie Plevy-Jones	wife
8	2	Russell James Plevy	child
8	3	Susan Ann Plevy	child
2	1	Martin Alec Peng	husband

The combination of Russell's employee ID and the index number 2 provides a convenient means of uniquely identifying Russell Jr. Thus, **`emp_id`** is half of the table's concatenated primary key in addition to being the foreign key that links the **`Dependants`** and **`Employees`** table.

6.5 Application to the project

Based on what you have learned in this lesson and Lesson 5, you are now ready to add some more tables to your database:

8 Add an **`Orders`** table to store the details of customer orders that you receive.

9 Ensure you use an AutoNumber for the `Orders.OrderID` field. In this way, ACCESS will automatically generate a unique `OrderID` every time you create a new order.

10 Create an `OrderDetails` table that uses `OrderID` as a foreign key (each order detail belongs to exactly one order, but each order can have many order details).

Since it is a foreign key, each value of `OrderDetails.OrderID` must refer to an existing value of `Orders.OrderID`. Thus, you cannot set `OrderDetails.OrderID` as an AutoNumber because doing so would create a new value of `OrderID` each time a new *order detail* is added. This is not what you want.

HINT: For a particular order, each product should appear only once as an order detail. Thus, rather than having one order detail for six "Fat Cat" mugs and a separate order detail for a dozen more "Fat Cat" mugs a few lines down in the same order, a single order detail for 18 should be used to consolidate the two. Given this policy, it is possible to set the primary key for the `OrderDetails` table to `OrderID` + `ProductID`.

Like all associative entities, Order Detail is also a weak entity because it relies on both the Order and Products entities for existence. As such, it is natural that `OrderDetails.OrderID` and `OrderDetails.ProductID` be foreign keys in addition to being the table's primary key.

7.1 Introduction: The advantage of "normalization"

A common mistake made by inexperienced database designers (or those who have more experience with spreadsheets than databases) is to ignore the recommendation to model the problem in terms of entities and relationships and to put all the information they need into a single, large table. Figure 7.1 shows such a table containing information about customers, regions, and salespeople.

The advantage of the single-table approach is that it requires less thought during the initial stages of application development. The disadvantages are too numerous to mention, but some of the most important can be illustrated using the small amount of data shown in Figure 7.1:

1. **Insertion anomaly** — Assume that we want to create a new sales region called "On-line". The sales rep for the new region would be responsible for any Internet-based selling initiatives. However, a new region cannot be added to the table unless a valid customer ID is specified. Since the On-line region does not have any customers at this point, the only way to add a new region is to introduce a "dummy" customer.

FIGURE 7.1: The "monolithic" approach to database design.

The table combines information about customers and regions

Customer name	RegionCode	Region	RepID	EMP_FNAME	EMP_LNAME
Sam's Stock Pot	C	Central	9	Ben	Sidhu
Gadgets "R" Us	C	Central	9	Ben	Sidhu
Loonie Mart #107	K	Key	9	Ben	Sidhu
Rosch Dry Goods Inc	E	East	3	Jocelyn	Scorer
The Chef's Assistant	N	North	4	Bill	Williams

The "Central" region contains more than one customer.

Sales reps are assigned to regions, not customers.

2. **Deletion anomaly** — Assume that THE CHEF'S ASSISTANT goes out of business and you delete its record from the table. Since THE CHEF'S ASSISTANT is the only customer assigned to the North sales region, deleting the customer also wipes out the only record you have that Bill Williams is the sales rep for the North region.
3. **Modification anomaly** — Suppose that Sabine Villeneuve takes over responsibility for the Central region. Since each region can have multiple customers, you have to make the change for all customers assigned to the Central region. Not only is this extra work, it creates the potential for inconsistent data. What happens, for example, if you forget to change some of the records? Is the sales rep for the Central region Sabine Villeneuve or Ben Sidhu?

7.1.1 Normalized table design

The anomalies identified above can be avoided by splitting the table in Figure 7.1 into three separate tables:

- **`Customers`** — information about customers only,
- **`Regions`** — information about regions only, and
- **`Employees`** — information about employees only.

Once the separate tables are created, what is needed is a means of linking to the tables together. As you saw in Lesson 6, linking tables in a relational database is accomplished through the use of foreign keys. For example, we use **`Customers.RegionCode`** to create a link to the appropriate record in the **`Regions`** table. Once we know a customer has a **`RegionCode`** = "C", we can switch to the **`Regions`** table, find the correct record, and determine everything we need to know about the region. Since other customers in the Central region point to the same record in the regions table, information about the region (such as its name, the sales rep who is responsible for the region, and so on) is stored in exactly one place.

A database schema in which every entity has its own table and redundancy is minimized is said to be **normalized**. The redundancy that does exist in a normalized table structure is minimized by using very short fields (e.g., **`RegionCode`**) to link the tables.

The science of normalization is a bit more complex than this quick summary. However, a common-sense understanding of the concept is all we require for the purposes of this project.

7.1.2 Making relationships explicit

In Lesson 6, you created the foreign key infrastructure for several one-to-many relationships. In this lesson, you are going to *declare* the relationships between tables explicitly and tell ACCESS to enforce **referential integrity**. Briefly, referential integrity is a tool that helps minimize the amount of garbage that gets entered into your database. A more formal definition of this important database concept can be found in subsequent sections.

7.2 Learning objectives

- understand how to create relationships in ACCESS
- edit and delete relationships
- create a relationship for a concatenated key
- understand referential integrity and its relationship to business rules

7.3 Exercises

Once you are 100-percent happy with your table structure, and before you populate the tables or go on to create queries and forms for you application, you should use the **relationships window** to formally declare the relationships between your tables.

Although primary keys and foreign keys are all you need to implement relationships in a relational database, the DBMS does not “know” about the relationships until you declare them. In ACCESS, you declare relationships by dragging and dropping within the relationships window.

7.3.1 Creating a relationship

Recall that the relationships window in ACCESS is similar to an entity-relationship diagram (see Figure 7.2). In this section, you are going to use the relationships window to specify a one-to-many relationship between the `Regions` and `Customers` tables.

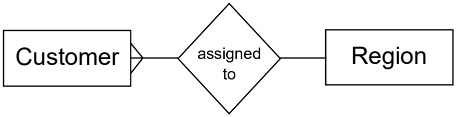
Show me (lesson7-1.avi)

1 Make sure the database window is in the foreground and select **Tools** → **Relationships** from the main menu.

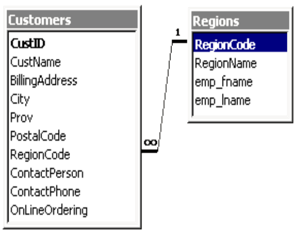
Remember, you can always bring the database window to the foreground by selecting **Window** → **<***database name***>** from the main menu.

At this point, the relationships window should be empty.

FIGURE 7.2: Comparison of an entity-relationship diagram and the relationship feature in ACCESS.



"Each customer is assigned to one and only one region. Each region may be assigned any number of customers, including zero."



2 To add tables, select **Relationships** → **Show Table** from the menu or press the show table icon ( ) on the toolbar.

As shown in Figure 7.3, a dialog box pops up that allows you to select one or more tables and add them to the relationships window.

3 While holding down the **Ctrl** key, click on the `Customers` and `Regions` tables.

4 Press the **Add** button and close the dialog box.

The dialog box in Figure 7.3 is known as a **modal dialog box**. If a dialog is modal, it does not permit you to do anything else within the program until the dialog box is closed (by pressing **OK**, **Cancel**, **Close**, or whatever choices are offered).

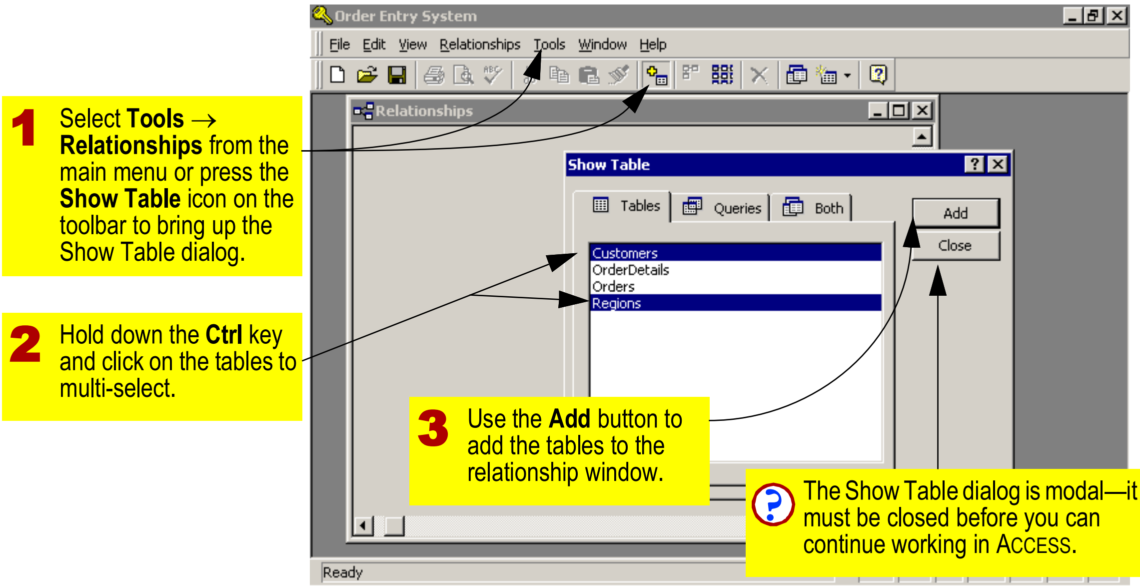
Once your tables have been added to the relationships window, you tell ACCESS which primary key/foreign key fields are used link the tables together. The general procedure is to drag the primary key from the "one" side of the relationship on to the foreign key on the "many" side of the relationship.

Although it should not matter, the sequence described above (drag from the "one" side to the "many" side) appears to be critical in some situations.

5 If necessary, resize the field list for the `Customers` table so that the `RegionCode` field is visible.

6 Select the `RegionCode` field from the `Regions` table (primary key on the "one"

FIGURE 7.3: Select the tables to add to the relationships window.



side) and drag it onto the **RegionCode** field in the **Customers** table (the foreign key on the “many” side), as shown in Figure 7.4.

At this point, the relationship properties box should pop up, as shown in Figure 7.5.

7 Ensure that the correct field(s) are used for the relationship.

For single-field relationships, you only have to make changes in the properties dialog if you have made a mistake dragging and dropping. For multiple-field relationships, however, you must always specify the related fields manually.

FIGURE 7.4: Drag the primary key on the “one” side of the relationship on to the foreign key on the “many” side of the relationship.

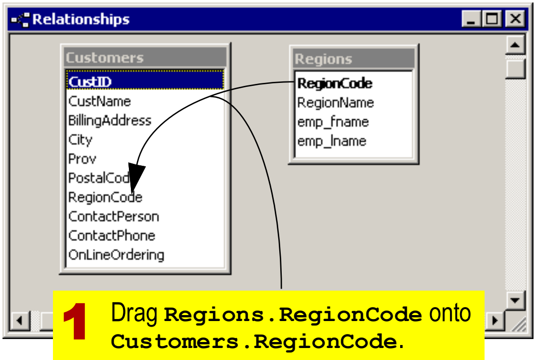
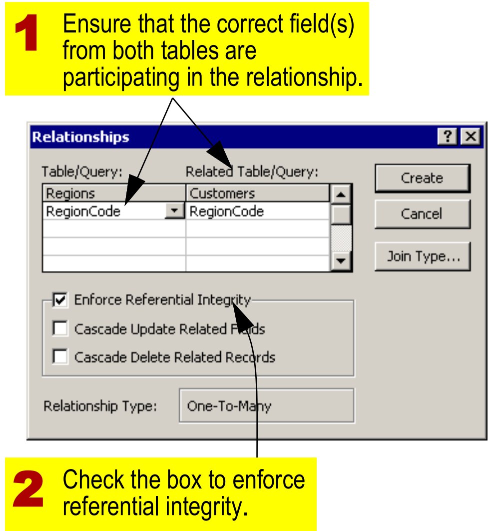


FIGURE 7.5: Set the properties of the relationship between ***Regions*** and ***Customers***.



8 Ensure referential integrity is enforced (see Section 7.4.3 in the discussion for more information on referential integrity).

When you are done, your relationship should look like the one in the bottom half of Figure 7.2.

7.3.2 Editing a relationship

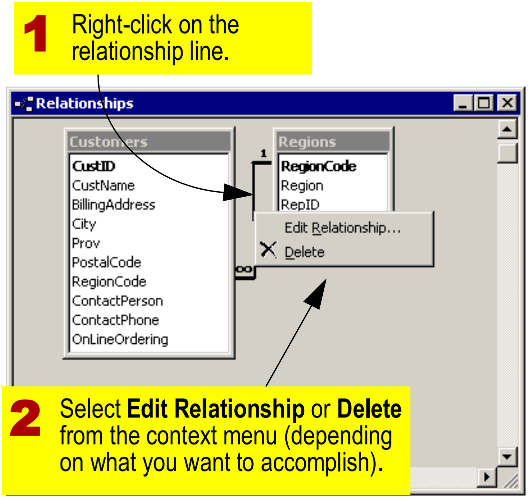
To edit or delete an existing relationship, you simply right-click on the relationship line and select from the resulting context menu.

Show me (lesson7-2.avi)

9 Right click the relationship line. The context menu, shown in Figure 7.6, should appear.

10 Select **Edit Relationship** to get the relationships property dialog.

FIGURE 7.6: To edit a relationship, use the context menu.



You cannot modify a table's relationships if the table is open. Because of this, it is good practice to close all windows except the database window before opening the relationship window. If you forget to do this, simply close the relationship property sheet, close the table in question, and try again.

7.4 Discussion

7.4.1 Creating a relationship using a concatenated key

The foreign key linking the `Customers` table with the `Regions` table is a single field: `RegionCode`. However, recall that if the primary key on the "one" side of the relationship is concatenated, the foreign key will also be concatenated. For example, the foreign key currently used to link the `Regions` table with the `Employees` table consists of two fields: `emp_fname` and `emp_lname`.

Since you do not have access to the `Employees` table at this point (it is coming in Lesson 8), you cannot practice creating relationships for concatenated key yet. However, the process is identical to that described in Section 7.3.1 with a couple of minor changes:

1. When selecting the fields on the "one" side (recall Figure 7.4), use the **Ctrl** key to multi-select all the field in the primary key.
2. When verifying the linking fields (recall Figure 7.5), you may have to set the linkages for the fields manually.

We will revisit the procedure for declaring concatenated foreign keys in Section 8.3.4.1 once we have the `Employees` table in place.

7.4.2 Populating tables on the “many” side

By enforcing referential integrity when you declare relationships, ACCESS prevents you from creating meaningless links when you enter data into the tables. Specifically, ACCESS prevents you from entering a value in the foreign key on the “many” side that does not exist in the primary key on the “one” side. As a consequence, enforcing referential integrity affects the order in which you can populate tables.

For example, you cannot add a complete `Customer` record until the `Regions` table is populated. This is because values for `RegionCode` in the `Customers` table must correspond to valid values of `RegionCode` in the `Regions` table. If no valid regions are defined, `RegionCode` in the `Customers` table must be set to NULL for all customers.

NULL is a special value in database terminology that means “the absence of a value”. Thus, when you are asked to set a field’s value to NULL, it does not mean you type in the letters N-U-L-L. Nor does it mean you enter a blank space or a zero. NULL means empty.

Since you populated the `Regions` table in Section 5.3.6, you are now in a position to populate the `Customers` table (including the `Customers.RegionCode` field).

One way to populate the `Customers` table is to go through the orders that have been faxed to you and copy the customer information from the order headers. To spare you this drudgery, however, you will learn how to take a spreadsheet containing this information and append it to your `Customers` table in Lesson 8.

7.4.3 Referential integrity

An important feature of modern DBMS systems is that they support enforcement of referential integrity at the table level. What is referential integrity? Essentially, referential integrity means that every record on the “many” side of a relationship must have a corresponding record on the “one” side (or else be NULL-valued). Enforcing referential integrity means that you cannot, for instance, create a new record in the `OrderDetails` table without having a record with the same value of `OrderID` in the `Orders` table.

In addition, referential integrity prevents you from deleting records on the “one” side if related records exist on the “many” side. This eliminates the problem of “orphaned” records created when parent records are deleted.

Referential integrity is especially important in the context of transaction processing systems. Imagine that someone comes into your store, makes a large purchase, asks you to bill customer number 123, and leaves. What if your order entry system allows you to create an order for customer 123 without first checking that such a customer exists? If you have no customer 123 record, where do you send the bill?

In systems that do not automatically enforce referential integrity, these checks have to be added to the application using a programming language. This is just one example of how table-level constraints can save you programming effort.

7.4.4 Numeric foreign keys

Unwanted referential integrity errors sometimes occur when you use a numeric (e.g., long integer) field for a foreign key. Recall from Figure 5.2 that each field has a **Default** property that you can specify when creating the table. When a new record is added to the table, ACCESS sets the field to the default value.

The problem is that ACCESS automatically assumes that the default value for numeric fields is zero (the default is NULL for non-numeric fields). However, you often do not have a value of zero in the table on the "one" side of the relationship.

To illustrate, consider the foreign key used to link your `Customers` and `Orders` tables. If you have used an AutoNumber for the `CustID` field in your `Customers` table, your customers will have `CustID` = 1, 2, …. However, when you add a new record to the `Orders` table, the value of the foreign key, `Orders.CustID`, will be set by ACCESS to its default value of zero. Since you do not have a record with `CustID` = 0 in the `Customers` table, a referential integrity error occurs unless you change the value of `Orders.CustID` to some other value.

There are two ways to get around this problem:

1. Set the **Default** property of any numeric foreign keys to NULL (i.e., leave the property blank) instead of zero when you are designing the table. In this way, "unknown" values will be NULL (which is perfectly legal), instead of zero.
2. Leave the default property set to zero, but create a record in the table on the "one" side with a primary key of zero. For example, create a dummy customer called "unknown" with a `CustID` = 0. In many ways, this approach is preferable to (1) above because NULL values are ambiguous. They can indicate "not applicable", "unknown", "incomplete", and so on.

If the primary key on the "one" side of the relationship is an AutoNumber, you cannot add a ***<primary key>*** = 0 record to your table (AutoNumbers start at one). But there is no reason that you cannot use a different number. For example, you designate `CustID` = 1 the "unknown" customer and set the default property for `Orders.CustID` to one.

7.5 Application to the project

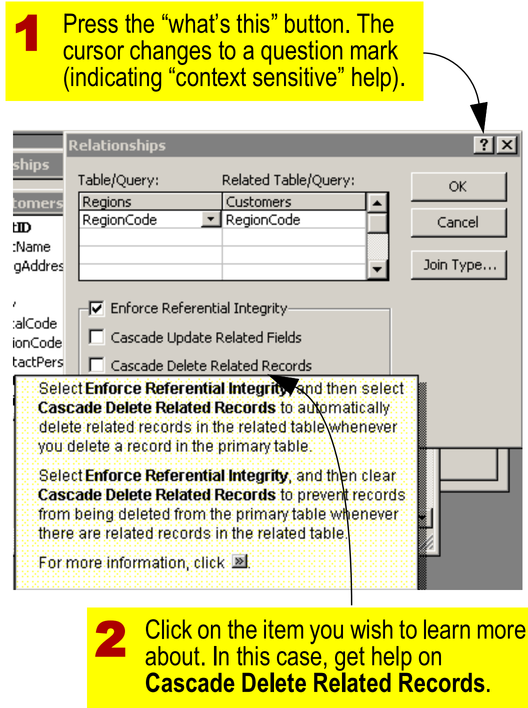
11 Declare relationships for all the tables you have created so far. Make sure that referential integrity is enforced in every case.

12 When the relationship properties window is open, press the "what's this" button and click on the "Cascade Delete Related Records" check box, as shown in Figure 7.7. This brings up the context-sensitive help box. Press the "more information" button at the bottom to read more about cascading deletes.

13 For the relationship between `Orders` and `OrderDetails`, check the box to specify cascading deletes (see Figure 7.5).

Whenever you are dealing with a **weak entity** (an entity which depends on another entity for existence), it often makes sense to specify cascading deletions. In this case, the concept of an `OrderDetail` record without a corresponding `Order` record is meaningless.

FIGURE 7.7: Use context-sensitive help to learn about cascading deletes



8.1 Introduction: Using existing data

There are a number of different ways to create and populate a table:

- Create the table definition from scratch and then populate the table manually with data values (as you did in Lesson 5).
- Create the table definition from scratch and then *append* data from some other electronic format to the table.
- *Import* the table definition and data from another database or application (such as MICROSOFT EXCEL or a text file).
- Create a link to an external data source. In this case, the data is not actually stored in your database file, but is accessible from within ACCESS like any other table.

The first approach (populating the tables from scratch) can be seen as a last resort. The data that you need to build an application often exists in electronic format somewhere already. For example, if you are "downsizing" an application from a mainframe environment to a desktop DBMS, the mainframe data can probably be extracted and saved as plain text files.[1] Similarly, if you are upsizing from a chaotic spreadsheet-based system, then it may be possible to import the spreadsheets directly into the database. Obviously, getting the data in electronic format can save an enormous amount of time and avoid many keyboarding errors.

Of course, there is one important caveat: Mainframe reports and spreadsheets are seldom in normalized form. Consequently, conversion into sensible database structure may involve a certain amount of manual reorganization and transformation.

In this lesson, you will explore techniques for importing data and linking to existing data sources. The general objective of introducing these skills is to permit you to take advantage of existing sources of electronic data whenever possible.

[1] To get the data in the format you require in a mainframe environment, you often have to be very nice to the individual who controls the central computer (i.e., the geek in the glass room). This in itself is often sufficient justification for downsizing.

8.2 Learning objectives

- append data from a spreadsheet to an existing table
- import data from text file into a new table
- create a linked table using data from a non-ACCESS database

8.3 Exercises

8.3.1 Appending data from a spreadsheet

In the tutorial package, you will find a MICROSOFT EXCEL spreadsheet named **`CustList.xls`**. The spreadsheet contains a list of your customers.[1]

1 Open the **`CustList.xls`** spreadsheet in EXCEL.

2 Compare the columns of data in the spreadsheet to the field names you used in Section 5.3.1 when creating the **`Customers`** table (i.e., open your **`Customers`** table in design view).

[1] For the purpose of these lessons, we are working with a very small amount of data. For example, your entire customer base is assumed to consist of five firms. Rest assured that the theory and techniques described in the lessons are independent of the amount of data you are dealing with (within reason).

8.3.1.1 Spreadsheet preparation

Notice that the spreadsheet headings are *similar* to the field names of the **`Customers`** table. There are, however, two exceptions:

1. The spreadsheet does not contain any information about regions or whether the customer has been authorized to conduct on-line ordering. In contrast, the table has a **`RegionCode`** field and an **`OnLineOrdering`** field.
2. The spreadsheet contains a column named “Billing Address” (with a space between the words) whereas the database field is called “BillingAddress” (no space between the words in accordance with the field naming conventions introduced in Section 5.4.5.1).

To append records from a spreadsheet into an existing table, the first row of the spreadsheet must contain column names that match the field names of the target table exactly. The import wizard uses the column name information to ensure the data ends up in the right place.

3 Edit the spreadsheet so that the heading “Billing Address” reads “BillingAddress” (one word) as shown in Figure 8.1.

Since each column of data in the spreadsheet is labeled, there is no requirement to have “dummy columns” for missing fields. For example, there is no need to insert a blank column in the spreadsheet and label it “RegionCode”.

4 Save and close the modified spreadsheet.

8.3.1.2 Using the import wizard

Show me (lesson8-1.avi)

5 Return to ACCESS, ensure the database window is in the foreground, and select **File** → **Get External Data** → **Import** from the main menu.

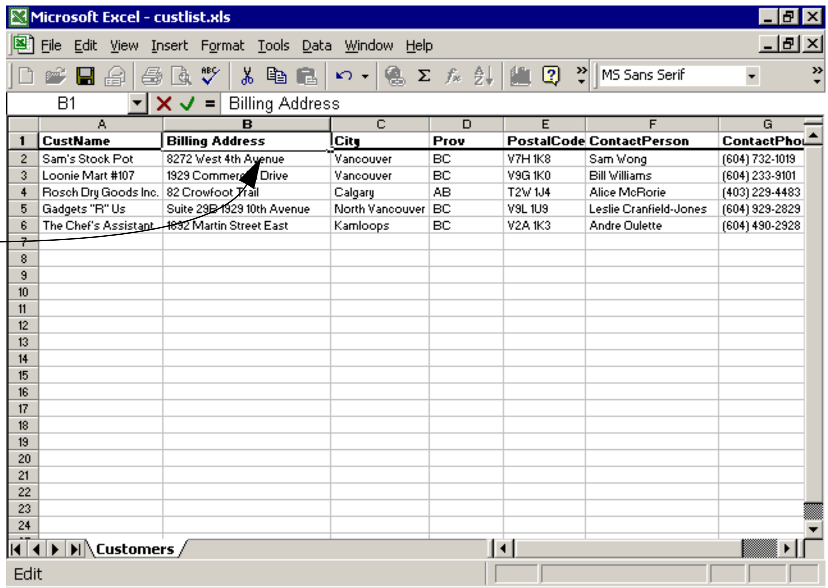
6 In the “Open” dialog, select the `CustList.xls` spreadsheet as the source of the data to be imported, as shown in Figure 8.2.

FIGURE 8.1: Modify the contents of the spreadsheet that contains the customer data.

1 Open the CustList.xls file in EXCEL.

2 Edit the column name “Billing Address” so that it conforms to the naming convention you used when creating the `Customers` table in ACCESS.

3 Save and close the spreadsheet.



7 Follow the instructions provided by the wizard to complete the import process, as shown in Figure 8.3 and Figure 8.4.

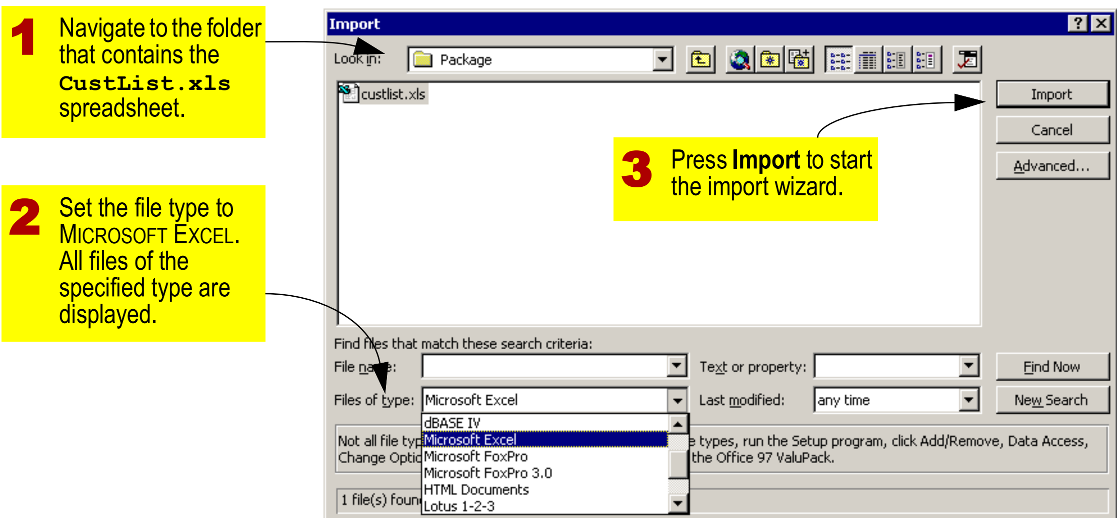
You should get a message reporting that the import process was successful.

If there is a problem with the import wizard, the error message provided by ACCESS is not particularly helpful. All you can do is insure the column headings in your spreadsheet match the field names in the target table and try again.

8 Open your `Customers` table in datasheet view to verify that the data has been transferred correctly.

By importing the data from the spreadsheet, you have created a completely independent copy of the data. Thus, if a customer address is changed in the spreadsheet, it must also be

FIGURE 8.2: Locate and open the EXCEL workbook containing the customer data you wish to import.



changed in the **Customers** table. For this reason, importing works best when the database application is meant to *replace* the spreadsheet-based application, rather than coexist with it.

FIGURE 8.4: Use the import wizard to append records to the ***Customers*** table (part 2).

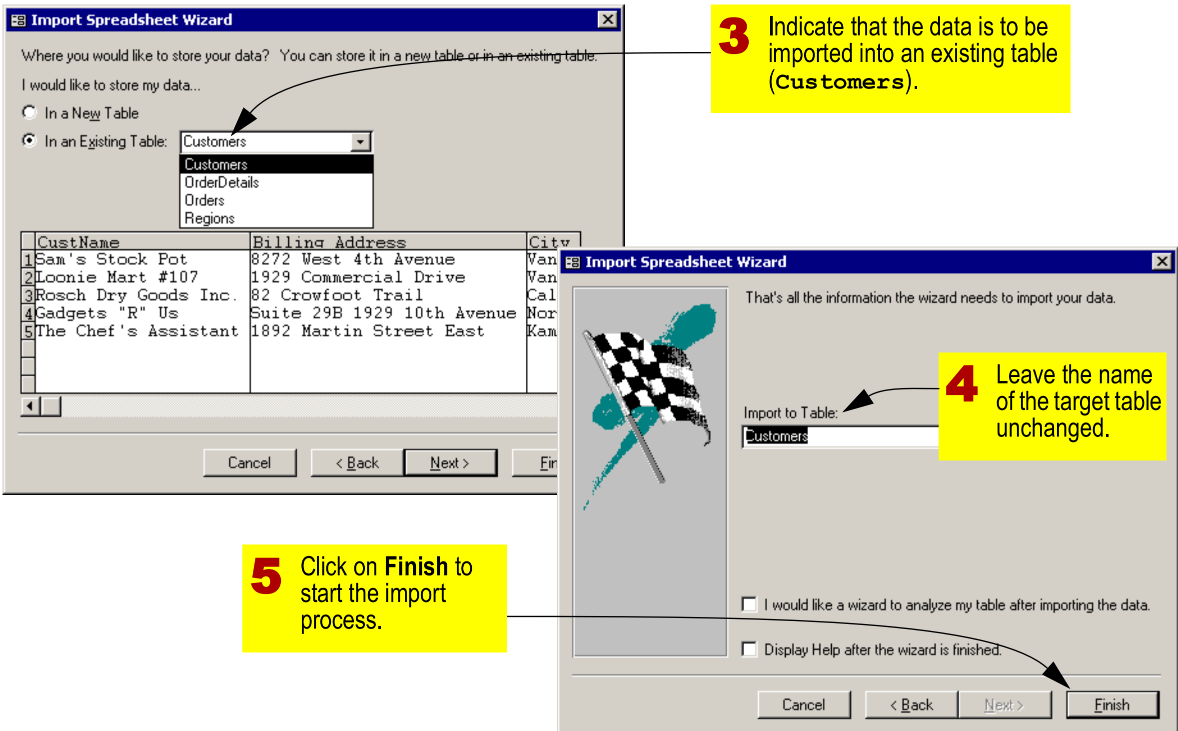
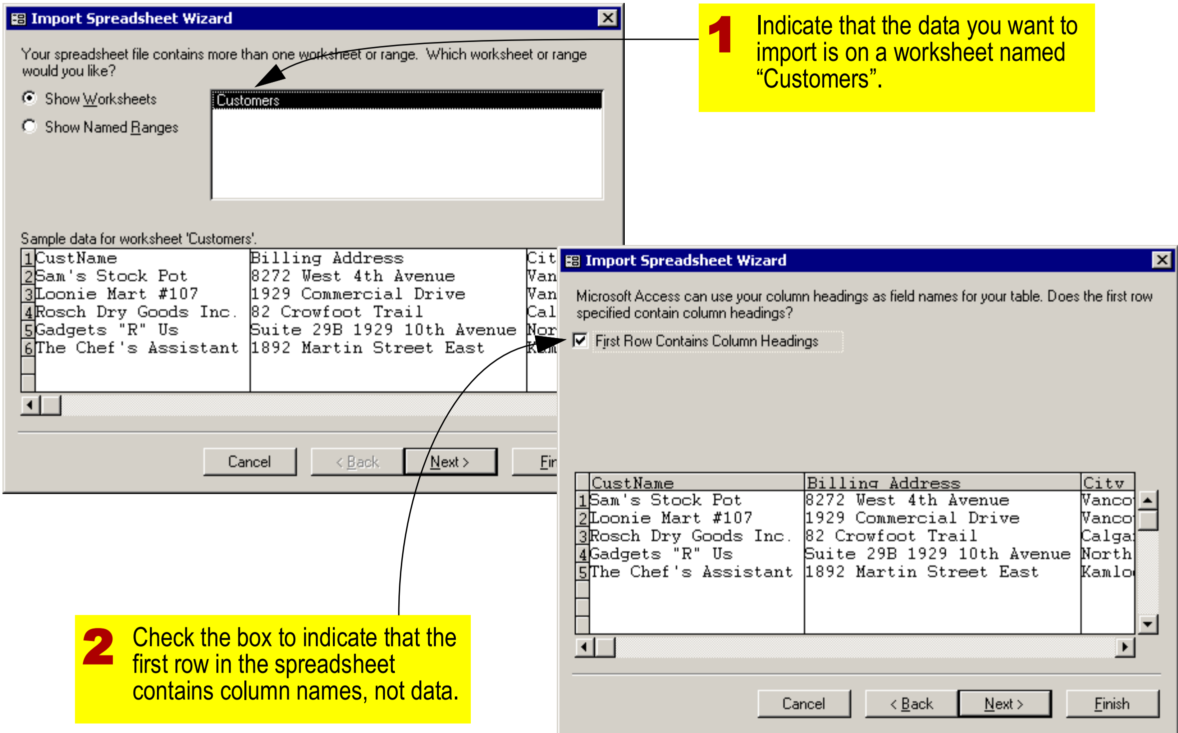


FIGURE 8.3: Use the import wizard to append records to the ***Customers*** table (part 1).



8.3.2 Importing data from a text file

In this section, we are going to assume that you have been provided with an ASCII text file called `Inventor.txt` that contains information about the inventory level of all your products. ASCII is an acronym for the American Standards Code for Information Exchange. Briefly, ASCII is a standard for converting patterns of bits and bytes into platform-independent, human-readable text. ASCII (or "plain text") files typically have extensions such as ".txt". ".asc", or ".csv".

If in doubt, an easy way to determine whether a file is in ASCII format is to open it using WINDOWS NOTEPAD. If you can read the contents of the file, then it is ASCII.[1] If you see a bunch of special characters, blocks, and smily faces, then the file is in one of many non-ASCII—or "binary"—formats (such as spreadsheet or a graphics file formats).

8.3.2.1 Exploring the ASCII text file

9 Open `Inventor.txt` in the NOTEPAD text editor that comes with WINDOWS. On most WINDOWS systems, you can start NOTEPAD using **Start** → **Programs** → **Accessories** → **Notepad**.

10 Examine the format of the data. The key elements of this particular text file are shown in Figure 8.5.

11 Close the text file.

Although the name of the data file is `Inventor.txt`, the file contains information about products generally. Inventory level (`QtyOnHand`) is simply one of several product attributes, such as description, price, and so on.

When you are building systems, do not be fooled by the naming conventions of others. For example, in a banking environments, information about clients (name, address, etc.) can often be found in a data source called "accounts". In many cases, some detective work is required to make sense of the data you already have.

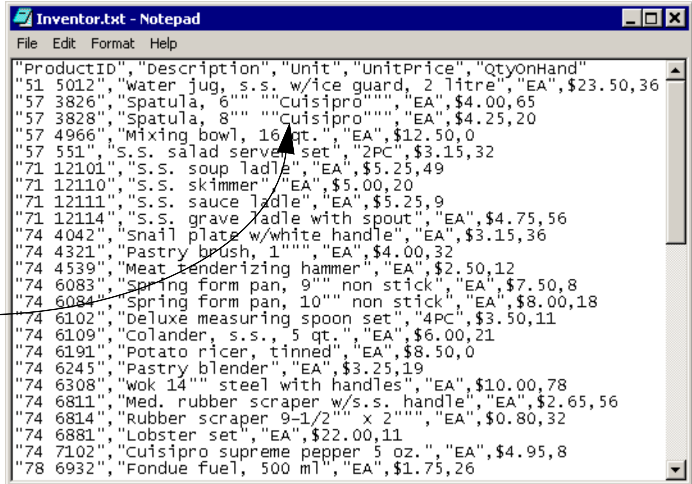
---

1 More recent versions of NOTEPAD can also read Unicode text. Unicode was developed in response to the inadequacy of ACSII for representing languages other than English. Whereas ASCII uses seven bits to represent up to 128 different characters, Unicode currently uses 16 bits and supports 24 different languages.

FIGURE 8.5: Examine the format of the ***Inventor.txt*** ASCII file.

An ASCII text file contains plain text. In the **`Inventor.txt`** file, each record is on a separate line and fields are delimiters by commas. Textual values are enclosed within quotation marks.

Although nested quotation marks often cause import programs to work incorrectly, the ACCESS import wizard appears to handle such issues very well. If the nested quotation marks do cause problems, you have to edit the text file (using search and replace) and re-run the import wizard.



8.3.2.2 Importing a new table

Rather than append the data to an existing table as you did in Section 8.3.1, you are going to import the text file into a completely new table.

Show me (lesson8-2.avi)

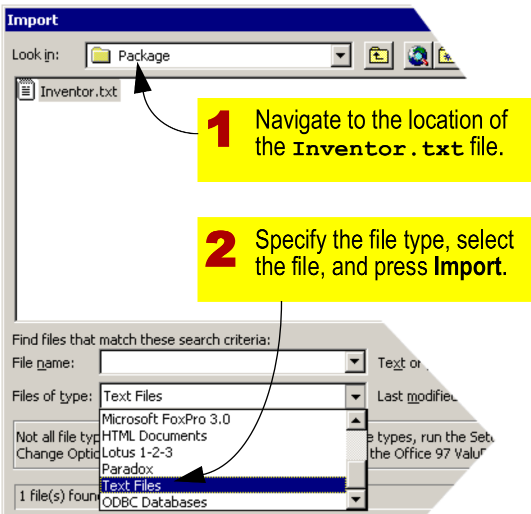
12 Ensure the database window is in the foreground and select **File** → **Get External Data** → **Import** from the main menu.

13 In the “Open” dialog box, specify the location of the **`Inventor.txt`** file, as shown in Figure 8.6.

14 Follow the instructions provided by the import spreadsheet wizard as shown in Figure 8.7 and Figure 8.8. Save the resulting table as new table named **`Products`**.

15 Open the **`Products`** table to verify that the data was imported correctly.

FIGURE 8.6: Locate and open the text file containing the product data.



It might seem like less work to import a new table rather than append records to an existing table. To append, you have to first create the table structure and then edit the source file to ensure that its headings conform to the table structure.

It turns out, however, that both approaches require about the same amount of effort. Once you have imported a table using the import wizard, you need to open the table in design mode and change many of the design decisions made by the import wizard.

For example, the import wizard sets the length of all text fields to the maximum allowable value (255 characters). In addition, the field names are taken directly from the heading row in the text file and no captions or defaults are specified.

Once you have created a table by importing data, you must remember to switch to design mode and change the field properties (e.g., data type, size, caption, and default) to appropriate values.

8.3.3 Creating a link to a different database

Recall that your employee data is stored in an off-the-shelf payroll system that you bought many years ago. Despite its age, the payroll system still does what it is supposed to do and you have no intention of replacing it or upgrading it. After asking around a bit, you discover that the data is actually saved in dBASE IV format for DOS.

Although you could use the import wizard to *import* the employee data, doing so would create an independent copy of the employee data. Thus, every change made to employee

FIGURE 8.7: Use the import wizard to create a ***Products*** table (part 1).

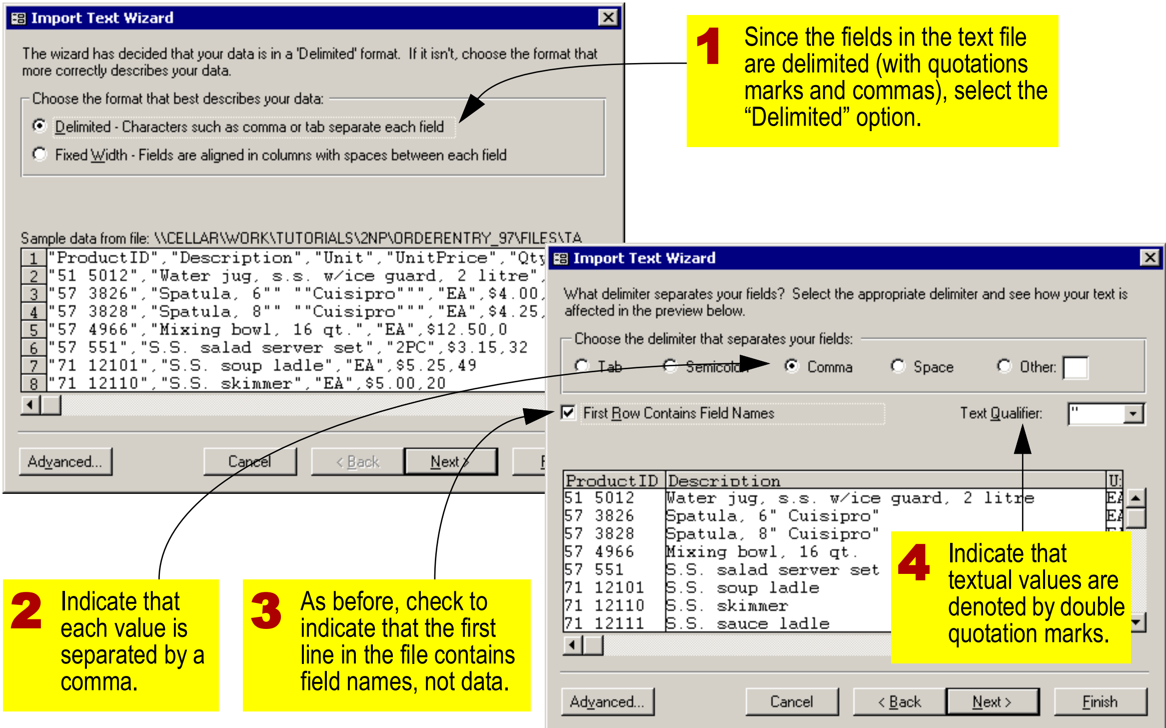
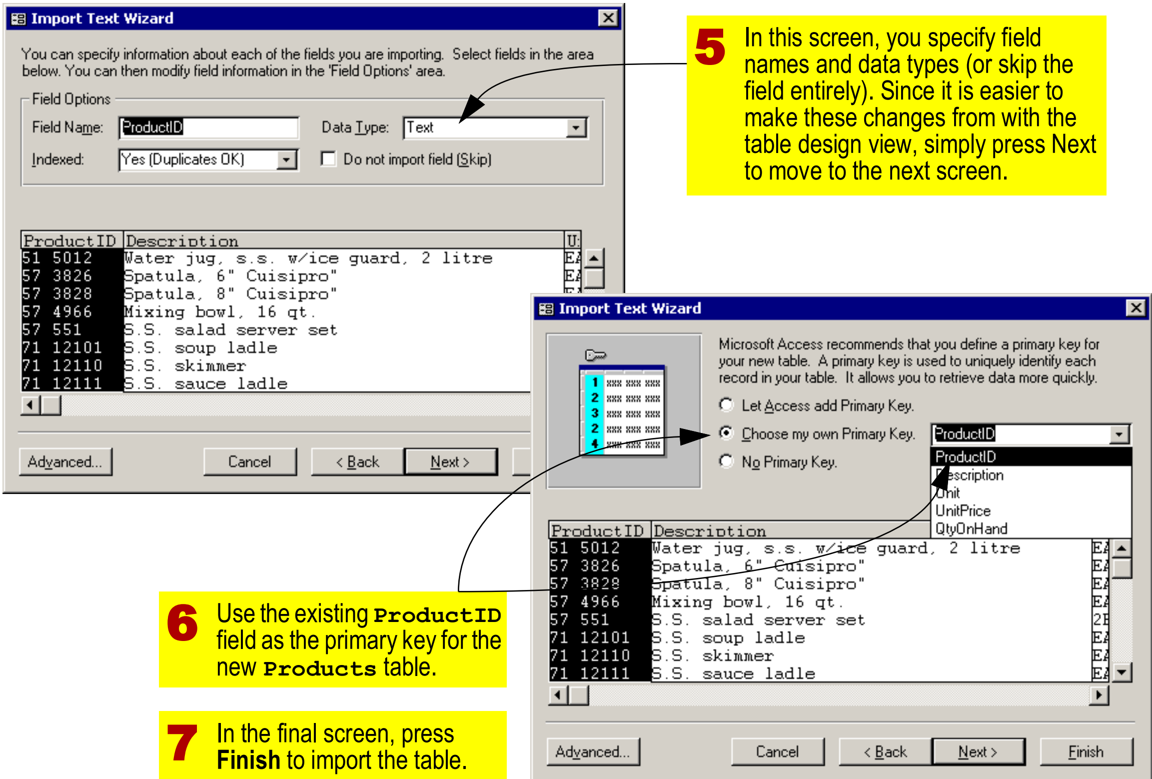


FIGURE 8.8: Use the import wizard to create a ***Products*** table (part 2).



information in the payroll system would also have to be made to the `Employees` table in the order entry system.

Murphy's Law of Information Systems states that if two copies of the same data are stored electronically, they will become inconsistent almost immediately.[1]

ACCESS has a feature called **linked tables** that permits you to use a table from different databases as if it were part of your own database. Moreover, the source table does not necessarily have to be located in an ACCESS database file. ACCESS provides direct support for many popular desktop databases (such as dBASE, FOXPRO, and PARADOX) and virtually any data source can be linked through ODBC middleware (ODBC is described in much greater detail in Lesson 9).

In this section, you are going to create a link to the table used to store employee data in the payroll system. In dBASE IV, each database object (table, query, form, and even index) is stored in a separate file. Thus, the `PAY_EMPS.dbf` file contains a single table. The index file for the table, which is used to identify the primary key and speed-up searches, is named `PAY_EMPS.mdx`. Both these files can be found in the project package.

It does not take long working with separate files for tables, indexes, and other database objects before you come to appreciate the single-file approach used by ACCESS.

Show me (lesson8-3.avi)

16 From the database window, select **File** → **Get External Data** from the main menu. Instead of selecting **Import** as you have done previously, select **Link Tables**.

17 In the "Link" dialog, find the `PAY_EMPS.dbf` file included in the project package. The file type should be set to dBASE IV.

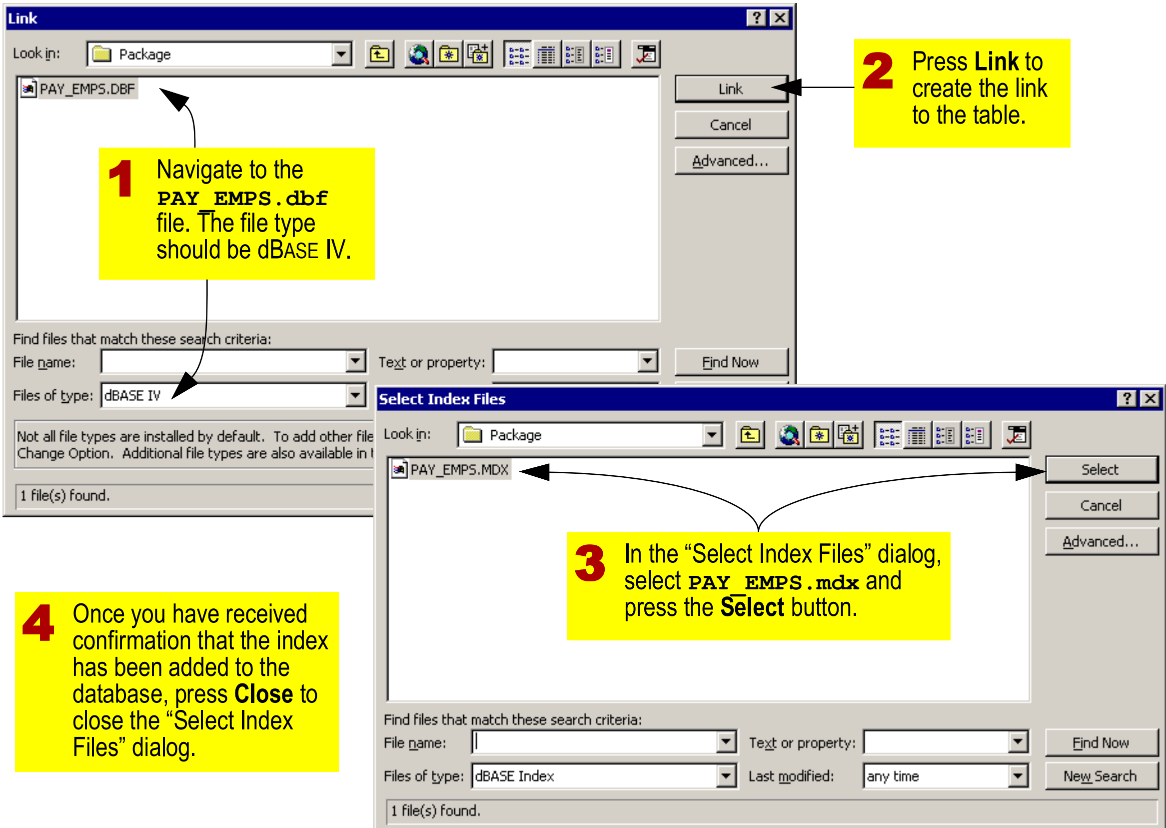
18 Select the file and press the **Link** button, as shown on the left-hand side of Figure 8.9.

If you look carefully, the name of the "Link" dialog has changed to "Select Index File" and the file type is "dBASE Index".

2K If you are using ACCESS 2000, you will not get the "Select Index File" dialog. In addition, ACCESS 2000 does not permit you

[1] This is not a *real* Murphy's Law—I just made it up.

FIGURE 8.9: Create a link to the dBASE IV file ***PAY_EMPS.dbf*** and its index file ***PAY_EMPS.mdx***.



to modify the table once the link to the dBASE table is created. The problem is that MICROSOFT has changed the way in which ACCESS 2000 interacts with dBASE files. To fix the problem, you can either visit the Microsoft site and download an updated copy of the JET 3.5 database engine (start by searching the MICROSOFT site for knowledge base article “Q263561”). Alternatively, you can simply ignore the problem—we will not be updating the **`PAY_EMPS`** table anyway.

19 Ensure **`PAY_EMPS.mdx`** is highlighted and press the Select button, as shown on the right-hand side of Figure 8.9. You should get a message indicating that the index has been added.

Once you have selected the index, the “Select Index File” dialog does not go away. This is because it is possible to have multiple indexes per table in ACCESS.

20 Press the Close button without adding any further index files.

You should now get a dialog asking you to select a unique record identifier as shown in Figure 8.10. Since only one dBASE index was specified, and since the primary key specified in the index is the field **`EMP_ID`**, you do not have much of a choice to make.

21 Select **OK** to **`EMP_ID`** as the unique record identifier (key). You should get a message that the link was made successfully.

22 Press **Close** to close the “Link” dialog box.

You should now have a linked table called **`PAY_EMPS`** in the database window, as shown in Figure 8.11. Since the name “PAY_EMPS” does not conform to our table naming convention, you should change it to something else.

23 Right-click on the **`PAY_EMPS`** linked table in the database window, select **Rename** from the context menu, and change the name of the link to **`Employees`**.

FIGURE 8.10: Select EMP_ID as the unique record identifier for the linked table.

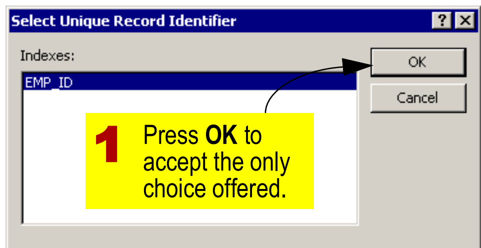
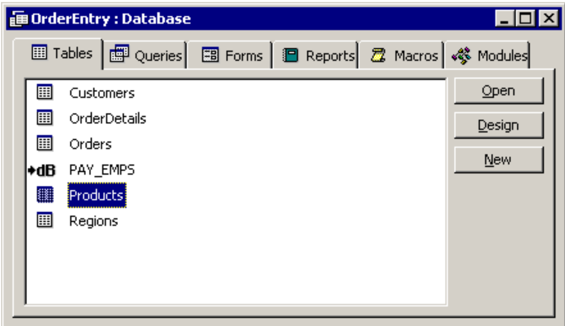


FIGURE 8.11: The database window shows a linked table named PAY_EMPS.



24 Open the **`Employees`** table in datasheet mode.

Note that the linked table is simply a window into the **`PAY_EMPS.dbf`** file. Thus, if you make any changes to the data (e.g., change Gerard Huff's first name to Gerry), the change is made directly to the payroll system's data. Of course, there is no payroll system in this case, so there is very little stopping you from experimenting with the linked data.

In the context in which the employee data is used here, it is actually very important that the data from the source table be read-only and sensitive data should be excluded. The last thing you want is for an order entry clerk to accidentally change job classifications or pay levels in the payroll system.

You must be mindful of security issues when creating links between databases. In this example, anyone with access to the order entry system can now open the **`Employees`** linked table and look at and change the **`EMP_PAY_LE`** (pay levels) field for all the employees in the company.

8.3.4 Changing the foreign key

In Section 6.3.3, we made a provisional commitment to use a concatenated key consisting of **`emp_fname`** + **`emp_lname`** to uniquely identify employees.

Now that we have created a link to the payroll system and have had the opportunity to see the fields and data types used in the system, we can make a more informed decision about what field to use as a foreign key. Specifically, we know that a field called **`EMP_ID`** exists and can be used to uniquely identify employees in the organization.

Remember that ACCESS, like most databases, ignores the case of field names and other database objects. Thus, it does not matter that you named your fields using lower case whereas the fields in the dBASE IV file are named using upper case.

8.3.4.1 Concatenated foreign keys revisited

Before we change the foreign key, we are going to take the opportunity to revisit the topic of declaring relationships for concatenated keys.

Show me (lesson8-4.avi)

25 Ensure that all tables are closed. Then open the relationships window.

26 Use the **Show Table** button add the new Employees table to the relationship window.

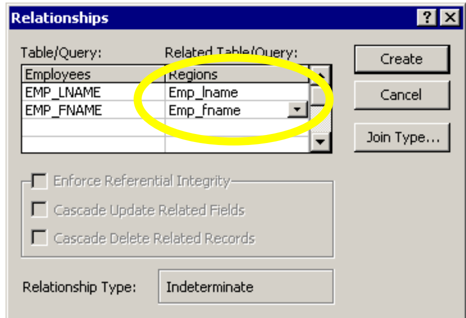
27 Multi-select the foreign key on the "one" side (that is, hold down the **Control** key and click on `EMP_FNAME` and `EMP_LNAME` in the `Employees` table).

28 Drag the fields on to the "many" side (the `Regions` table).

29 In the "Edit Relationships" dialog, make sure the correct fields in both tables are selected, as shown in Figure 8.12.

Although ACCESS selects the correct fields when a single-field foreign key is used, it

FIGURE 8.12: Ensure the correct fields are selected for the relationship.



does not do a very good job of selecting fields when multiple fields are used. As a consequence, you must complete the step shown in Figure 8.12 manually.

You will notice that the check boxes for specifying referential integrity and cascading deletes are disabled.

Since ACCESS is unable to automatically enforce referential integrity for linked tables, you have to write your own routines in a programming language or use other techniques (discussed in later lessons) to achieve this functionality.

8.3.4.2 Using EMP_ID as a foreign key

Changing foreign keys in a fully-implemented database application is much like changing the plumbing in an old house—it is hard, messy work and if all goes well, no one can tell the difference. On the other hand, if things do not go well…

30 Right click the relationship you created in Section 8.3.4.1 and select **Delete**.

Since `EMP_FNAME` and `EMP_LNAME` are no longer going to be used as the foreign key, they no longer belong in the `Regions` table.

31 Switch to the database window and open the `Regions` table in design mode.

32 Select `EMP_FNAME` and `EMP_LNAME` (use the field selectors to the left of the field names) and press delete. Ignore any warning messages that appear about deleting indexes.

33 Create a new field called `RepID` and set its data type to Long Integer. The field will contain the `EMP_ID` of the employee representing the region.

The pros and cons of "RepID" as a field name are discussed in Section 8.4.

34 Open the relationships window and create a relationship between `Employees.EMP_ID` and `Regions.RepID`.

8.4 Discussion: Naming consistency across multiple systems

In the exercises above, you created a field called `RepID` to store the employee number of the individual responsible for each sales region. In the human resources database, however, employee numbers are stored in a field called `EMP_ID`.

You can create a relationship between the `Regions.RepID` and `Employees.EMP_ID` fields as long as the two have exactly the same data type (in this case, Long Integer). That is, fields in a relationship do not need to have the same name. The downside of using different field names is that it is not obvious that the two fields contain the same data or are related in any way. On the positive side, the name `RepID` makes it clear that the field contains the ID of the sales representative for the region.

As a general rule of thumb, it is best to use the same name for the same piece of data. But in large organizations (or even in small organizations in which application development is decentralized), it is very hard to get people to converge on a single naming scheme without making an large invest in an organization-wide data model

Many CASE tools have a repository for attribute alias. For example, the fact that **RepID** is an alias for **EMP_ID** could be stored in the CASE tool for future reference.

8.5 Application to the project

Although you have imported customer data, recall that the spreadsheet does not contain any information about the regions to which customers have been assigned.

35 Add the following information to the **Customers** table:

Customer name	Region code
Sam's Stock Pot	C
Loonie Mart #107	K
Rosch Dry Goods Inc.	E
Gadgets “R” Us	C
The Chef's Assistant	N

The situation is similar for the **Regions** table: although you have created the **RepID** field to store the **EMP_ID** values of each region’s sales rep, you have not yet populated the **Regions.RepID** column.

36 Add the following information to the **Regions** table:

Region	EMP_ID of sales rep
Central	9
East	3
Key	9
North	4
South	NULL
West	10

Note that the there is currently no sales rep assigned to the South region. To record this fact in the **Regions** table, simply leave **RepID** blank for the South region (i.e., do not try to type N-U-L-L into the field).

37 Ensure you have modified the field properties of the tables you have imported. For example, the names, data types, and lengths of the fields in the **Products** table need to be changed before relationships are created.

An input mask is not recommended for **ProductID** for two reasons. First, the field is only partially structured so there is little you can do to constrain values. Second, as you will see later on, it is better to have users pick valid `ProductIDs` values from a list.

If you ignore the recommendation above and decide to create an input mask for the `ProductID` field, ensure you understand the implications of Section 5.4.6.2.

At this point, all your **master tables** should be populated. Do not worry about populating your **transaction tables** yet—this is what we are building the order entry system for.

For the purpose of the assignment, the term “transaction” tables refers to tables that contain information about individual transactions (e.g., `Orders`, `OrderDetails`). “Master” tables, in contrast, are tables that contain relatively stable, non-transactional information (e.g., `Customers`, `Products`).

Lesson 9: Client/server and ODBC

9.1 Introduction:

The great thing about ACCESS is that it provides powerful and easy-to-use tools for organizing data, creating forms, producing reports, and automating tasks. These desktop tools are a significant improvement over the arcane command-line interfaces that ship with many industrial-strength database systems.

The not-so-great thing about ACCESS is that the database engine itself it is not designed to support high transaction volumes or a large number of simultaneous users. For example, one simply could not run an airline reservation system on top of an ACCESS database.

Fortunately, ACCESS can play the role of the "client" when connecting to an industrial-strength client/server database. The implication is that you can continue to work within ACCESS without having to store multiple copies of your organization's data in multiple independent desktop database systems. Instead, many desktop systems can link to a single source of data that is stored and administered centrally.

You had a glimpse of how this might work in Section 8.3.3 when you created a link to a dBASE IV file containing payroll data. However, dBASE IV is definitely not a client/server database and the file we linked to was on the same machine as the ACCESS database. Client/server databases are designed to be used by multiple simultaneous users over network connections.

In order to use ACCESS to connect to a database server over a network, you need something called middleware. In this demonstration, we are going to use Open Database Connectivity (ODBC) middleware to create a link to data on a different computer using a different operating system and running a different DBMS.

9.1.1 Doing versus demonstration

What I used to do with my students is provide them with the Internet Protocol (IP) address of a SQL SERVER database server, give them a valid user name and password for the server, and make them create an ODBC connection to the SQL SERVER database over the Internet. Although this was certainly a cool exercise when it worked, the more typical outcome was conflicting ODBC driver versions, flaky networks, and frustration. And all these problems occurred in (what was supposed to be) a standardized computing lab environment.

Client/server computing relies on reliable high-speed network connections between the clients and the server. Without robust networking infrastructure, and compatible middleware, client/server is more trouble than it is worth.

My new approach is to simply demonstrate the process of setting up an ODBC connection to a client/server database. Although a passive demonstration is no substitute for hands-on experience, it is simply not practical to let everyone who works through these tutorials access my database server.

Of course, it is possible to set up your own client/server database in order to work through this lesson.[1] Just be warned that setting up a client/server database server is tricky and is probably not worth the effort and angst at this early stage.

[1] ACCESS 2000 includes a client/server database—the MICROSOFT DATA ENGINE (MSDE)—on the installation CD-ROM. Although MSDE is a scaled down version of MICROSOFT's flagship database product, SQL SERVER, ACCESS 2000 continues to use its own JET database engine by default.

You are not expected to perform the exercises in this lesson. Instead, you should just sit on your hands and follow along.

9.1.2 The client-server environment

To set the stage for the demonstration, let us assume the following: Instead of storing your employee data in a payroll system built on top of dBASE IV files, you purchased the human resources module from an enterprise resources planning (ERP) vendor like ORACLE, PEOPLESOFT or SAP.

At their core, ERP systems are simply standardized applications that run on top of large, integrated relational database systems.

In this scenario, we are assuming that you want to by-pass the ERP application and read its underlying database directly. In this way, you can use up-to-date employee information in the order entry application that you are building.

9.1.2.1 Database server details

Assume that the relational DBMS being used by the ERP system is INTERBASE. Since INTERBASE is open-source software, it can be freely downloaded.

Typically, companies that spend tens of millions of dollars on ERP installations do not decide to run the whole thing on top of an open-source database they downloaded from the Internet (although this may be changing). I am using INTERBASE as an example because—who knows—you may want to download a copy and try the exercises in this lesson at some point in the future when you have a whole weekend to waste.

Before we connect to the database server, there are a handful of technical details we should know about the server itself:

- The database server is running INTERBASE version 6.0 on top of the LINUX operating system (downloaded from REDHAT).[1]
- Assume that the IP address of the database server is `misux.bus.sfu.ca`.
- A view called `SALES_REPS` has been created to show only those employees classified as sales people and to hide confidential information about pay levels.

Views and queries will be discussed in greater detail in Lesson 10. Briefly, a view is like a “virtual table” that is used to filter and organize data from one or more database tables. For example, a view could provide the data in the `Employees` table sorted by last name. A second view could show the same data sorted by employee number. The important thing is that neither view changes the ordering or the structure of the data in the `Employees` table.

- An ODBC driver to communicate with the database server has been purchased and installed on the client machine. In this example, the INTERBASE ODBC driver has been provided by third-party company called EASYSOFT.

Although the high-level language used for creating tables and queries is standardized across all major DBMS vendors, the low-level languages and interfaces required for client/server communication are far from standardized. Consequently, a special ODBC driver is required to translate standard high-level commands into the

---

1 Rather than being an industrial-strength ERP platform, this machine is a crummy old Pentium 120 that I use for development purposes. Since it is an unwanted machine (without a monitor or a mouse) running an open-source DBMS on top of an open-source operating system, the total cost to build the database server was negligible.

idiosyncratic low-level commands required by each DBMS.

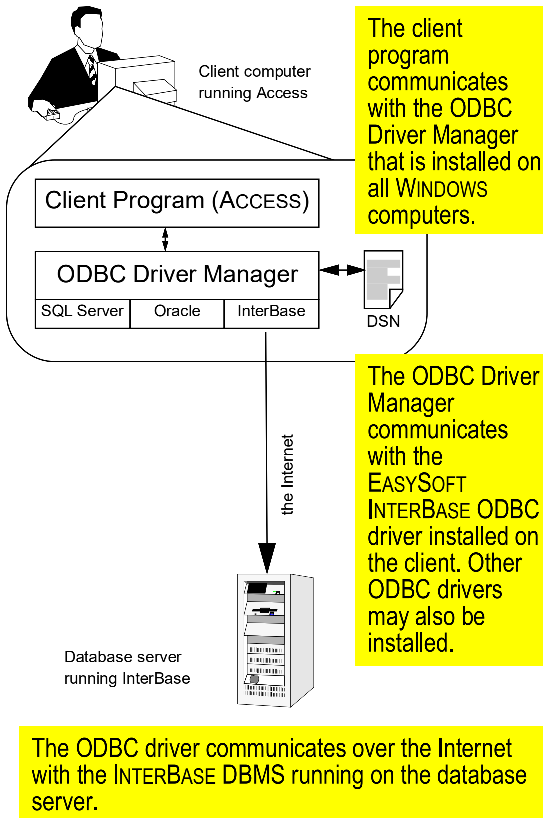
9.1.2.2 Setting up an ODBC connection

To get data from the database server to the client machine and into a program like MICROSOFT ACCESS, some infrastructure has to be in place:

1. The client computer has to know how to find the database server on the network.
2. The server needs some means of authenticating the client application as a user. Not just anyone should be able to log into the server and download payroll information.
3. Some form of network transport has to be provided so that my client computer can send the server instructions and the server can send back data.
4. The data received from the server must be reassembled into rows and columns.

Fortunately, all of this infrastructure is provided by the ODBC middleware. The ODBC standard is discussed in greater detail in Section 9.4.1. At this point, you can think ODBC as an adapter that permits different systems from different vendors to be linked together, as shown in Figure 9.1.

FIGURE 9.1: The role of ODBC software in a client/server connection.



9.2 Learning objectives

- see how an ODBC data source is configured in WINDOWS.
- understand how ACCESS can be used as a front-end to a client/server database
- see how easy it is to change the data source when using the ODBC infrastructure
- see how other WINDOWS applications can use an ODBC connection
- gain a basic understanding of what ODBC middleware is and how it is used

9.3 Exercises

9.3.1 Linking to a client/server database

To connect the database server, I must first define a “data source name” (DSN) on the client machine. The DSN simply permits me to specify and save important details concerning the connection, such as the server name, the user name that should be used to log into the server, and so on.

Show me (lesson9-1.avi)

1 First, I invoke the ODBC Data Source Administrator Tool from the WINDOWS Control Panel.

2 I select the System DSN tab and press the **Add** button, as shown on the left-hand side of Figure 9.2.

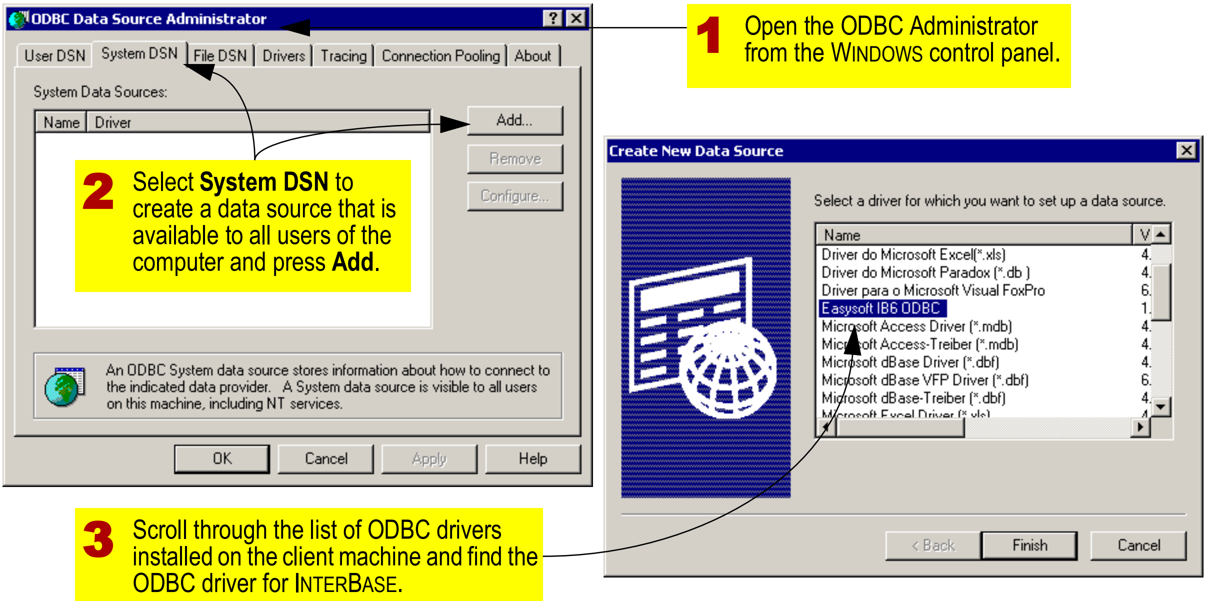
If you are wondering what all the tabs are for in “ODBC Data Source Administrator” window in Figure 9.2, there are two different ways to save the DSN information for a particular database connection: in the WINDOWS registry (“Machine DSN”) and to a small text file (“File DSN”). Machine DSNs can be further subdivided into those that can be seen and used by all users of a system (“System DSN”) and those that are specific to a particular user (“User DSN”). The basic function of the DSN is the same in all cases, however.

9.3.1.1 Specifying the data source details

3 I select a driver that is designed to communicate with the DBMS running on the server. In this case, I select the EASYSOFT INTERBASE version 6.0 driver, as shown on the right-hand side of Figure 9.2.

From a pricing point of view, there are three types of ODBC drivers: free, expensive, and insanely expensive. ACCESS automatically installs free ODBC drivers for a number of database systems,

FIGURE 9.2: Create a new data source name (DSN) for the database server.



including MICROSOFT SQL SERVER and ORACLE (although you need a client license from ORACLE to use the ODBC driver). Prices for third-party drivers can run into the hundreds of thousands of dollars (I am not kidding). The EASYSOFT driver retails for about US$100 and several open-source drivers for INTERBASE are under development.

4 I click the **Finish** button to exit the ODBC Data Source Administrator Tool.

Next, the configuration tool for the EASYSOFT driver takes over from the ODBC Data Source Administrator. The EASYSOFT dialog collects additional information that is specific to INTERBASE database servers.

5 First, I enter a name for the DSN and a description. Then I type in the Internet address of the database server (`misux.bus.sfu.ca`) and the location of the target database on the server (`/home/brydon/IBData/Payroll.gdb`), as shown in Figure 9.3.

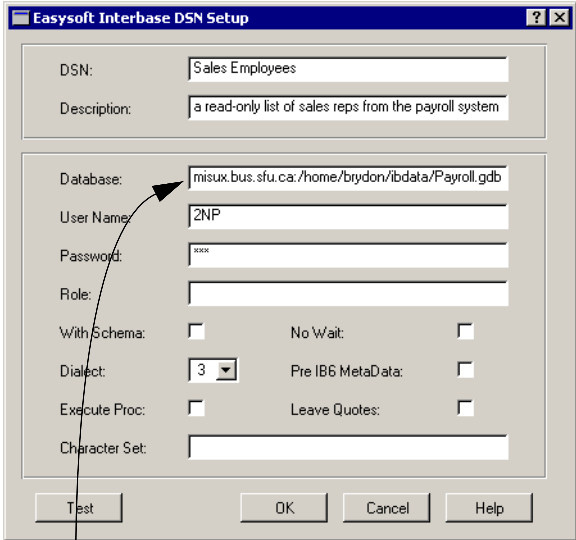
6 Then, I enter a valid user name and password.

Like all client/server databases, INTERBASE has a robust security infrastructure. The “2NP” account used in this example has been granted read-only access to the `SALES_REPS` view, but no access to other, more sensitive information such as that contained in the `PAY_EMPS` table.

7 Finally, I press the **Test** button to make sure I have entered the DSN information correctly. The good news is shown in Figure 9.4.

Since the test was successful, I have a working ODBC connection to a remote database.

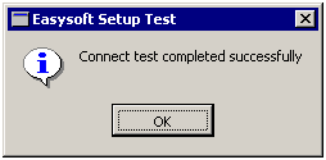
FIGURE 9.3: Enter the InterBase-specific information required to make a connection to the server.



1 Enter the location of the database (including the IP address of the server and the location of the database file on the server).

The nice thing about using the Internet protocol (TCP/IP) to identify the location of the server is that the server can be

FIGURE 9.4: Test the ODBC connection to ensure it works.



anywhere in the world. As long as I have the server's IP address (in this case, **`misux.bus.sfu.ca`** or **`142.58.223.133`**), I can access the data.

9.3.1.2 Using the ODBC connection to create a linked table

Now that an ODBC link is in place, it can be used by any program that supports the ODBC standard, such all MICROSOFT OFFICE applications, high-end statistics packages, even the shipping software provided by many courier companies. In my case, I want to use the ODBC connection to create a linked table in ACCESS.

Show me (lesson9-2.avi)

8 I return to ACCESS and initiate the linked table dialog as you did in Section 8.3.3.

9 Instead of selecting a specific file type in the "Link" dialog, I select "ODBC Database".

When I select an OBDC data source, a list of both "machine" and "file" data sources pops up.

10 I select the **Machine Data Source** tab and find the DSN of the payroll system connection I set up earlier, as shown in Figure 9.5.

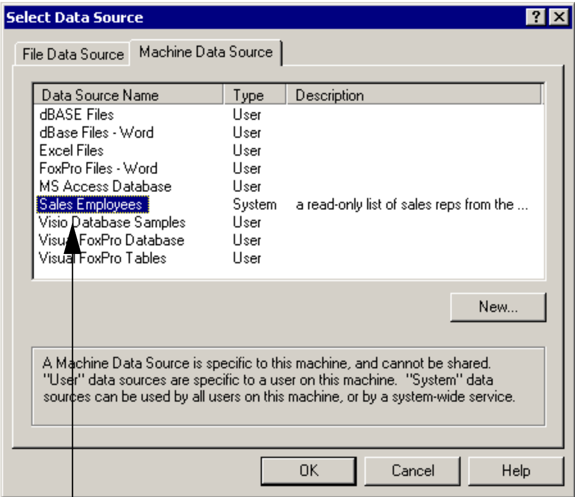
If everything goes well, ACCESS will use the ODBC link to request a list of tables and views in the payroll database. As Figure 9.6 shows, some of the tables are system tables (prefixed by **`RDB$`**). System tables contain data used by the DBMS itself and are of little interest to us.

11 I select the **`SALES_REP`** view, as shown in Figure 9.6.

Unlike tables, views do not have a primary key defined. Thus, I am asked to select one field (e.g., **`EMP_ID`**) as the "unique record identifier" for the purpose of the linked table, as shown in Figure 9.7.

Note that the icon for the ODBC linked table is different than that of the other tables in the ACCESS database.

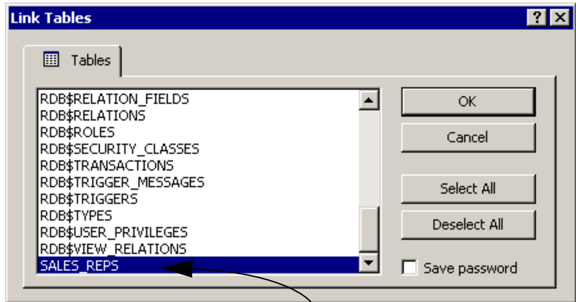
FIGURE 9.5: Use the ODBC connection as the data source for the linked table.



1 Select **Machine Data Source** and find the payroll system I set up in the previous section.

FIGURE 9.6: Create links to one or more tables in the remote database.

The ODBC driver returns all the tables in the database, including the system tables.



1 Select the **`SALES_REP`** table (which is actually a view) and press **OK**.

12 To demonstrate the security features of the linked table, I open it in data sheet mode and make a change to one of the records. As Figure 9.8 shows, the "2NP" account is permitted to view, but not change the data on the server.

If the database administer were to grant modify privileges to the "2NP" account, the ACCESS application could be used to add, modify, and delete data from the remote database. Although such functionality does not make sense in the context of payroll data, there are other situations in which an ACCESS front-end to

FIGURE 9.7: Select a field to be the unique record identifier for the ***SALES_REP*** linked table.

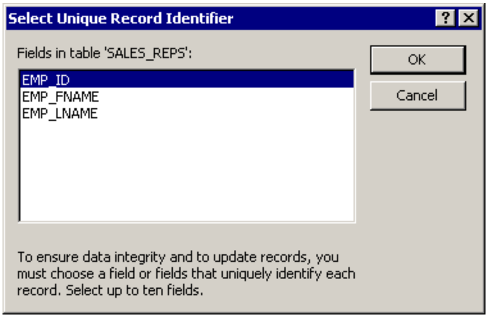
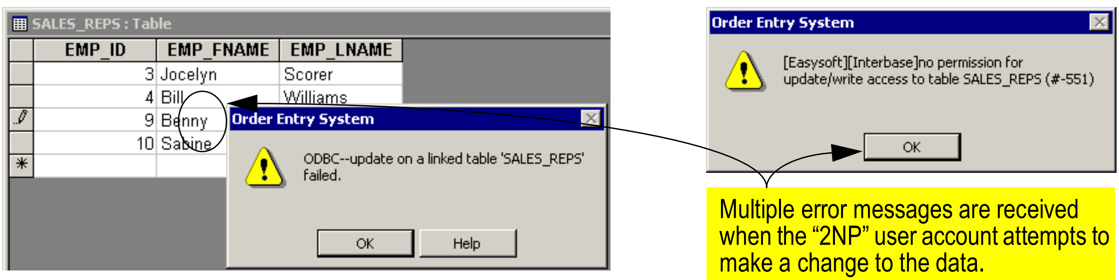


FIGURE 9.8: User “2NP” does not have update permission on the ***Employees*** table.



a client/server database is extremely convenient.

Accessing data over a network is bound to be slower than accessing a local copy. In some cases, network delays or unreliability make client/server access infeasible. In such situations, data **replication** and **synchronization** become very attractive technologies.

9.3.2 Changing the target database

To illustrate the power of middleware, assume that you have built additional ACCESS objects (such as queries, reports, and forms) on top of the `SALES_REPS` linked table. Assume as well that the people in charge of the ERP system decide to switch from an INTERBASE DBMS to MICROSOFT SQL SERVER on a different machine. In this section, I will demonstrate how easy it is incorporate this change into an ACCESS application.

Show me (lesson9-3.avi)

13 First, I create a new DSN. However, instead of using the INTERBASE driver, I use the SQL SERVER driver that ships with ACCESS.

14 I fill in the SQL SERVER-specific dialog boxes provided by the ODBC driver required to fully specify the DSN, as shown in Figure 9.9 and Figure 9.10.

As Figure 9.9 illustrates, each ODBC driver requires slightly different DSN information. For example, SQL SERVER supports multiple network protocols and therefore requires that the desired network protocol be specified in the DSN.

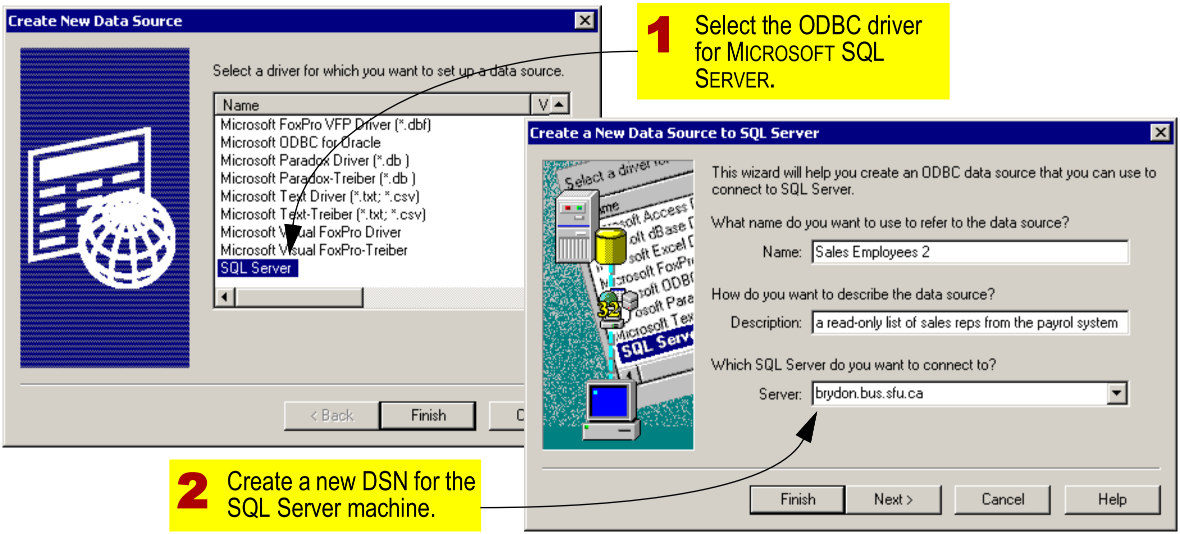
15 I delete the existing linked table (which, of course, has no effect whatsoever on the table stored on the database server) and create a new link using the SQL SERVER DSN, just as in Section 9.3.1.2.

At the end of the process—which is only superficially different from the INTERBASE example—I have a linked table with the same data as before. In most cases, no additional changes are required to my ACCESS application—that is, all my queries, forms, reports, and VISUAL BASIC programs work just as they did when I was running INTERBASE off a LINUX machine.

9.3.3 Changing the client application

Once an ODBC data source is set up on the client computer, it is possible for other applications to use the same connection. In this section, I will use an ODBC connection to bring employee information into MICROSOFT EXCEL. It is important to note that the techniques shown

FIGURE 9.9: Create a new DSN using the SQL SERVER ODBC driver (part 1)



below can be used to bring data into WORD (e.g., for a mail merge) or virtually any other WINDOWS program.

Show me (lesson9-4.avi)

16 I start by opening a new worksheet in EXCEL and selecting **Data → Get External Data → Create New Query** from EXCEL's main menu.

Rather than having their own functionality for handling ODBC connections, all OFFICE applications except ACCESS rely on a separate program called MICROSOFT QUERY to do the dirty work.

Like many of the helper and add-in programs in OFFICE, MICROSOFT QUERY may or may not have been installed when you installed OFFICE. If you get an error message similar to the one shown in

FIGURE 9.10: Create a new DSN using the SQL SERVER ODBC driver (part 2)

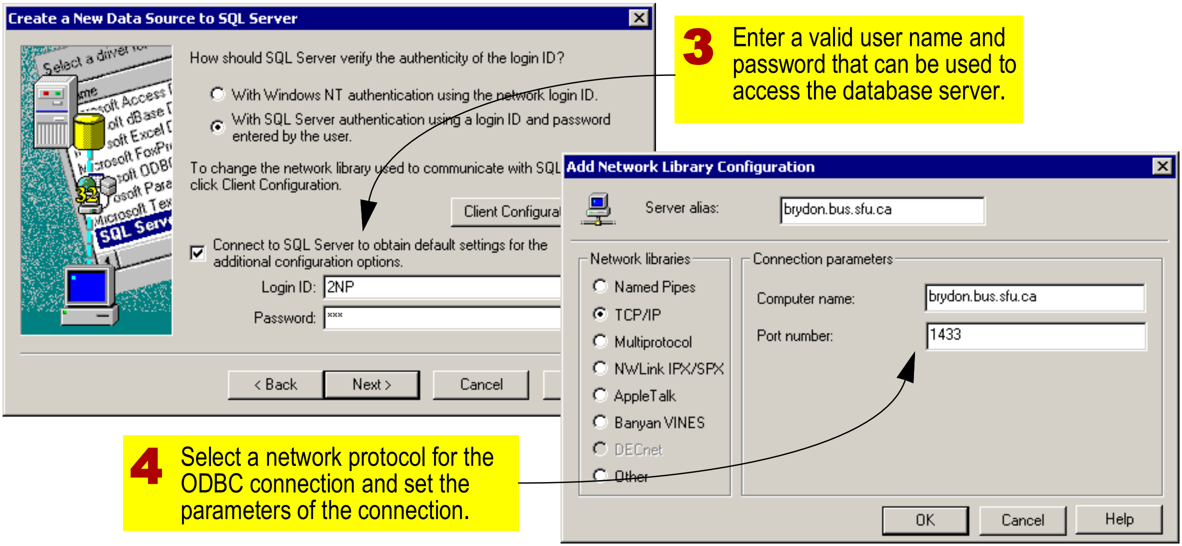


Figure 9.11, you must re-run the setup program from the OFFICE CD-ROM, as shown in Figure 9.12.

Assume that I am using EXCEL 97 and that I have created a “file DSN” that is identical in every way to the “system DSN” created in Section 9.3.1.1 (except that file DSNs are saved to a text file rather than to the WINDOWS registry).

17 From within MICROSOFT QUERY, I select the file DSN called `Sales Employees 3` from the list, as shown on the left-hand side of Figure 9.13.

18 I then expand a table and select the columns I want to include in my spreadsheet, as shown on the right-hand side of Figure 9.13.

FIGURE 9.11: MICROSOFT QUERY is not installed on the computer.

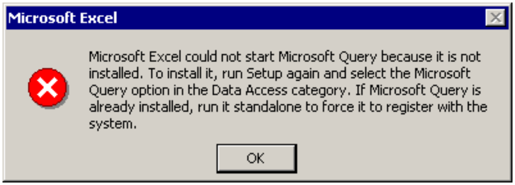


FIGURE 9.13: Use the INTERBASE ODBC connection to bring data into MICROSOFT QUERY.

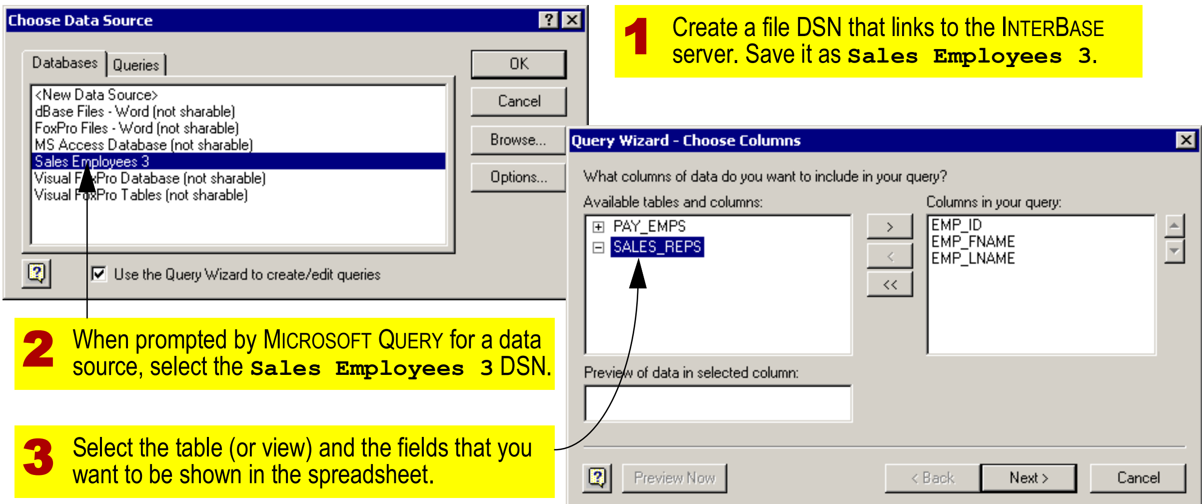
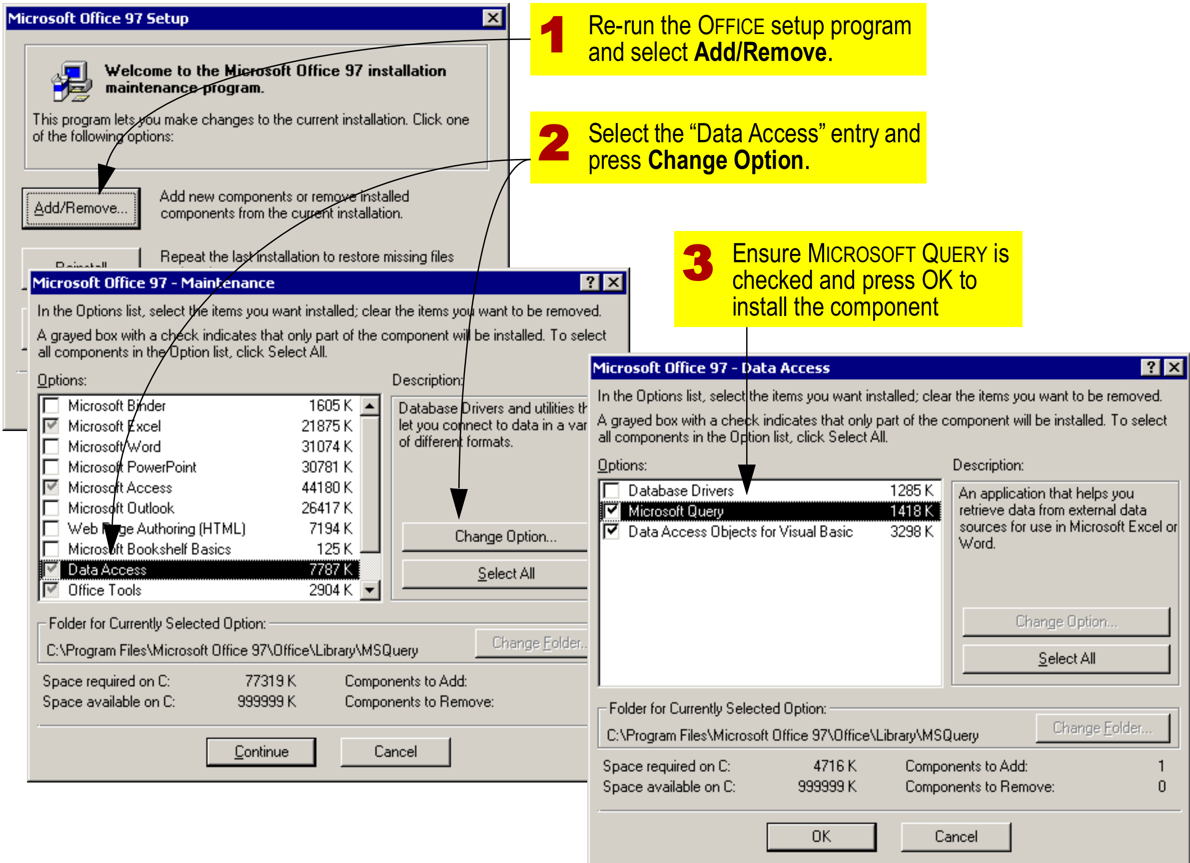


FIGURE 9.12: Re-run the OFFICE setup program to install MICROSOFT QUERY.



Unlike ACCESS, MICROSOFT QUERY hides the system tables (e.g., **RDB$CHECK_CONSTRAINTS**) by default.

MICROSOFT QUERY asks me if I want to filter and sort the data and where I want the data to be placed in the spreadsheet.

19 I select a location in the spreadsheet and the data magically appears.

20 To make sure I have the most up-to-date data from the database, I press the **Refresh Data** button on the “External Data” toolbar, as shown in Figure 9.14.

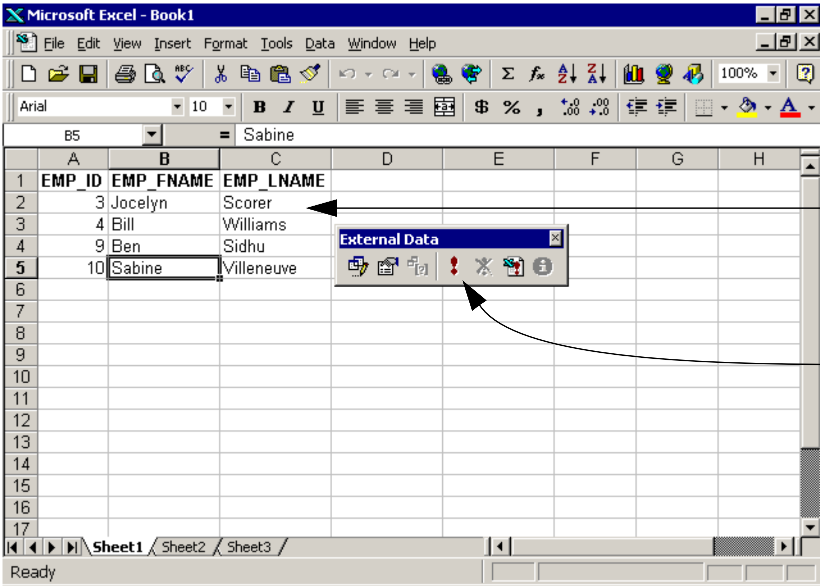


FIGURE 9.14: Data from the database server is copied into the spreadsheet.

1 Specify the location within the EXCEL worksheet for the data returned by MICROSOFT QUERY.

2 Press the **Refresh Data** button to re-query the source database.

Changes to the database are not automatically noticed by the ODBC connection. Hence the need to refresh.

The flow of information from the database to the spreadsheet is one-way. That is, if a change is made to the payroll system, the changes to the data to appear in the spreadsheet (once the **Refresh Data** button is pressed). However, if the data in the spreadsheet is changed, none of the changes are saved back to the database. Moreover, any changes to the data in the spreadsheet are overwritten the next time the data is refreshed.

With a surprisingly small amount of programming within EXCEL, it is possible to use the ODBC connection to update the client/server database from the spreadsheet. However, given the lack of structure and control in spreadsheets, it is not clear that using EXCEL as a two-way front-end to your data is a good idea. In the case of payroll data that is feeding an ERP system, anything other than read-only access is a very bad idea.

9.4 Discussion

9.4.1 ODBC and client/server databases

If you accept the premise that data is an organizational resource, then the notion of many users working with (and hoarding) their own personal copies of data on their desktops is bound to be troubling. And so it should be.

An elegant way around this centralization/decentralization dilemma is to use ACCESS as the front-end to tables stored on a centrally maintained, industrial strength, multi-user DBMS. In this way, users can continue to work with ACCESS as if the tables were stored locally. However, in reality, they are looking at and interacting with the same data as everyone else in the organization.

ODBC (open database connectivity) is a MICROSOFT-developed standard that provides a common interface to virtually all databases. The key to ODBC is drivers that translate basic database query commands into commands understood by the particular data source at the other end of the wire.

As long as you have the correct ODBC driver and can generate ODBC-compliant commands, you really do not have to know anything about the source of the data—it could be ORACLE, it could be MICROSOFT SQL SERVER, it could be one of hundreds of other types of data storages systems.

There is plenty of information on ODBC on the MICROSOFT web site. ORACLE, on the other hand, prefers to more or less deny the existence of ODBC. Not surprisingly, there are a number of third-party vendors of ODBC software that neatly bridge the ideological chasm between MICROSOFT and other database vendors such as ORACLE. For example, both MERANT (formerly INTERSOLV) and Vancouver's own SIMBA provide middleware products that take care of many of the frustrating client/server networking issues as well as provide ODBC → native database translation.

MICROSOFT is in the process of superseding ODBC with new standards (e.g., OLE DB and ADO). Check the MICROSOFT web site for the latest information on this ever-changing alphabet soup of data-access standards and middleware. You will gain some experience with ADO in Lesson 28.

9.4.2 The M2M Internet

Much of the discussion about the Internet to date has been around the use of browsers to surf the world wide web (WWW).[1] Business models based on the web—such as "business to consumer" (B2C)—rely on people communicating with machines over a network that is essentially free and ubiquitous (at least in some parts of the world).

In this lesson, you have seen a different use of the Internet: machine-to-machine (M2M) communication over the same TCP/IP networks that carry the traffic from million of web surfers. The potential of machine-to-machine communication over a wide-area network such as the Internet is huge: many things are possible including software-on-demand, distributed databases, distributed computation (e.g., SETI@Home), interactive gaming, and the list goes on and on.

[1] Indeed, "surfing the web" is so common that the term has ceased to be regarded as a mixed metaphor.

Lesson 10: Basic queries using QBE

10.1 Introduction: Using queries to get the information you need

At first glance, it appears that splitting information into multiple tables and relationships creates more of a headache than it is worth. People generally like to have all the information they require on one screen (like a spreadsheet, for instance); they do not want to have to know about multiple tables, foreign keys, relationships, and so on.

Saved queries address this problem. Queries allow the user to join data from one or more tables, order the data in different ways, calculate new fields, and specify criteria to filter out certain records. The important thing to keep in mind is that a query contains no data—it merely reorganizes the data from the table (or tables) on which it is built without changing the "underlying tables" in any way.

Once a query is defined and saved, it can be used in exactly the same way as a table. Because of this, it is useful to think of queries as "virtual tables". Indeed, in the majority of DBMSes, saved queries are called **views** because they allow different users and different applications to have different *views* of the same data.

10.2 Learning objectives

- create different types of queries
- understand how queries can be used to answer questions
- develop a naming convention for queries
- understand the difference between an "updatable" and "non-updatable" recordset

10.3 Exercises

10.3.1 Creating a query

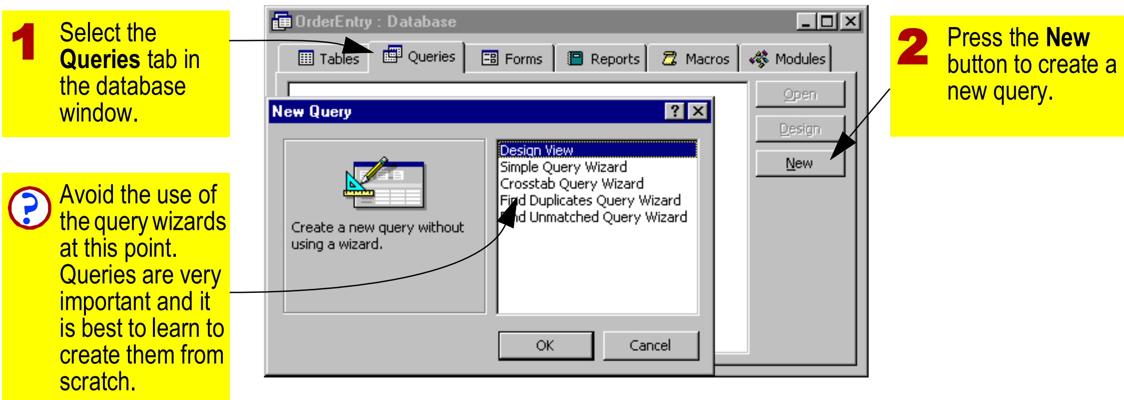
Show me (lesson10-1.avi)

1. Use the **New** button in the **Queries** pane of the database window to create a new query as shown in Figure 10.1.

2. Add the `Products` table to the query as shown in Figure 10.2.

3. Examine the basic elements of the query design screen as shown in Figure 10.3.

FIGURE 10.1: Create a new query.



4 Save your query (**Ctrl-S**) using the name `qryBasics`.

The queries you build in these exercises are for practice only. That is, they are not used in your order entry project and it is not absolutely critical that they be saved as part of your database. On the other hand, there is little reason *not* to save them.

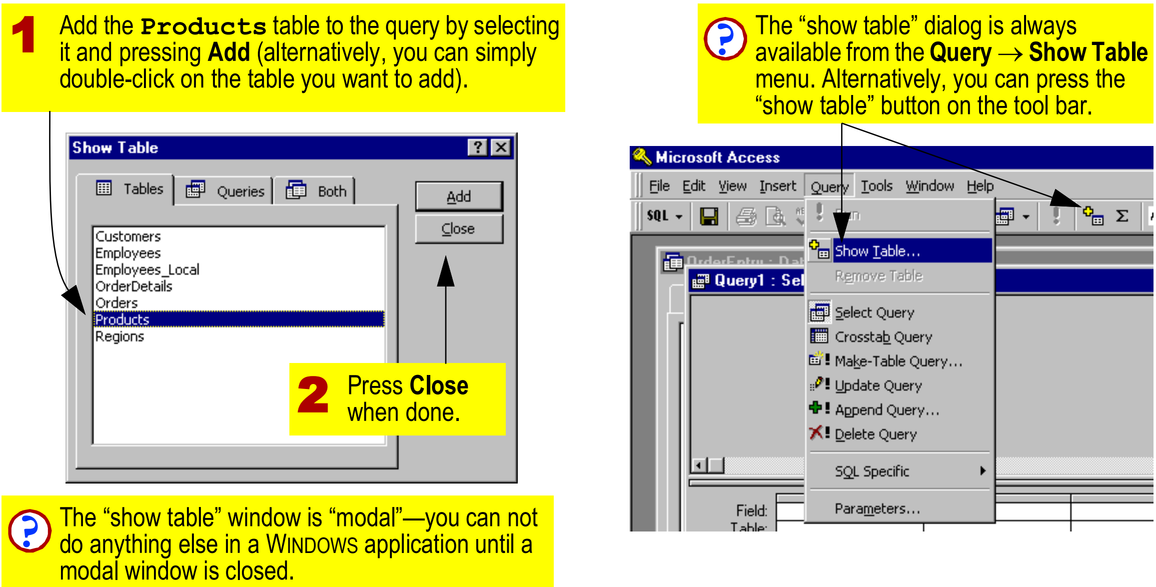
10.3.2 Five fundamental query operations

In the following sections, you are introduced to five fundamental query operations: projection, selection, sorting, joining, and calculated fields. The operations are *fundamental* in the sense that they are supported by all relational database systems.

10.3.2.1 Projection

Projecting a field into a query simply means including it in the query definition. The ability to base a query on a subset of the fields in an

FIGURE 10.2: Add tables to your query using the “show table” dialog.



underlying table (or tables) is particularly useful when dealing with tables that contain some information that is confidential and some that is not confidential.

For instance, the **PAY_EMPS** table you used in Section 8.3.3 contains a field describing the employee’s pay level. If you created a saved query that projected all the employee fields *except* **EMP_PAY_LE** and gave users permission to view the saved query instead of the **PAY_EMPS** table, then unauthorized users would have no way of viewing sensitive pay information.

Recall that a view called **SALES_REPS** was used in Lesson 9 to show only the employees names and ID numbers from the **PAY_EMPS** table.

Show me (lesson10-2.avi)

5 Perform the steps shown in Figure 10.4 to project the **ProductID**, **Description**, and **UnitPrice** fields into the query definition.

6 Select **View** → **Datasheet** from the menu to see the results of the query. Alternatively, press the datasheet icon ( ) on the tool bar.

7 Select **View** → **Query Design** to return to design mode. Alternatively, press the design icon ( ) on the tool bar.

10.3.2.2 Selection

You select records by specifying conditions that each record must satisfy in order to be included in the results set. To achieve this in "query-by-example", you enter *examples* of the results you desire into the criteria row.

Show me (lesson10-3.avi)

FIGURE 10.4: Project a subset of the available fields into the query definition.

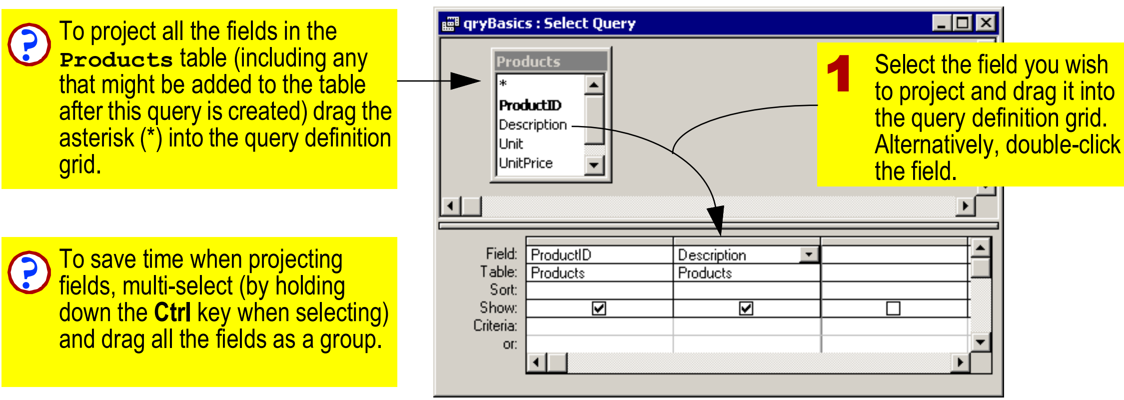
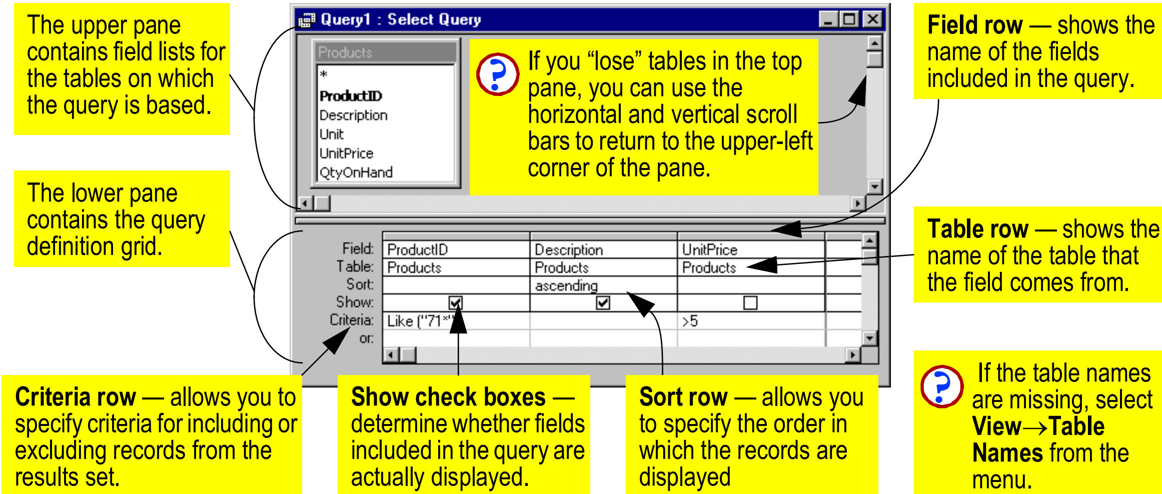


FIGURE 10.3: The basic elements of the query design screen.



8 Perform the steps shown in Figure 10.5 to answer the following question:
"Which products sell for more than $20?"

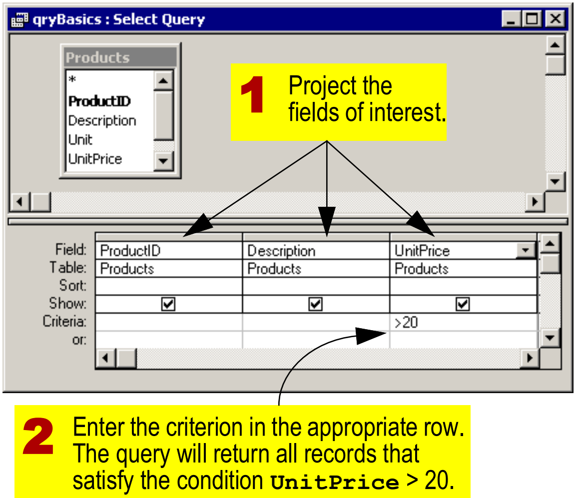
9 Select **View → Datasheet View** from the main menu to see the results of the query, as shown in Figure 10.6. Since this is a select query, the resulting **recordset** contains only those records that satisfy the criteria.

10.3.2.3 Complex selection criteria

It is also possible to create complex selection criteria using **Boolean** (logical) constructs such as AND, OR, and NOT.

10 Use complex selection criteria to answer more complex questions:

FIGURE 10.5: Select records using a "greater than" criterion in the *UnitPrice* field.



- *"Which products cost less than $5 each?"* (see Figure 10.7)
- *"Which products cost less than $2 each or cost less than $5 for unit sizes greater than one?"* (see Figure 10.8)

10.3.2.4 Sorting

When you use a query to sort, you do not change the physical order of the records in the underlying table (that is, you do not sort the table). As a result, different queries based on the same table can display the records in a different sequence.

11 Set the **Sort** row to sort the results of your query by `Description` in ascending (A→Z) order (see Figure 10.3).

When multiple fields are used in a query to sort (e.g., sort first by last name, then by first name), fields on the left take precedence over fields on the right.

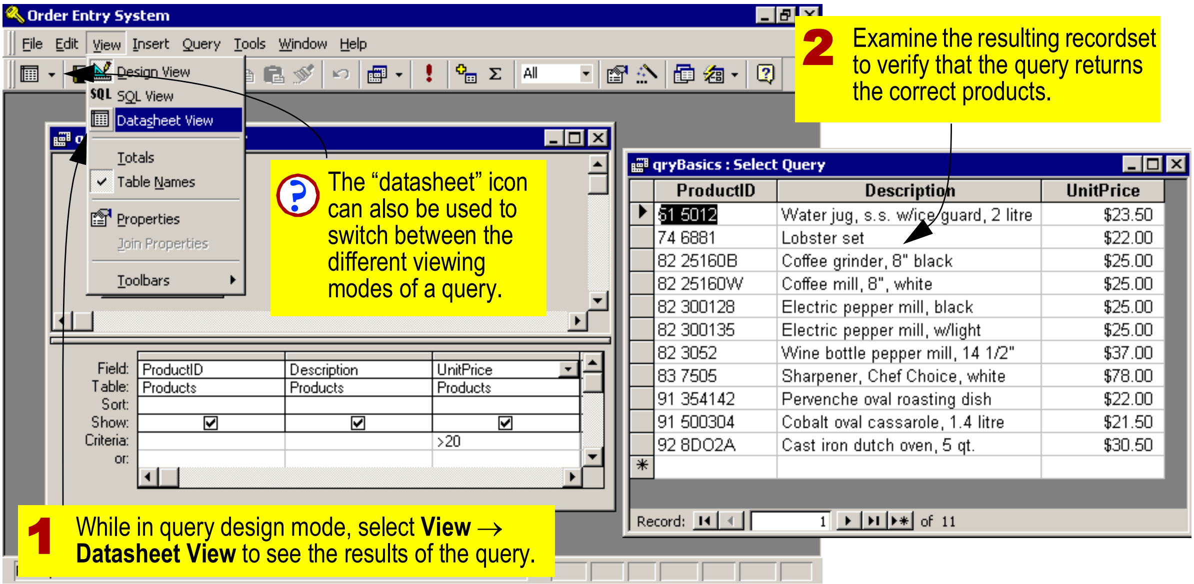
Since a query is never used by itself to display data to a user, you can move the fields around within the query definition to get the desired sorting precedence. You then reorder the fields in the form or report for presentation to the user.

10.3.2.5 Joining

In Lesson 7, you were advised to break you information down into multiple tables with relationships between them. In order to put this information back together in a usable form, you use a join query.

12 Save and close `qryBasics`.

FIGURE 10.6: View the results of a select query.

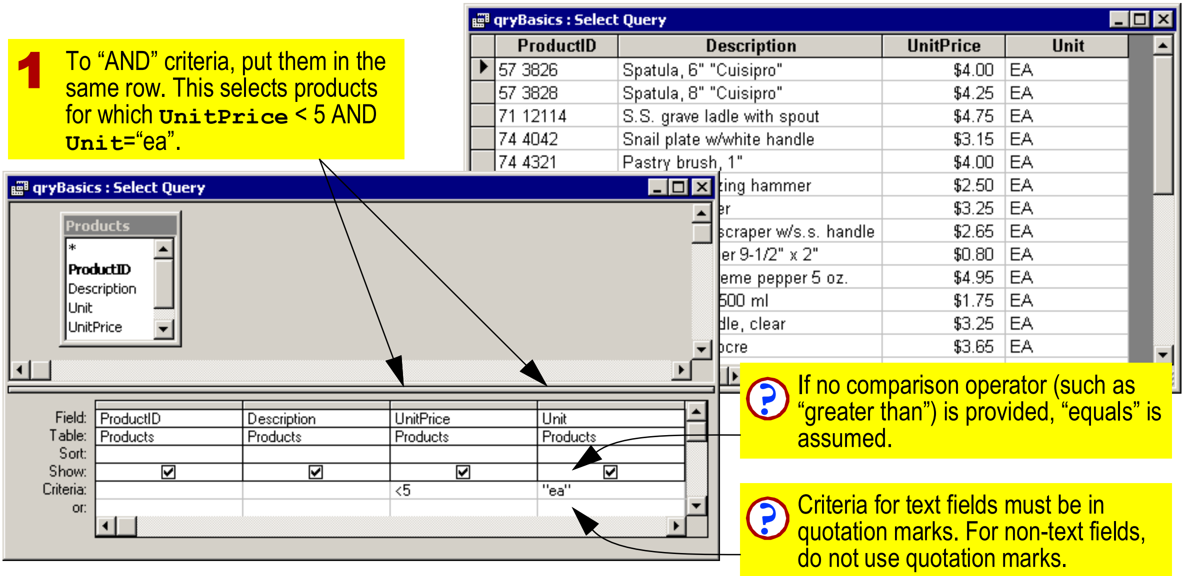


13 Open the relationships window and ensure you have a relationship declared between `Customers` and `Regions`. If not, declare one and ensure referential integrity is enforced (review Section 7.3 as required).

14 Create a new query called `qryJoin` based on the `Customers` and `Regions` tables.

15 Project `Regions.RegionName`, `Customers.CustName`, and `Customers.City` as shown in Figure 10.9.

FIGURE 10.7: Complex queries using ANDed criteria.

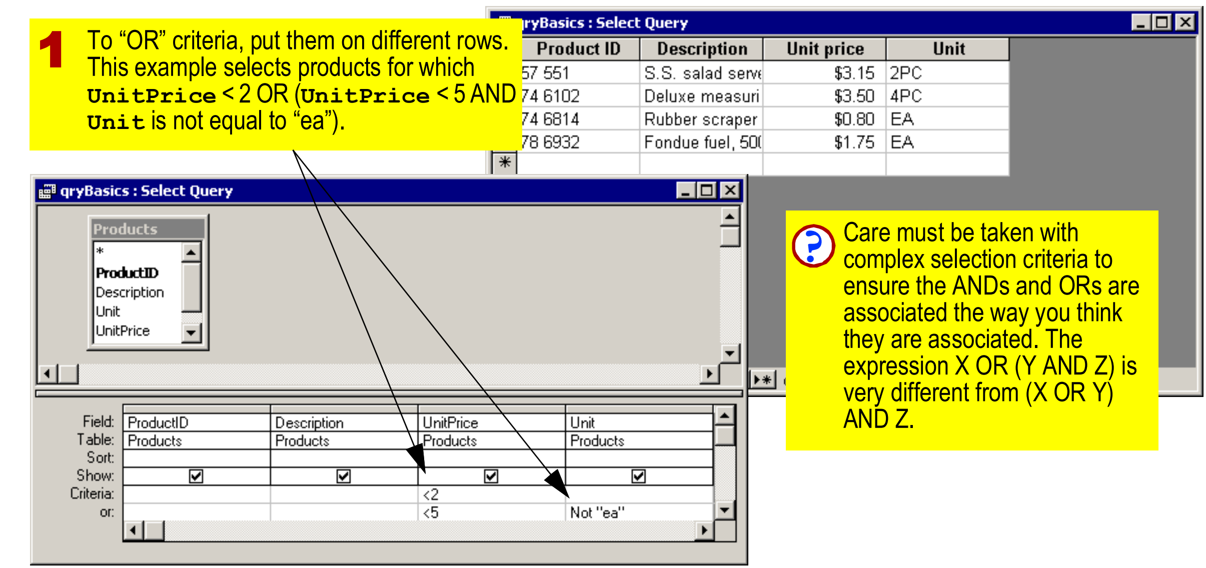


When a query is based on more than one table, it is often the case that certain field names are used in multiple tables (e.g., `RegionCode` is used in both `Customers` and `Regions`). To eliminate ambiguity, the ***<table name>.<field name>*** notation is used to refer to fields.

16 Click on the column selector for `Regions.RegionName` and drag the field to the far right of the query definition grid (this is to show you how easy it is to re-order your fields once they have been added).

17 Switch to datasheet view and notice that information from both tables is shown.

FIGURE 10.8: Complex queries using ORed criteria.



If you neglected to populate the **Customers.RegionCode** field in Section 8.5, the result of your join query will be empty (no records). A join matches each record on the "many" side with its corresponding record on the "one" side. If **Customers.RegionCode** is NULL for all customers, no records are returned since there is no record in the **Regions** table with a **RegionCode** = NULL.

10.3.3 Using queries to edit records

Once ACCESS knows the **RegionCode** of a customer, it can uniquely identify the region to which the customer has been assigned. This allows us to show the more user-friendly region name instead of the single-letter **`RegionCode`** field. The result is similar to the "monolithic" table design discussed in Section 7.1.1. However, there are some important differences, as you will see in the following exercises.

10.3.3.1 Editing a record on the "many" side

Show me (lesson10-4.avi)

18 Add **`Customers.RegionCode`** to your query. Ensure you add **`Customer.RegionCode`**, not **`Regions.RegionCode`**.

19 Switch to datasheet view, as shown in Figure 10.10.

20 Change the **`RegionCode`** value for ROSCH DRY GOODS from "E" to "K" (assume the firm is being transferred to the special "key accounts" region). When you click on the record selector to save the record, you will notice that the value of the **`RegionName`** field changes automatically.

FIGURE 10.10: Use a join query to make a change to a record on the "many" side of the relationship.

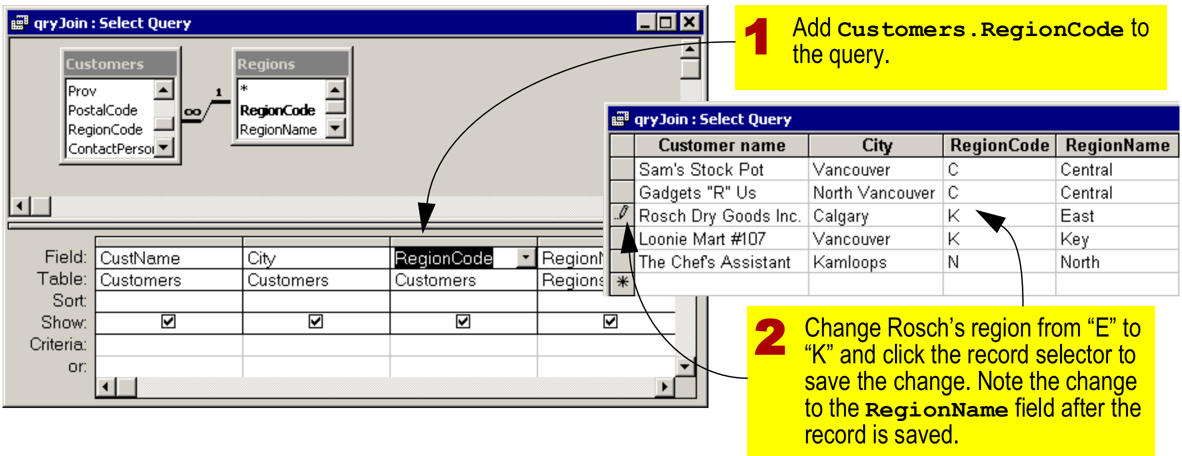
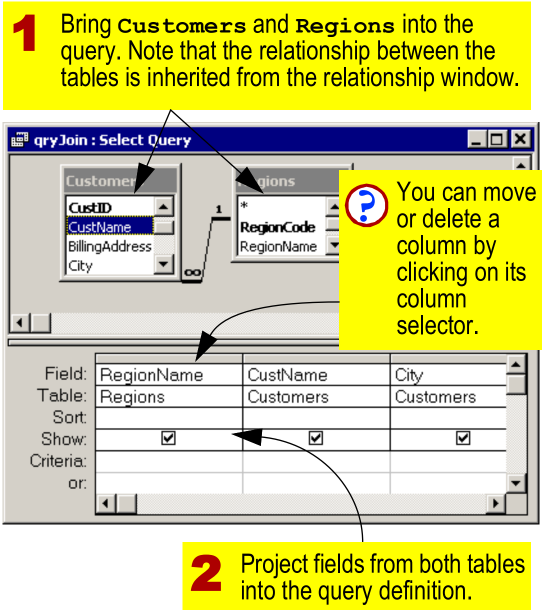


FIGURE 10.9: Create a query that joins ***Customers*** and ***Regions***.

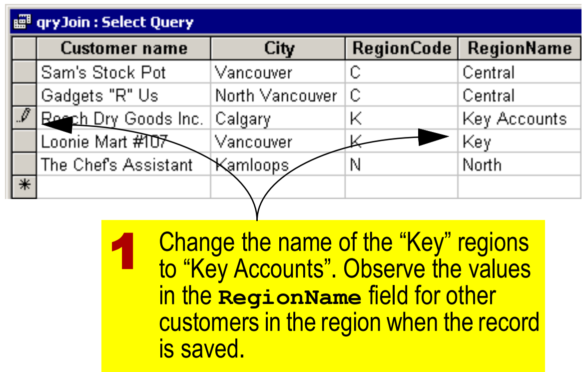


10.3.3.2 Edit a record on the "one" side

Show me (lesson10-5.avi)

21 Click on one of the "Key" values in the **RegionName** field and change it to a more descriptive value "Key Accounts", as shown in Figure 10.11.

FIGURE 10.11: Use a join query to make a change to a record on the "one" side of the relationship.



22 Save the record and notice how the change propagates to all key account regions.

Normalized tables and join queries provide a very efficient means of administering data. In this exercise, it is important to realize that the **RegionName** field in the query is a direct "window" into the **Regions** table. Since the name of the region is only stored in the database once, it only needs to be changed once.

10.3.4 Using queries to add records

When adding records to a table that is on the "many" side of a relationship, it is often helpful to use a join query to provide *feedback* to the user during data entry.

23 Create a new query for adding customers called **`qryCustomerAdd`**. Project the following fields into the query definition: **`Customers.*`**, **`Regions.RegionName`**.

Since the purpose of this query is to add records to the **`Customers`** table, it is clear that you need to project *all* of the customer fields. The asterisk (*) provides a convenient means of doing this, even if the fields in the table change over time.

24 Switch to datasheet view and add a new customer record (make one up).

25 Note that when the **`RegionCode`** is specified, the name of the region is "looked-up" and filled in automatically. This is what is meant by feedback—the additional region information helps you determine whether you have entered the correct **`RegionCode`**.

10.4 Discussion

10.4.1 Naming conventions for database objects

As discussed in the section on field names in Section 5.4.5.1, there are relatively few naming restrictions for database objects in ACCESS. However, a clear, consistent method for choosing names can save time and avoid confusion later on.

Although there is no hard and fast naming convention required for the project, the following guidelines should be appended to those introduced in Section 5.4.5.1:

- **Use meaningful names** — An object named **`Table1`** does not tell you much about the contents of the table. Furthermore, since there is no practical limit to the length of the names, you should not use short, cryptic names such as **`s96w_b`**. As the number of objects in your database grows, the time spent carefully naming your objects will pay itself back many times.
- **Give each type of object a distinctive prefix (or suffix)** — This is especially important in the context of queries since tables and queries cannot have the same name. For example, you cannot have a table named **`Orders`** and a query named **`Orders`**. However, if all your query names

are of the form **qryOrders**, then distinguishing between tables and queries is straightforward.

Table 10.1 shows a suggested naming convention for ACCESS database objects (you will discover what these objects are in the course of doing the tutorials).

TABLE 10.1: A suggested naming convention for ACCESS database objects.

Object type	Prefix	Example
table	(none)	OrderDetails
query	qry	qryBackOrders
parameter query	pqry	pqryItemsInOrder
form	frm	frmOrders
sub form	sfrm	sfrmOrderDetails
report	rpt	rptInvoice
sub report	srpt	srptInvoiceDetails
macro	mcr	mcrOrders
Visual Basic module	bas	basUtilities

10.4.2 Using queries to populate tables on the "many" side of a relationship

In Section 10.3.4, you added a record to the **Customers** table to demonstrate the automatic lookup feature of ACCESS. A common mistake when creating queries for adding or modifying data on the "many" side of a relationship is to forget to project the foreign key of the table you intend to populate.

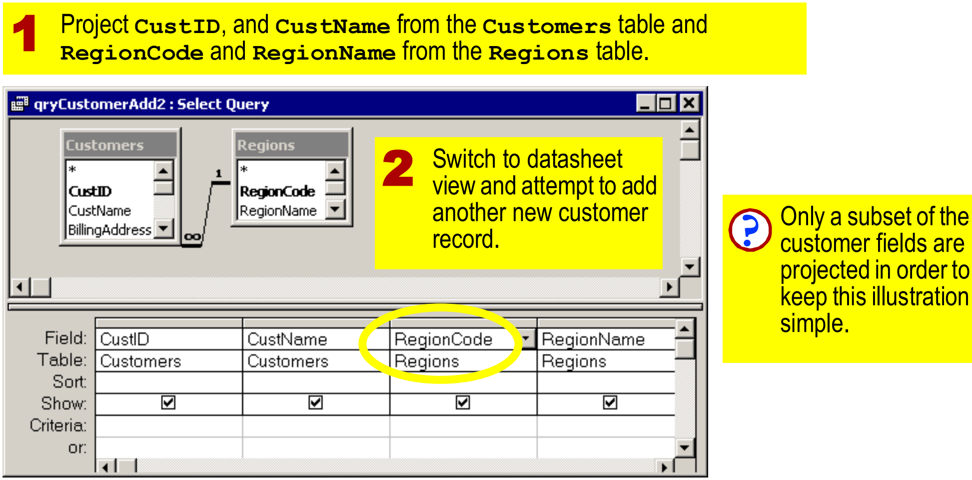
For example, faced with two tables containing the **RegionCode** field, you might project the one from wrong table (the "one" side) into your query definition. To illustrate the problem, do the following:

26 Create a new join query called **qryCustomerAdd2** that projects **Regions.RegionCode** instead of **Customers.RegionCode**, as shown in Figure 10.12.

27 Attempt to add a new record. You will be unable add anything and will get an error message in the status bar at the bottom of the screen.

Even if you were allowed to add a record using this query, the results would be dangerous. In **qryCustomerAdd2**, the **RegionCode** field is bound to the **Regions** table instead of the

FIGURE 10.12: Create a data-entry query without a foreign key.



Customers table. Entering a value of “N” into the field simply overwrites the current value of **Regions.RegionCode**.

10.4.3 Non-updatable recordsets

Another problem that sometimes occurs when creating join queries is that the query is not quite right in some way. In such cases, ACCESS will allow you to view the results of the query, but it will not allow you to change the data in any way.

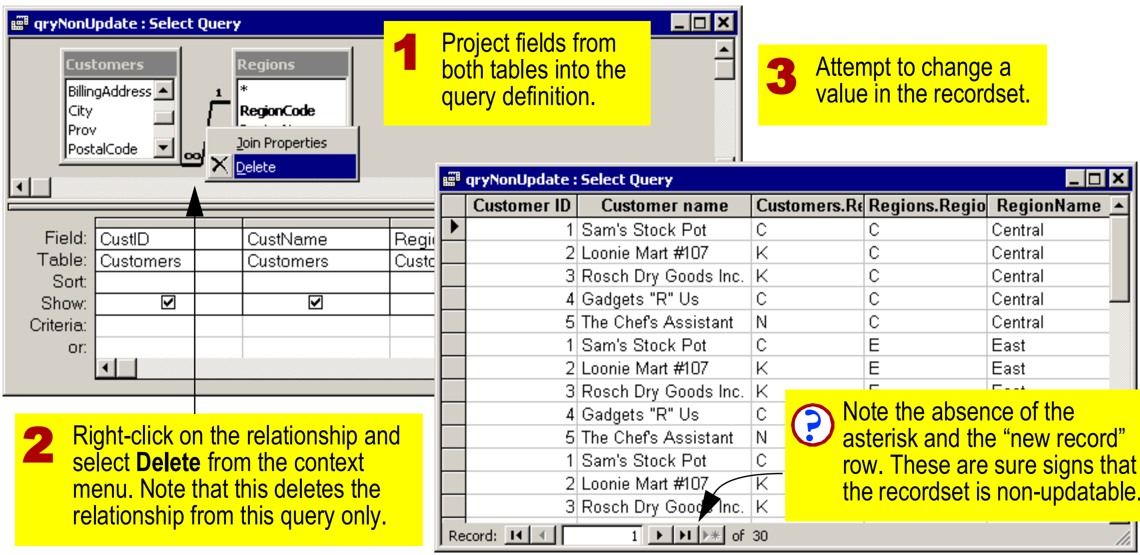
In this section, will look at a nonsensical query that results from an incompletely specified relationship. As you will probably discover, however, there are many different ways to generate nonsensical queries.

Show me (lesson10-6.avi)

28 Create a new query called **qryNonUpdate** based on the **Customers** and **Regions** tables.

29 Delete the **RegionCode** relationship and project a couple of fields from both tables, as shown in Figure 10.13.

FIGURE 10.13: Create a non-updatable recordset.



30 View the results of the query.

The result of this query is known as a **Cartesian join** (or cross product)—every customer is combined with every region regardless of the value of the **RegionCode** field. ACCESS recognizes that this is not a standard join query and designates the recordset as **non-updatable**.

Later, when building forms, you may accidentally base a form on a non-updatable recordset. You then may spend a great deal of time trying to get the form to work when the real problem is the underlying query. A quick check for a “new record” row in all your new queries can save time and frustration later on.

10.5 Application to the assignment

31 Create a query called `qryOrderDetails` that joins the `OrderDetails` and `Products` tables. When you enter a valid `ProductID`, the information about the product (such as name, quantity on hand, and so on) should appear automatically.

Show me (lesson10-7.avi)

Due to the referential integrity constraints you specified in Lesson 7, you will not be able to save any records you use to test `qryOrderDetails` until you have a corresponding `OrderID` in the `Orders` table. At this point, you can simply hit the **Esc** key to discard the test data without saving it.

If the name of the product does not appear when you enter a valid `ProductID`, you have used the wrong `ProductID` field. Review Section 10.4.2 and ensure you understand (and fix) the error before continuing.

32 Create a query to show additional information about the customer who placed the order when looking at data in the `Orders` table.

33 Enter the first order (or at least a handful of order details from the first order) into your system by typing the information directly into tables or queries. Doing this correctly is going to require some thought.

Adding a new order involves creating a single `Orders` record and several `OrderDetails` records. You must also consult the `Products` table to determine the quantity of each item to ship (you cannot ship product that you do not have in inventory), and determine the default selling price of each good.

HINT: Much of the effort involved in switching between tables is eliminated if you use the `qryOrderDetails` query you created above for this exercise.

Entering orders into your system will be much less work once the input forms and event-driven programs are in place. The goal at this point is to get you thinking at a very specific level about the order entry process and what needs to be automated.

Lesson 11: Calculated fields using QBE

11.1 Introduction: Virtual fields

A calculated field is a "virtual field" in a query. The field is *virtual* in the sense that it is not stored anywhere in the database. Instead, it is calculated dynamically when the query is used.

The value of the calculated field is typically a function of one or more fields in the underlying table. For example, in the **`Employees`** table, you have fields for the employees' first and last names (**`EMP_FNAME`** and **`EMP_LNAME`** respectively). However, for some printed reports and mailing labels, you may want to combine the first and last names into a single value such as "Vivian Peng". Although you could add a new field to the table called **`emp_fullname`** and populate it, this would be an unsatisfactory solution for at least three reasons:

1. An **`emp_fullname`** field replicates information that already exists in the database (albeit in a different format) and is therefore wasteful of disk space.
2. If Vivian changes her last name, the change has be made in both the **`EMP_LNAME`** and **`emp_fullname`** fields.

Whenever a single change in the business environment requires multiple changes in the information system, you can be sure that the information system will eventually be full of inconsistent data.

3. The employee data you are using is from the payroll application. Since you have not been granted write-access to the payroll database, you are not permitted to change the structure of its tables or modify its data.

To get around these problems, you can define a calculated field called **`FullName`** (or whatever name you like) that is not stored in the **`PAY_EMPS`** database.

11.2 Learning objectives

- create a calculated field
- understand why ACCESS add square brackets around field names
- understand the use of the ampersand operator (&)

11.3 Exercises

11.3.1 Creating calculated fields

To create the **`FullName`** calculated field in an ACCESS query, you use the following syntax:

`FullName: EMP_FNAME & " " & EMP_LNAME`

Generally, the syntax of a calculated field is of the form: ***<field name>*: *<expression>***, where ***<field name>*** is a name you provide for the calculated field.

The name of a calculated field can be just about anything, as long as it is unique *within the query* (i.e., it cannot be the same as the name of a field in one of the query's underlying tables). Generally, it is best to follow the same conventions you use when creating regular fields.

The ***<expression>*** part is any combination of fields, operators, and functions that evaluates to the desired value. In the **`FullName`** example used above, the expression is simply the concatenation of two text fields with a space inserted (see Section 11.4.1 for more information about using the ampersand operator to concatenate text).

Show me (lesson11-1.avi)

1 Create a new query called **`qryEmployees`** based on the **`Employees`** table.

2 Create the **`FullName`** field, as shown in Figure 11.1.

3 View the query to verify the results, as shown in Figure 11.2.

4 Switch back to the design view of the query. You will notice that ACCESS has added square brackets to the names of your fields.

In ACCESS, square brackets are used to indicate the name of a field (or some other object in the ACCESS environment). If your field name contains one or more blank spaces (e.g., **`[Last Name]`**), the use of square brackets is *mandatory*: the brackets tell ACCESS that **`Last`** and **`Name`** are not two separate expressions. If you use single-word field names (strongly recommended), the use of square brackets is entirely optional.

11.3.2 Errors in queries

It may be that after defining a calculated field, you get the "Enter Parameter Value" dialog box shown in Figure 11.3 when you run the query.

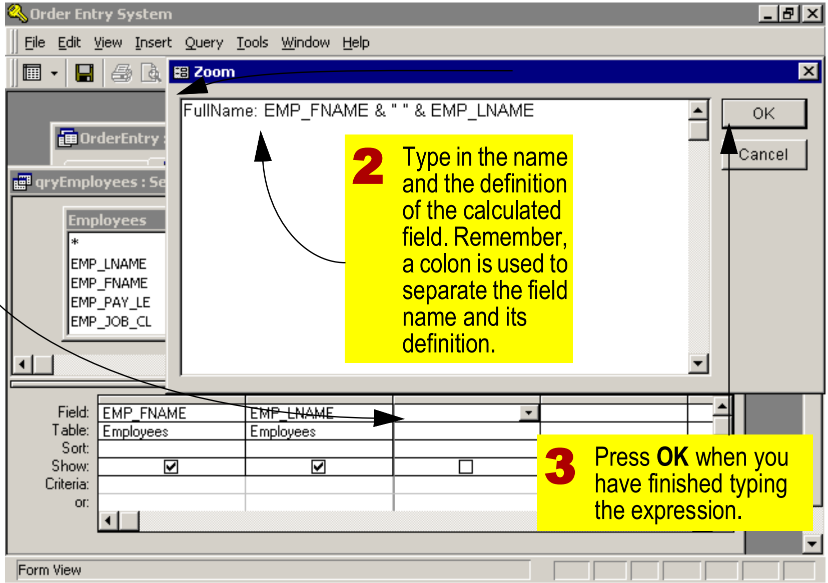
The "Enter Parameter Value" box pops up whenever you spell something in your

FIGURE 11.1: Create a calculated field based on two other fields.

1 Put the cursor in the field row of an empty column and invoke the zoom window by either pressing **Shift-F2** or right-clicking and selecting **Zoom** from the context menu.

The "zoom" window provides more room to type than the tiny space in the query definition grid.

ACCESS ignores case in expressions. Thus, the names **`emp_fname`** and **`EMP_FNAME`** are equivalent.



calculated field definition incorrectly. ACCESS cannot resolve the name of the misspelled field and thus asks the user for the unknown value.

Show me (lesson11-2.avi)

The term "spelling mistake" is used in its broadest sense to cover all sorts of goofs and slips. For example, one way to get an unwelcome "Enter Parameter Value" box is to use a non-existent field name such as **`ProductNo`** instead of **`ProductID`** in a calculated expression. You can stare at **`ProductNo`** for hours and insist that it is spelled correctly. The lesson: check the names of your fields very carefully when you create calculated fields.

FIGURE 11.2: View the results of the calculated field.

qryEmployees : Select Query

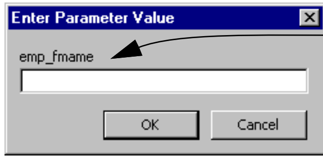
EMP_FNAME	EMP_LNAME	FullName
Gerard	Huff	Gerard Huff
Vivian	Peng	Vivian Peng
Jocelyn	Scorer	Jocelyn Scorer
Bill	Williams	Bill Williams
Hamid	Hassan	Hamid Hassan
Joy	Kakuchi	Joy Kakuchi
Richard	Mason	Richard Mason
Russell	Plevy	Russell Plevy
Ben	Sidhu	Ben Sidhu
Sabine	Villeneuve	Sabine Villeneu

The **`FullName`** field is calculated for each record by concatenating the employee's first name, a space, and last name.

To eliminate the unknown parameter problem, simply return to design mode and correct the spelling mistake.

As the tutorials progress, you may be surprised to see the "Enter Parameter Value" box in different situations that are seemingly unrelated to queries. For example, the box may pop up when opening a form. Do not panic when this happens—instead, simply look at the name of the expression name that ACCESS cannot evaluate and find the mistake in the underlying query. Then, fix the mistake.

FIGURE 11.3: The user is prompted for the value of a misspelled field.



A field name in the calculated field has been misspelled. Since ACCESS does not know the value of "emp_fmame", it asks the user.

11.3.3 Creating mathematical expressions

In Section 11.3.1, you used a calculated field to transform two text fields into a more useful format. However, as the name implies, calculated fields can also be used to evaluate mathematical expressions.

Assume, for example, that you are considering creating a price list of your products for customers in another country. It makes little sense to have a separate field stored in the table for each currency because there is a simple functional relationship between any two currencies. Not only would separate fields waste disk space, all price changes would have to be repeated for each currency. This would be bad news if you carried thousands of products.

In this section, you are going to create a list that shows the price of your products in Canadian Dollars and Belgian Francs.

5 Create a new query based on the `Products` table and save it as `qryPriceList`.

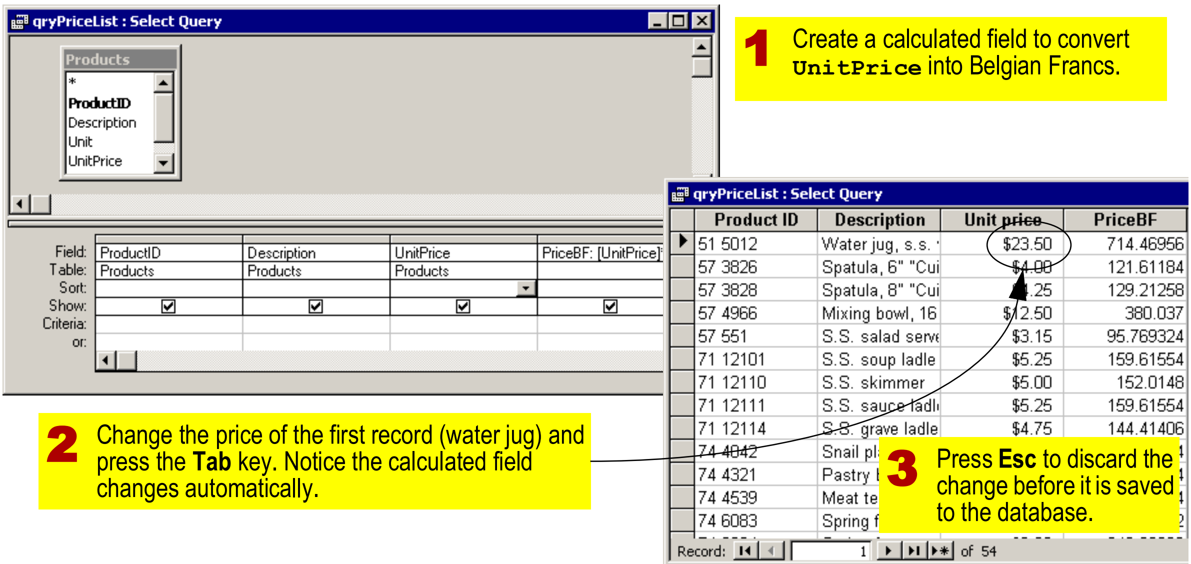
6 Project the `ProductID`, `Description`, and `UnitPrice` fields.

7 Create a calculated field to convert Canadian Dollars (the currency in which `UnitPrice` is stated) into Belgian Francs (assume that CDN$1 = BF30.40296):

`PriceBF: UnitPrice * 30.40296`

8 Switch to datasheet mode and inspect the results, as shown in Figure 11.4.

FIGURE 11.4: Use a calculated field to show product prices in multiple currencies.



The approach described above works well enough if you are going to print a price list every couple of months. However, in an environment in which exchange rates must be current (e.g., an online store), you would have to edit the `PriceBF` field constantly to update the "hard coded" exchange rate. Naturally, there are ways to do this automatically, but they are a bit beyond us at this early stage.

9 Change the value of `UnitPrice` for the first item in the datasheet. When you press the **Tab** key, you will see that the value of `PriceBF` changes instantly and automatically.

10 Press the **Esc** (escape) key to undo the price change without saving it to the database.

Recall from Figure 5.6 that changes to the record buffer are not actually saved to the disk until you move to a different record or explicitly save the record. The escape key (or **Ctrl-Z** or **Edit** → **Undo**) discards changes made to the record buffer before they are written to the disk.

11 Attempt to change the value of the `PriceBF` field. Notice the error message displayed in the status bar in the bottom-left corner of the ACCESS window.

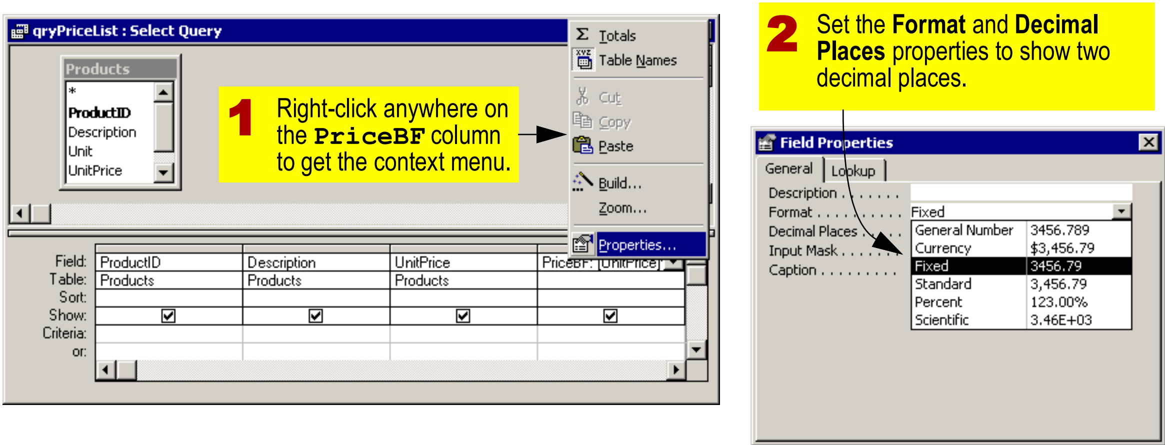
11.3.4 Formatting a calculated field

Users should never see queries or tables—all on-screen interaction with the application should occur through forms and printed output should be generated using the report writer. As such, it really does not matter what the output of a query output looks like because all data will be formatted in a form or report for end-user consumption anyway. This fact notwithstanding, it is often useful to specify the format of calculated fields so that the format is inherited when the forms and reports are created.

To illustrate, recall Figure 11.4. It is clear that `UnitPrice` has taken the currency format of the underlying `UnitPrice` field in the `Products` table. Since my computer is set up for use in Canada, all currency values are shown with the (Canadian) dollar sign. In contrast, `PriceBF` is interpreted by ACCESS simply as floating point number. Although there is no option (on a computer configured for use in North America) to format a field as Belgian Francs, you can designate the field as a fixed point number (with two decimal places to show the number of *centimes*).

Show me (lesson11-3.avi)

FIGURE 11.5: Format the `PriceBF` calculated field.



12 Switch to the design view of `qryPriceList`.

13 Right-click on the field selector for the `PriceBF` field, as shown on the left-hand side of Figure 11.5.

14 Select **Properties** from the context menu to get the properties dialog shown on the right-hand side of Figure 11.5.

15 Set the value of the **Format** property to "Fixed".

16 Set the value of the **Decimal Places** property to "2".

17 Verify that the `PriceBF` field now has only two digits to the right of the decimal.

Whenever the `qryPriceList.PriceBF` field is used in a form or a report from this point forward, it will be displayed with the correct number of decimal places.

To display the prices "in french", a number of trickier formatting changes are required. For example, the decimal has to be replaced with a comma. The following calculated field converts the **PriceBF** field into an appropriately-formatted text field (figuring out the expression is left as an exercise):

```
PrixFB: CStr(CInt(PriceBF)) & "," &
Right(PriceBF,2) & " FB"
```

11.3.5 Complex calculated fields

If you attempted to create an input mask for the **Products.ProductID** field, you will have already noticed that the field possess some structure. For example, the first two digits appear to indicate a product category.

A better table design would have the category information stored in a separate field in the **Products** table. However, you have very little freedom to make changes to the structure of the data since the current **ProductIDs** are used throughout your company and by your customers. When you encounter such legacy data, the best you can do is work around it.

In this section, we are going to assume the following scenario: Certain customers are interested only in small ceramic items and stainless steel utensils. You need to create a query to do the following:

1. Limit the product list to ceramics and stainless steel utensils only.
2. Map the product code to category names that are familiar to your customers (such as "ceramics" and "utensils")
3. Show the **ProductID** and **Description** fields and a calculated **Category** field.

Assume that **ProductIDs** that start with "71" are stainless steel utensils and **ProductIDs** that start with "88" are ceramic items.

18 Create a new query based on the **Products** table. Project **ProductID** and **Description**. Call it **qryProductListCategories** or something appropriate.

19 Enter the following criterion for the **ProductID** field:

```
Like "71*" OR Like "88*"
```

The **Like** operator allows you to use wildcards in your text-based criteria. In ACCESS, "*" is used to represent any sequence of characters and "?" is used to represent any single character.

To map the first two digits of **ProductID** to categories, you are going to use the "immediate if" function, **iif()**. The **iif()** function uses the following syntax:

NL
```
iif(<expression>, <output if true>,
<output if false>)
```

If you have done any programming, you will recognize `iif()` as a shorthand version of the following code:

```
NL  If <expression> = TRUE Then
NL    Return <output if true>
NL  Else
NL    Return <output if false>
NL  End If
```

The syntax of the `iif()` statement appears to depend on your locale. For example, some versions of ACCESS used in Europe require that the arguments in the `iif()` statement be separated by semicolons rather than commas. The best way to resolve an international issue (if you encounter one) is to verify the syntax of the function using your localized on-line help facility.

20 Create a new calculated field called `Category` as shown in Figure 11.6. Use the following field definition:

```
NL  Category: iif(Left(ProductID,2)=88,
    “Ceramics”, “Utensils”)
```

Show me (lesson11-4.avi)

21 Verify the results, as shown in Figure 11.7.

Predefined functions such as `iif()` and `Trim()` functions are discussed in Section 11.4.2.

11.4 Discussion

11.4.1 The concatenation operator

The ampersand operator (&) is like any other operator (e.g., +, -, ×, ÷) except that it is intended for use on strings of characters. In ACCESS, the ampersand simply adds one string on to the end of another string (hence its other name: the “concatenation” operator). For example, the expression

```
NL  “First string” & “Second string”
```

yields the result

```
NL  First stringSecond string
```

If a space is include within the quotation marks of the second string (`“ Second string”`), the result is:

```
NL  First string Second string
```

Use of the ampersand to concatenate text is not widespread outside of MICROSOFT ACCESS and MICROSOFT VISUAL BASIC. In many computer languages, the plus sign (+) is used instead.

11.4.2 Predefined functions

In computer programming, a function is a small program that takes zero or more **arguments** (or **parameters**) as input, does some processing, and returns a value as output. A *predefined* (or *built-in*) function is a function that is provided as part of the programming environment.

For example, `cos(x)` is a predefined function in many computer languages—it takes some number `x` as an argument, does some processing to find its cosine, and returns the answer. Note that since this function is predefined, you do not have to know anything about the algorithm used to find the cosine, you just have to know the following:

1. what to supply as inputs (e.g., a valid numeric expression representing an angle in radians),
2. what to expect as output (e.g., a real number between -1.0 and 1.0).

FIGURE 11.6: Create a calculated field using the "immediate if" function

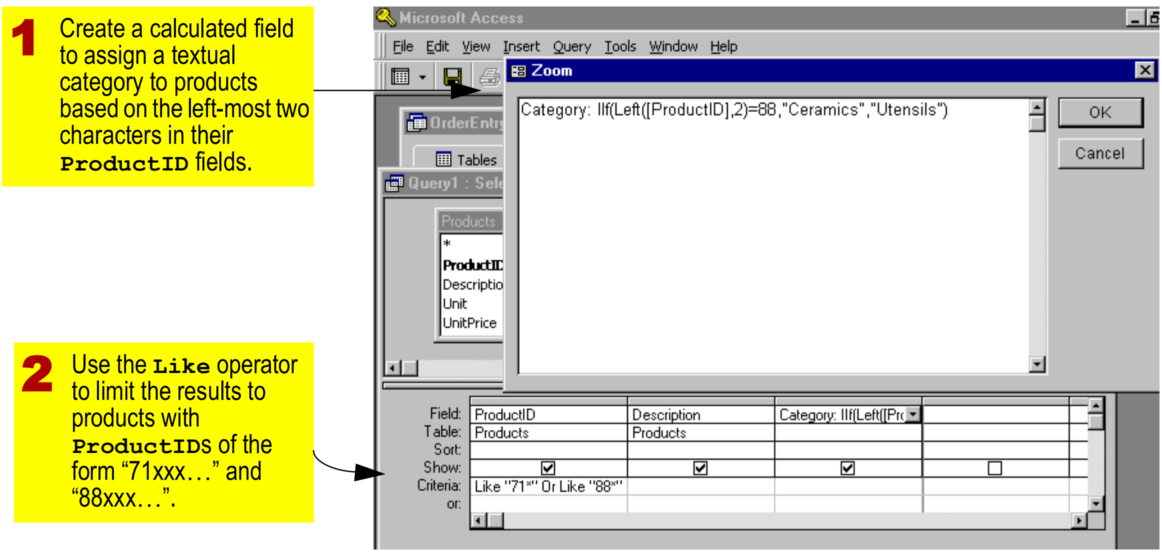


FIGURE 11.7: Results showing the calculated **`Category`** field.

Query1 : Select Query

Product ID	Description	Category
88 3113196015	Salad plate, ocre	Ceramics
88 3122216004	Pasta dish, cobalt	Ceramics
88 3115106014	Sugar, hunter green	Ceramics
88 3115106004	Sugar, cobalt	Ceramics
88 3114106014	Creamer, hunter green	Ceramics
88 3114106057	Creamer, red	Ceramics
88 4017	Mug, "Fat Cat"	Ceramics
88 4077	Mug, white hearts	Ceramics
88 4491	Mug, window cats	Ceramics
88 4742	Mug, polar bear	Ceramics
71 12101	S.S. soup ladle	Utensils
71 12110	S.S. skimmer	Utensils
71 12111	S.S. sauce ladle	Utensils
71 12114	S.S. grave ladle with spout	Utensils

Record: 1 of 14

The on-line help system provides these two pieces of information (plus a usage example and some additional remarks) for all predefined functions in ACCESS.

11.5 Application to the assignment

22 Add a calculated field called **`ExtendedPrice`** to the **`qryOrderDetails`** query you created in Section 10.5. Extended price is defined as the number of items of a particular product multiplied by the price of each object.

You have several choices to make when defining this field: Do you use quantity ordered or quantity shipped? Do you multiply by the default price of the product (**`UnitPrice`**) or the price at which the product is sold to the customer (**`ActualPrice`**)? Make sure you can answer these *business* questions before creating the field.

12.1 Introduction: The difference between QBE and SQL

Query-By-Example (QBE) and Structured Query Language (SQL) are both well-known, industry-standard languages for extracting information from relational database systems. The advantage of QBE (as you saw in Lesson 10 and Lesson 11) that it is graphical and relatively easy to use. The advantage of SQL is that it has achieved nearly universal adoption within the relational database world.

With only a few exceptions (which you probably will not encounter in this project) QBE and SQL are completely interchangeable. If you understand the underlying concepts (projection, selection, sorting, joining, and calculated fields) of one, you understand the underlying concepts of both. In fact, in ACCESS you can switch between QBE and SQL versions of your queries with the click of a mouse.

Although you typically use QBE to create queries in ACCESS, the ubiquity of SQL in the rest of the database world necessitates a brief overview.

12.2 Learning objectives

- understand the difference between QBE and SQL
- create an SQL query
- use SQL as a data definition language

12.3 Exercises

12.3.1 Select queries

Recall from Lesson 10 that a select query is a query that allows you to select and organize data from one or more underlying tables. In these exercises, you are going to use SQL to achieve the same result.

The queries you build in these exercises are for practice only. They are not an integral part of your order entry project and it is not critical that they be saved as part of your database.

1. Create a new query, but close the “Show Table” dialog box without adding any tables.

2. Select **View** → **SQL** from the main menu to switch to the SQL editor. You get white text

editor that contains nothing but an empty SELECT statement.

A typical SQL statement resembles the following:

```
NL  SELECT ProductID, Description
NL  FROM Products
NL  WHERE Unit <> "ea";
```

There are four parts to the SQL statement:

1. **SELECT** $<field_1, field_2, \ldots, field_n>$ … — specifies which fields to project;
2. … **FROM** $<table>$ … — specifies the underlying table (or tables) for the query;
3. … **WHERE** $<condition_1$ **AND/OR** $condition_2, \ldots,$ **AND/OR** $condition_n>$ — specifies one or more conditions that each record must satisfy in order to be included in the results set;
4. **;** (semicolon) — all SQL statements must end with a semicolon (but if you forget, ACCESS will add one for you).

Despite the use of the new line symbol (NL) in the examples in this lesson, the SQL interpreter ignores whitespace and allows lines to be split between any two words. In order to make the statements more readable, however, it is common practice is to use a new line for each major SQL keyword (**SELECT**, **FROM**, **WHERE**, etc.).

You will now use these basic constructs to create your own SQL query:

Show me (lesson12-1.avi)

3 Type the following into the SQL window:

```
NL  SELECT ProductID, Description
NL  FROM Products
NL  WHERE Unit <> "ea";
```

If you look closely at your keyboard, you will notice that there is an equals sign (=) but no not-equals sign (≠). In some computer languages—including SQL—not-equals is represented using the less-than and greater-than signs together (<>).

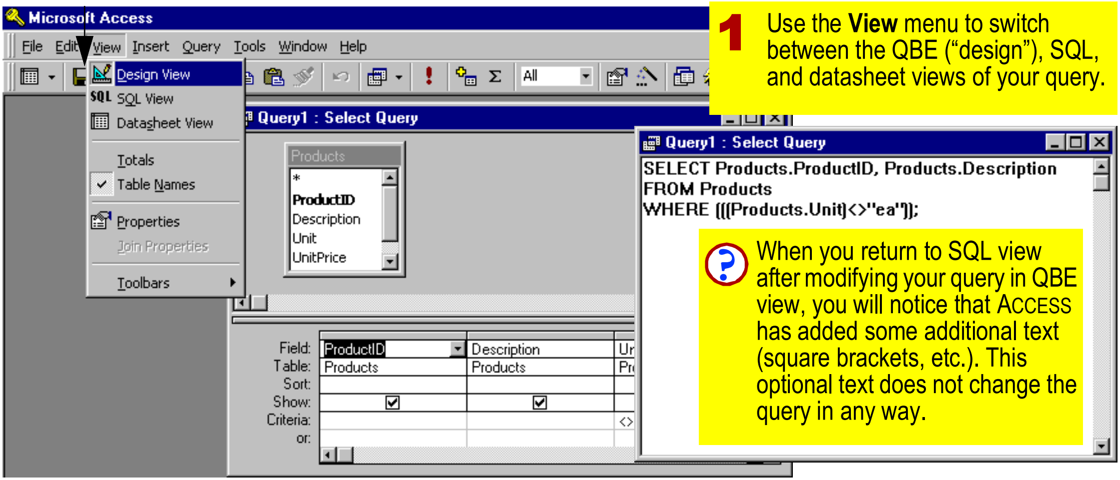
4 Select **View** → **Datasheet** to view the result, as shown in Figure 12.1.

5 Select **View** → **Query Design** to view the query in QBE mode, as shown in Figure 12.2.

ACCESS automatically translates between QBE and SQL versions of the query.

6 Save your query as **qrySQL**.

FIGURE 12.2: The SQL and QBE views are interchangeable.



12.3.2 Complex WHERE clauses

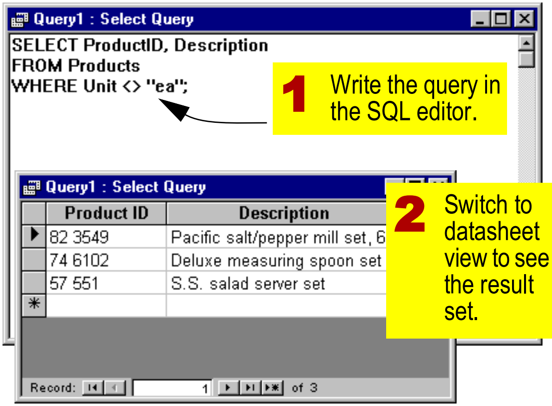
As in QBE, you can use Boolean operators such as AND, OR, and NOT in your WHERE clauses to specify complex selection criteria.

7 Change your query to show all the products that normally sell for less than $2 each AND all the products that are not sold in units of one, but which normally sell for less than $5.

```
NL  SELECT *
NL  FROM Products
NL  WHERE UnitPrice < 2
NL  OR (UnitPrice < 5 AND NOT
    Unit = "ea");
```

As in QBE, selecting the asterisk (*) is shorthand for all fields in the underlying table. Thus, `SELECT *` is identical to `SELECT ProductID, Description, Unit, UnitPrice ….`

FIGURE 12.1: A simple SQL select query.



SQL is a computer language. And like any computer language, it is picky about what you type. If your query contains spelling mistakes or other errors, it will not execute.

Note that since `Unit` is a text field, its criterion must be a string (in this case, a **literal string** enclosed in quotation marks). `UnitPrice`, in contrast, is a numeric field and its criterion must be a number (no quotation marks).

12.3.3 Join queries

When you create a join query in QBE and switch to SQL, you see an “INNER JOIN” statement that is unique to MICROSOFT’s SQL for the JET database engine. The ANSI[1] standard version of SQL uses a different syntax for joining tables (which ACCESS also supports). In this section, you will use the ANSI standard approach to join the `Customers` and `Regions` tables.

8 Create a new SQL query called `qrySQLJoin` and use the WHERE clause to specify the join relationships between the tables:

```
NL  SELECT CustName, City, RegionName
NL  FROM Customers, Regions
NL  WHERE Customers.RegionCode =
    Regions.RegionCode
NL  AND RepID = 9
NL  ORDER BY CustName;
```

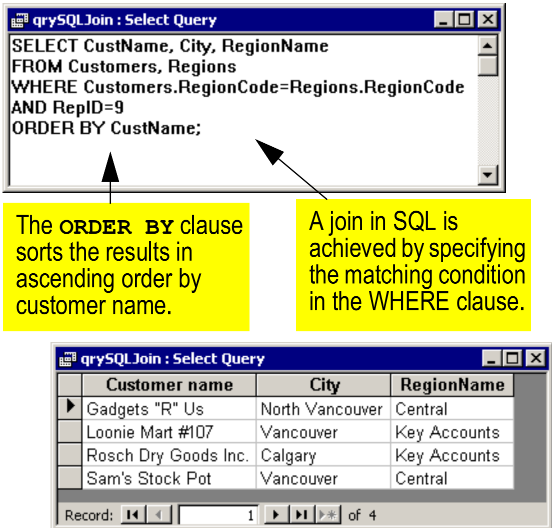
9 View the result set and confirm that all the customers in regions serviced by `RepID` = 9 (Ben Sidhu) are included, as shown in Figure 12.3.

If you neglected to populate the `Regions.RepID` column in Section 8.5, your query will return an empty recordset.

---

1 American National Standards Institute

FIGURE 12.3: Using SQL to join two tables.



When we created join queries in QBE (recall Section 10.3.2.5), we did not worry about linking primary keys to foreign keys—all the relationships were inherited from those created using the relationships window in Lesson 7.

However, the relationships window is a feature of ACCESS, not relational databases generally. Thus, SQL provides its own mechanism for specifying joins using the WHERE clause. To illustrate, consider the first part of the WHERE clause you just created:

```
WHERE Customers.RegionCode =
Regions.RegionCode
```

The only records that are joined from the **`Customers`** and **`Regions`** table are those for which the value of **`RegionCode`** is the same in both tables. Thus, you use the WHERE clause in SQL to specify equality between the primary key on the "one" side and the foreign key on the "many" side. This equality condition is equivalent to the little connecting lines between tables used by ACCESS in the QBE interface and the relationships window.

The second part of the WHERE clause (**`AND RepID = 9`**) is simply a standard selection criterion.

12.3.4 SQL as a data definition language

A distinction is often made between two types of database language constructs: data definition language (DDL) and data manipulation language (DML). So far, we have used QBE and SQL for data manipulation (joining, sorting, etc.). For data definition tasks (e.g., creating tables), we have used ACCESS's table design form. However, like most databases, ACCESS support the use of SQL as a DDL.

10 Open a new SQL query and type the following:

```
NL CREATE TABLE Suppliers
NL (SuppID INTEGER NOT NULL,
NL SuppName VARCHAR(20),
NL Phone VARCHAR(14),
NL CONSTRAINT PK_Suppliers PRIMARY KEY
   (SuppID));
```

Rather than using the ACCESS-specific data types Long and Text, the query uses standard SQL data types “Integer” and “Varchar” respectively. Fortunately, ACCESS is smart enough to map the standard SQL data types to its own internal data types.

SQL is (nominally) a vendor-independent standard. As a consequence, the DDL SQL statement in Figure 12.4 will execute correctly on ORACLE, MICROSOFT SQL SERVER, or any other SQL-compliant DBMS. This would not be the case if ACCESS-specific data types were used.

11 Select **Query** → **Run** from the main menu or press the exclamation mark (**!**) on the toolbar.

12 Switch to the **Tables** tab of the database window and open the new `Suppliers` table in design view.

As Figure 12.4, shows, the SQL statement created the table, specified the field names and data types, and set the primary key.

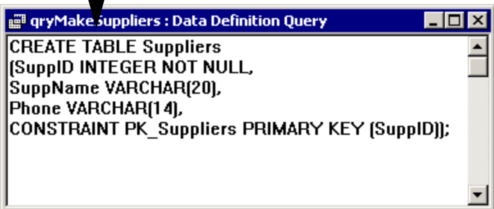
12.4 Discussion

Although the syntax of SQL is not particularly difficult, writing long SQL queries is tedious and error-prone. For this reason, you are advised to use QBE for your project.

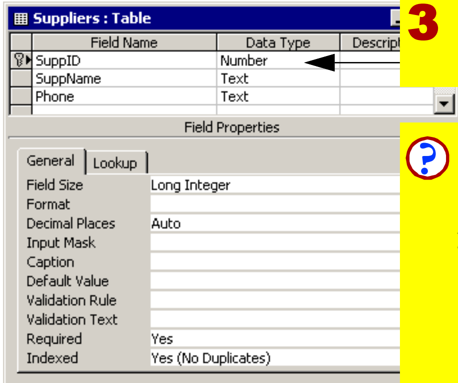
When you say you know something about databases, it usually implies you know the DDL and DML aspects of SQL in your sleep. If you plan to pursue a career in information systems, a comprehensive SQL reference book and lots of practice can be worthwhile investments.

FIGURE 12.4: Using SQL as a data definition language.

1 Write a data definition SQL statement to create a list of suppliers.



2 Execute the SQL statement by selecting **Query** → **Run** from the main menu.



3 Examine the resulting table definition.

Note that SQL does not support a number of the ACCESS-specific field properties (such as input masks). These must be created using the table design form.

13.1 Introduction: Using forms as the core of an application

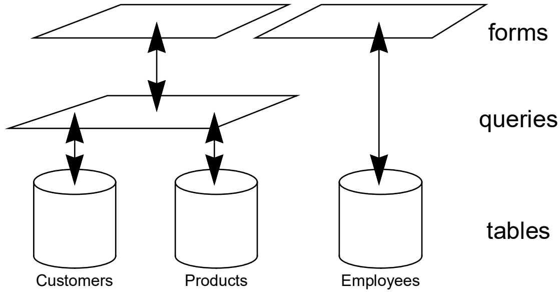
Forms provide a user-oriented interface to the data in a database application. Moreover, forms permit you, as a developer, to

- specify in detail the appearance and behavior of the data on screen, and
- exert a certain amount of control over the user's ability to modify the data.

Like queries, forms do not contain any data. Instead, they provide a "lens" through which tables and queries can be viewed. The lens metaphor for describing the interaction between tables, queries, and forms is shown in Figure 13.1.

In this lesson, we are going to explore the basic elements of form creation using ACCESS' form design tools.

FIGURE 13.1: The relationship between forms, queries, and tables.



13.2 Learning objectives

- create a simple form
- use the properties of an object to make its contents read-only
- understand the difference between a "bound" and "unbound" text box
- create a form using the form wizard
- understand the difference between a "columnar" (single-column) and "tabular" form

13.3 Exercises

13.3.1 Creating a form from scratch

Although ACCESS provides an excellent wizard for creating simple forms, you will start by building a form from scratch. Working without the wizard will give you a better appreciation for what it is that the wizard actually does and provide you with the basic knowledge needed to customize and refine the wizard's output.

Show me (lesson13-1.avi)

1 Create a new blank form based on the **`Customers`** table, as shown in Figure 13.2.

The basic elements of the form design screen are shown in Figure 13.3.

2 Use the **View** menu to display the **toolbox** and **field list** if they are not already visible (see Figure 13.3).

13.3.1.1 Adding bound text boxes

Show me (lesson13-2.avi)

3 Add a "bound" text box for the **`CustID`** field by dragging **`CustID`** from the field list to the form background, as shown in Figure 13.4.

FIGURE 13.2: Create a new form to display data from the ***`Customers`*** table.

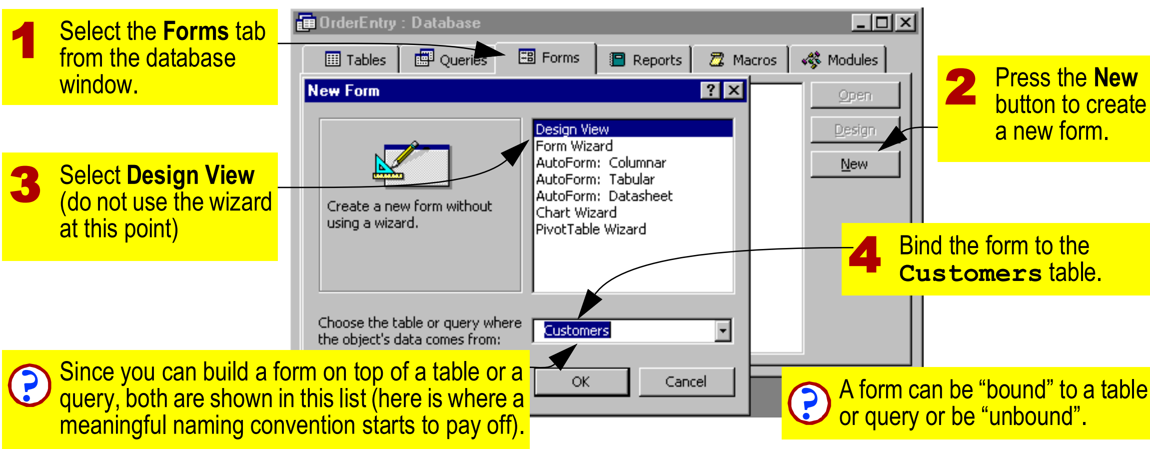
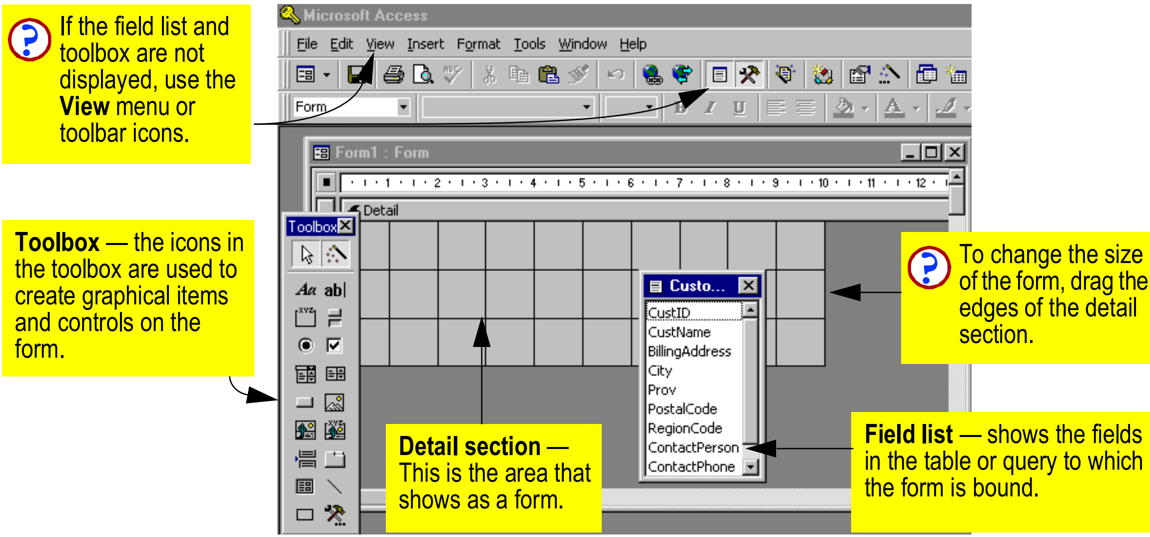


FIGURE 13.3: The basic elements of the form design screen.



4 Select the `CustID` text box using the mouse and reposition it in the upper left region of the form.

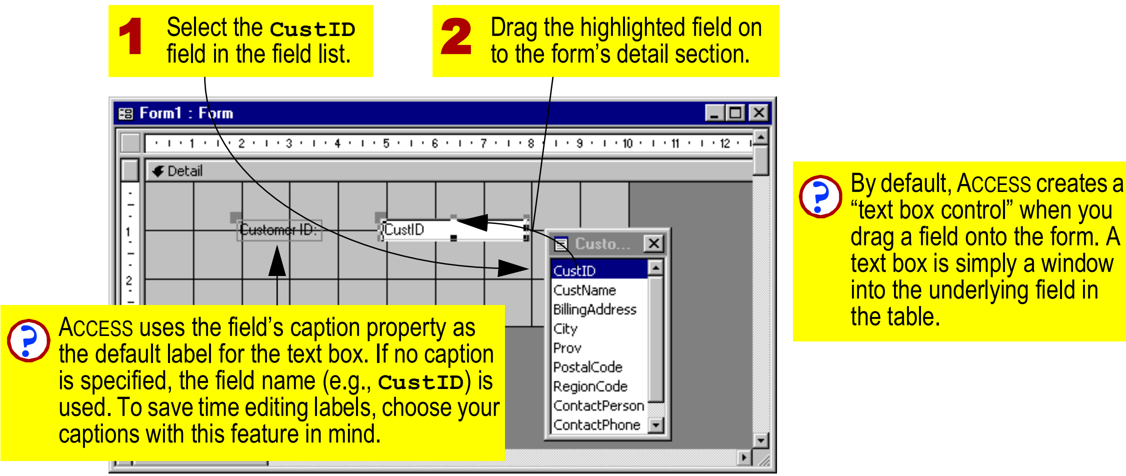
Remember that you can always use the “undo” feature to reverse mistakes. Select **Edit** → **Undo** from the menu or simply press **Ctrl-Z** (this works the same in virtually all WINDOWS applications).

5 Drag the remaining fields on to the form (do not worry about whether the fields are lined up perfectly).

6 Save the form as **`frmCustomers`**.

7 Select **View** → **Form** to see the resulting form. Alternatively, press the form view icon ( ).

FIGURE 13.4: Create a bound text box for the `CustID` field.



8 Select **View** → **Form Design** or press the design view icon ( ) to return to design mode.

13.3.1.2 Using a field's properties to protect its contents

Every object on an ACCESS form (e.g., text box, label, detail section, etc.) has a set of properties that can be modified. In this section, you are going to use the **Locked** and **Enabled** properties to control the user's ability to change the information in a field.

Show me (lesson13-3.avi)

9 Select the `CustName` text box and right-click to bring up its property sheet, as shown in Figure 13.5.

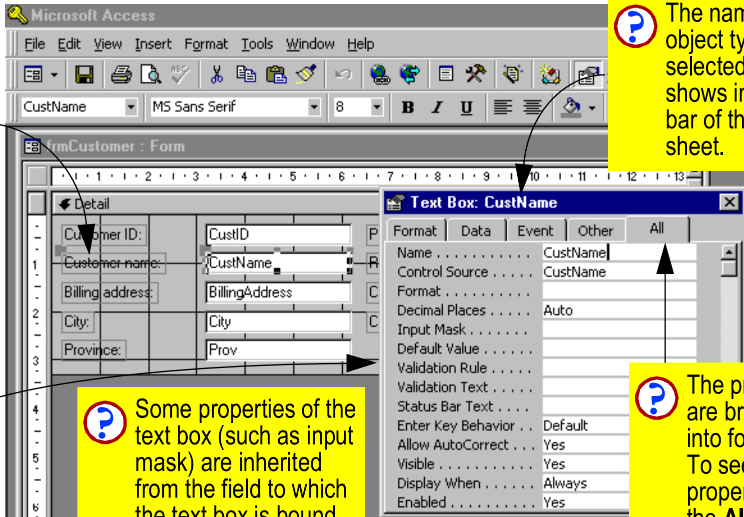
FIGURE 13.5: Bring up the property sheet for the **`CustName`** text box.

1 Select the object (e.g., the **`CustName`** text box) for which you wish to see the properties.

When an object has been selected, it is bordered by eight dark “sizing handles”.

2 Right-click once on the selected object to get the context menu.

3 Select **Properties** to get the property sheet for the object.



The name and object type of the selected object shows in the title bar of the property sheet.

Some properties of the text box (such as input mask) are inherited from the field to which the text box is bound.

The properties are broken down into four groups. To see all the properties, select the **All** tab.

10 Scroll down the property sheet to the **Locked** property and set it to Yes, as shown in Figure 13.6.

11 Switch to the form view and attempt to change the contents of the **`CustName`** field. The contents of the field are indeed locked.

A stronger form of protection than locking a field is “disabling” it.

12 Return to design mode and make the following changes: reset the **Locked** property to No and set the **Enabled** property to No.

FIGURE 13.6: Change the **Locked** property of ***CustID*** to Yes.

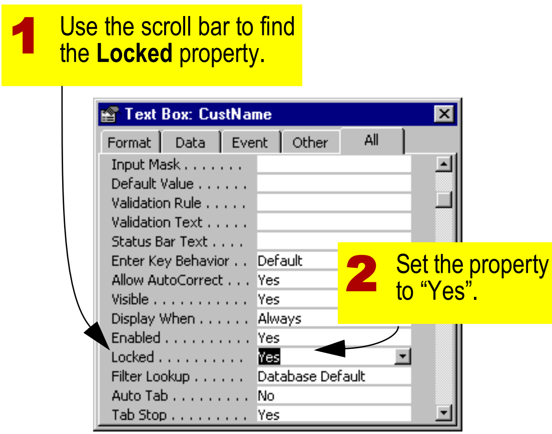
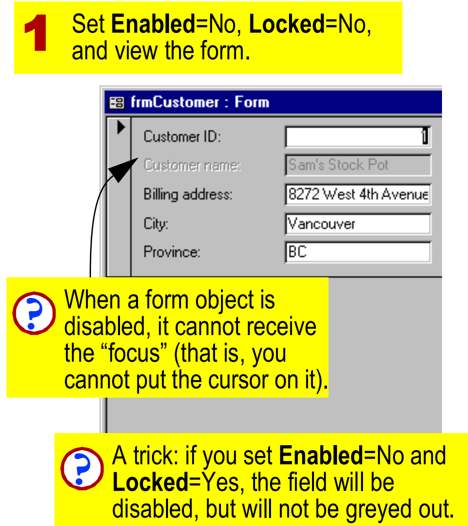


FIGURE 13.7: Disable ***CustName*** and attempt to change the value in the field.



13 Attempt to change the contents of the **CustName** field in form view, as shown in Figure 13.7.

13.3.1.3 Adding an unbound text box

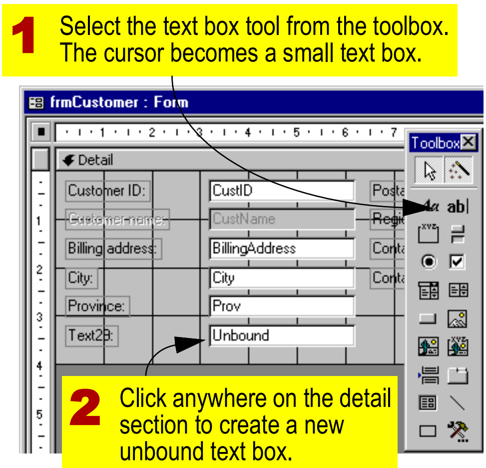
All the text boxes created in the previous section were "bound" text boxes—that is, they were bound to a field in the underlying table or query. When you change the value in a bound text box, you are making the change directly to the data in the underlying table.

It is possible, however, to create objects on forms that are not bound to anything. Although you will not use many "unbound" text boxes in your kitchen supply project, it is instructive to see how they work.

Show me (lesson13-4.avi)

14 Select the text box tool (abl) from the toolbox and create an unbound text box, as shown in Figure 13.8.

FIGURE 13.8: Create an unbound text box.



15 Switch to form view and enter a value in the new unbound text box.

16 Use the record navigation buttons to step through the different customers. Notice that unlike the bound text boxes for `CustID`, `BillingAddress`, and so on, the value in the unbound text box does not change.

13.3.1.4 Binding an unbound text box to a field

The only difference between a bound and an unbound text box is that the **Control Source** property of a bound text box is set to the name of a field. In this section, you are going to change the unbound text box shown in Figure 13.8 to a bound text box.

17 Bring up the property sheet for the unbound text box. Change its **Control Source** property from NULL to `CustID`, as shown in Figure 13.9.

13.3.2 Creating a single-column form using the wizard

Now that you understand the basics of creating and modifying bound text boxes, you can rely on the form wizard to create the basic layout of all your forms.

Show me (lesson13-5.avi)

18 Create a new form bound to the `Products` table using the form wizard, as shown in Figure 13.10.

FIGURE 13.9: Set the **Control Source** property of an unbound text box.

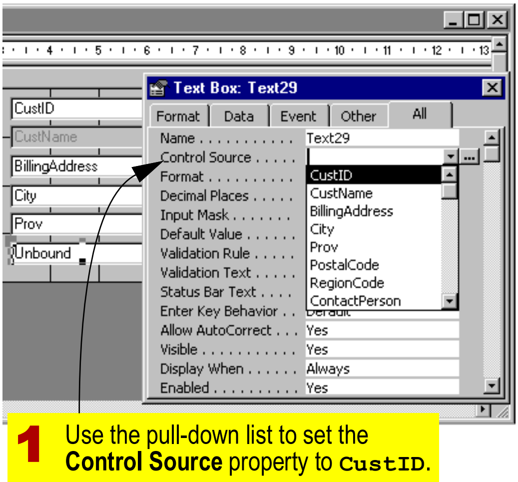
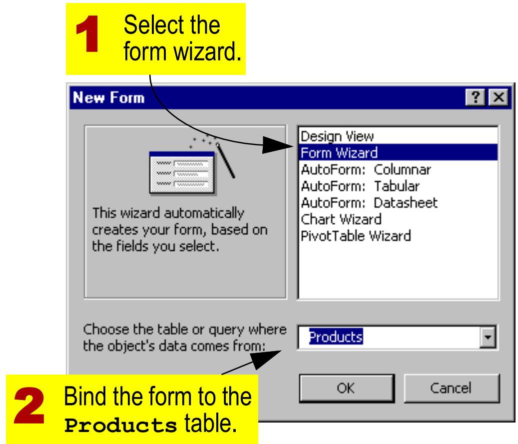


FIGURE 13.10: Create a new form using the form wizard.



19 Use the form wizard to specify the fields you want on your form and the order in which they appear. Select "columnar" when prompted for the form type.

"Columnar" forms are called "single column" forms in version 2.0.

The primary advantage of the wizard is that it automatically creates, formats, and aligns the bound text boxes. Of course, once the wizard has created a form, you are free to modify it in any way.

If you make a major layout error when creating a form (e.g., you put the fields in the wrong order) it is often easier to start over using the wizard than to fix the problem manually.

13.4 Discussion

13.4.1 Columnar versus tabular versus datasheet forms

Columnar forms show one record per page. **Tabular** forms, in contrast, show many records per page and are used primarily as subforms (subforms are discussed in Lesson 14). There is also a a **datasheet** form type, but it is seldom used since it gives the developer relatively little control over the look and behavior of the data. The three different types of forms are shown in Figure 13.11.

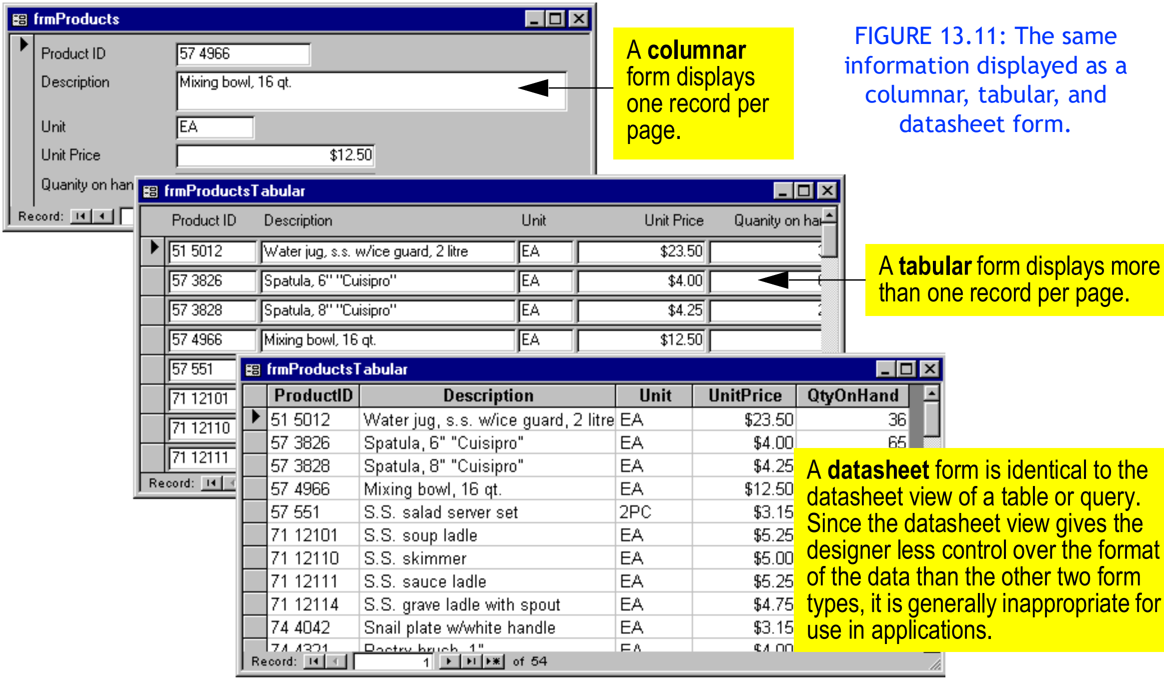


FIGURE 13.11: The same information displayed as a columnar, tabular, and datasheet form.

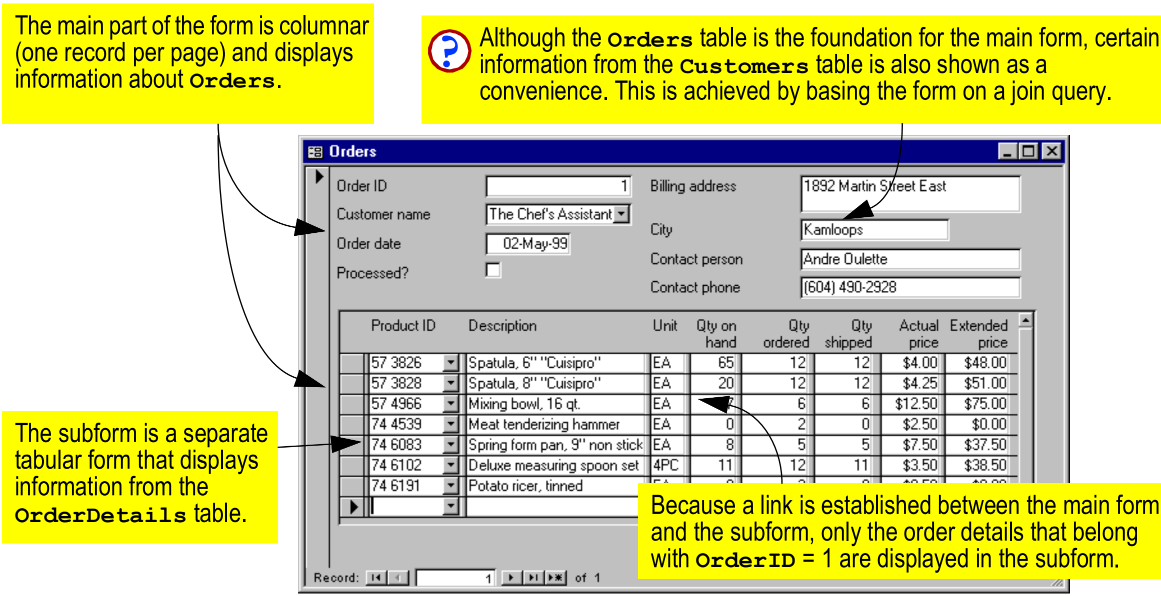
13.5 Application to the project

20 Use the wizard to create columnar forms for all your *master tables*. Note that in some cases (e.g., `Customers`) you will want to base the form on a join query rather than table in order to show information such as the name of the region.

14.1 Introduction: The advantages of forms within forms

A columnar (single-column) main form with a tabular subform is a natural way of displaying data from tables that participate in a one-to-many relationship. For example, the form shown in Figure 14.1 is really two forms: the **main form** contains information about a specific order; the **subform** shows all the order details associated with the order.

FIGURE 14.1: A typical form/subform combination.



In the case of an order form with an order details subform, the `OrderID` field provides a link between the two forms. The connection through `OrderID` allows ACCESS to synchronize the forms, meaning:

- when you move to another order, only the order details associated with the currently visible order are shown in the subform;
- when you add a new order detail, the foreign key in the `OrderDetails` table is automatically filled in (in fact, there is no need to even show `OrderID` in the subform).

Show me (lesson14-1.avi)

Although you will quickly learn to take a feature such as form/subform synchronization for granted, it is worthwhile to consider what this feature does and what it would take if you had to implement the same feature yourself using a programming language.

14.2 Learning objectives

- understand form/subform synchronization
- create a form/subform combination
- manually link a subform to its main form

14.3 Exercises

Although there are a number of different ways to create a subform within a main form, the recommended procedure is the following:

1. create and save two forms—a columnar main form and a tabular subform (recall the distinction between columnar and tabular forms from Figure 13.11);
2. drag the subform on to the main form; and,
3. verify the linkage between the two forms.

14.3.1 Creating the main form

1 If you did not do so in Section 10.5, create a query that joins the `Orders` and `Customers` tables and save it as `qryOrders`.

Show me (lesson14-2.avi)

Since the purpose of the `qryOrders` query is to facilitate the population of the `Orders` table, ensure you understand the implications of Section 10.4.2. Specifically, ensure that all the field from `Orders` are projected into the query.

2 Use the wizard to create a columnar form based on the `qryOrders` query.

Show me (lesson14-3.avi)

3 When you get to the wizard step shown in Figure 14.2, make sure you select the "by Orders" option.

The form wizard recognizes the one-to-many relationship between `Customers` and `Orders` and offers to create a subform for you automatically. We prefer to create the subform ourselves.

FIGURE 14.2: Prevent the form wizard from creating a customer/order subform.

An annoying problem that occurs in ACCESS version 8 is that the form wizard builds the form on an "ad hoc" query, rather than the saved query you specify. For example, instead of binding the form to `qryOrders` (as specified in Step 1), the wizard sets the form's **Record Source** property to an embedded SQL statement, as shown in Figure 14.3.

4 Verify that the **Record Source** property for your new form is correct. If not, delete the embedded SQL statement and bind the form to the appropriate saved query.

If the form is bound to an embedded SQL query, any changes you make to `qryOrders` (such as sorting by `OrderDate`) will not be reflected in the form.

5 Rearrange the fields so that they make efficient use of the top part of the form, as shown in Figure 14.1.

6 Save the form as `frmOrders`.

14.3.2 Creating the subform

Once the main form is created, you create a second, tabular form using essentially the same procedure. A subform is like any other form, except that it will ultimately be nested within a main form.

Since you will want to show information about products (such as quantity on hand and description) when you add order details, the subform should be bound to the **`qryOrderDetails`** query that you completed in Section 11.5.

If you use the wizard to create a form based on a query and then make significant changes to the query (e.g,. add or delete columns), you will also have to make changes to the form. For this reason, it is good practice to test your queries and make sure you are 100-percent happy with them before creating your forms.

7 Use the wizard to create a tabular subform based on **`qryOrderDetails`**, as shown in Figure 14.4 and Figure 14.5

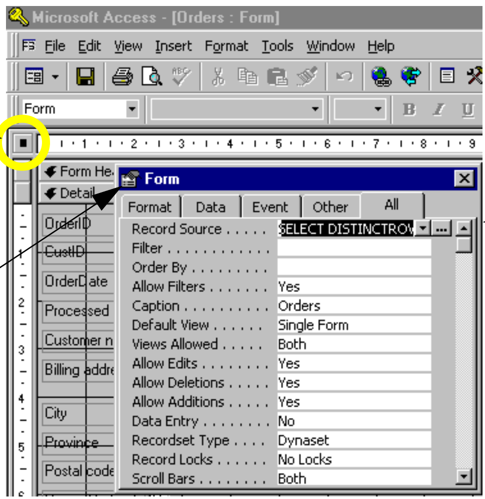
Show me (lesson14-4.avi)

8 Save the form as **`sfrmOrderDetails`**.

FIGURE 14.3: Set the form's **Record Source** property to the desired saved query.

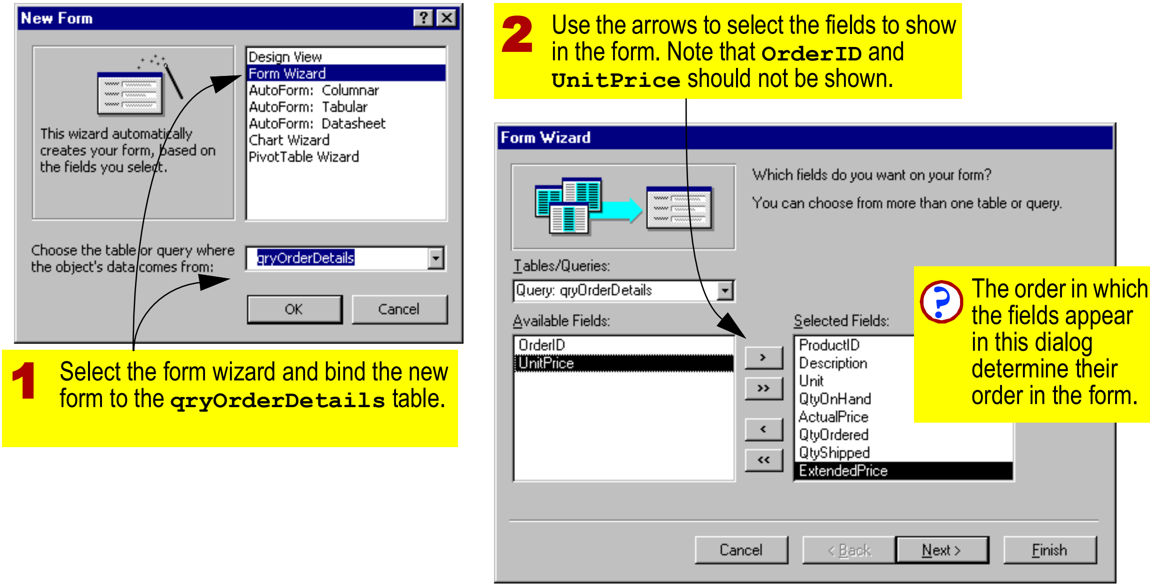
1 Click on the intersection of the vertical and horizontal rules to select the form as a whole.

2 Right-click to bring up the properties sheet for the form (the name of the selected object shows in the title bar of the properties sheet).



3 Note the embedded SQL statement. Use the pull-down list to set the **Record Source** property to **`qryOrders`**.

FIGURE 14.4: Use the wizard to create the ***OrderDetails*** subform (part 1).

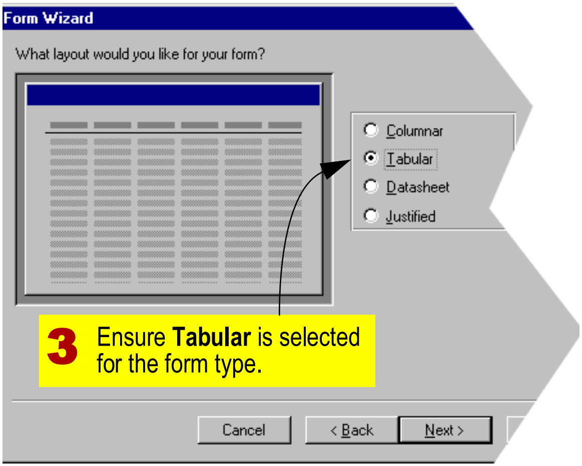


Subforms created by the wizard typically require some fine tuning in order to reduce the amount of space they occupy. A number of editing issues are highlighted in Figure 14.6.

9 Switch to design mode and rearrange the fields on your subform as required. You should not waste too much time on this.

10 Set the **Enabled** property to False for all the fields that the user should not be changing during order entry. For example, there is no reason for the user to change the `Description` or `QtyOnHand` fields when entering order details.

FIGURE 14.5: Use the wizard to create the ***OrderDetails*** subform (part 2).



Remember, if you also change the **Locked** property to True, the coloring of the text box will be normal instead of grayed-out.

14.3.3 Linking the main form and subform

To create a subform, you drag and drop the icon for the tabular form on to the columnar form.

Show me (lesson14-5.avi)

11 Open the main form (`frmOrders`) in design mode.

12 Use the **Window** menu to bring the database window into the foreground. Alternatively, you can press the database window icon ( ) on the tool bar.

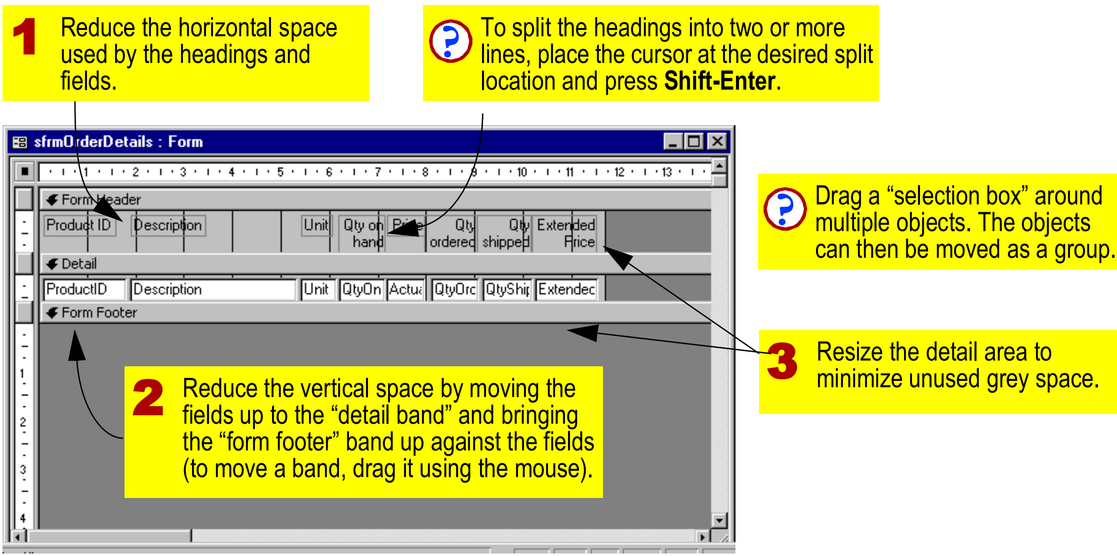
13 Perform the steps shown in Figure 14.7 to drag the subform on to the main form.

14 The result of the drag-and-drop operation is shown in Figure 14.8.

In ACCESS 2000, the blank subform control is replaced by a live window on to the design view of the subform. This feature permits you to edit both your main form and your subform(s) at the same time.

The advantage of the drag-and-drop method of creating a sub form is that the width of the subform control (the white window) is automatically set to equal the width of the tabular subform. Naturally, if you change the size of the tabular subform later on, you will have to manually adjust the size of the subform control on the main form (or delete the subform control and repeat the drag-and-drop procedure).

FIGURE 14.6: Edit the subform to reduce the amount of space it uses.

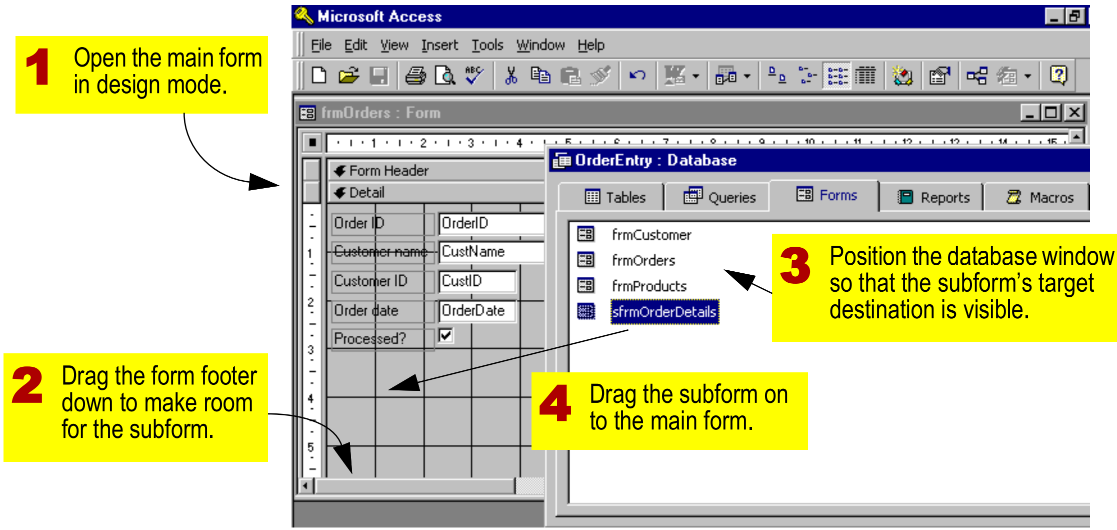


14.3.4 Linking forms and subforms manually

If both the form and the subform are based on tables, and if relationships have been declared between the tables, ACCESS normally has no problem determining which fields "link" the information on the main form with the information in the subform. However, when the forms are built on queries, ACCESS has no relationship information to rely on. As such, you have to specify the form/subform links manually.

Since both the forms created in Section 14.3.3 were built on queries, ACCESS was unable to automatically determine the linking fields.

FIGURE 14.7: Drag the subform on to the main form.

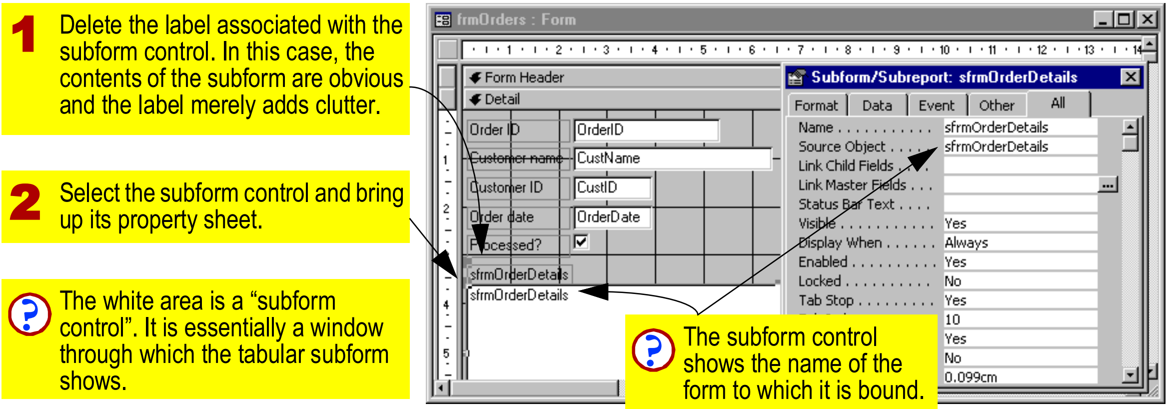


15 Verify the link between the form and the subform by examining the property sheet of the subform control, as shown in Figure 14.8.

If the link fields are incorrect or empty (as they are in this case), you can do one of two things:

1. Type in the name of the link field(s) manually. If more than one field is required to define the link, separate the field names with semicolons, for example: `DeptName; CourseNo.`
2. Use the builder provided in version 7 and greater to have ACCESS make a guess at how the main form and subform are related.

FIGURE 14.8: The drag-and-drop operation creates a subform control.



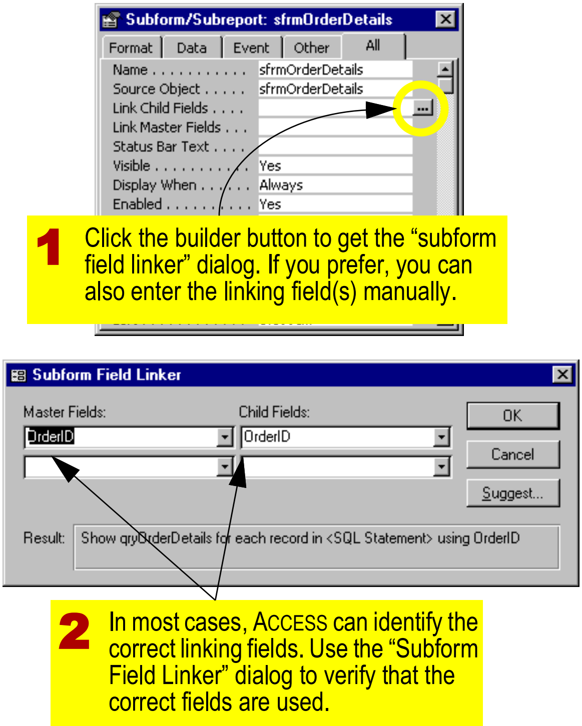
The terms "link child field" and "link master field" are specific to the form design tool in MICROSOFT ACCESS. However, you should recognize that "link child field" and "link master field" are identical to "foreign key" and "primary key" respectively. The main form is the master ("one" side) and the subform is the child ("many" side).

16 Use the builder to specify the link fields between the main form (`frmOrders`) and the subform (`sfrmOrderDetails`). This is shown in Figure 14.9.

Version 2.0 does not have a builder for linking fields.

17 View the resulting form. Notice that as you move from order to order, the number of order details shown in the subform changes. This indicates that form/subform synchronization is working.

FIGURE 14.9: Set the link fields for the form/subform.



14.3.5 Non-synchronized forms

In this section, you will delete the link fields created in Figure 14.9 in order to explore some of the problems associated with non-synchronized forms.

Failure to verify and set link fields is very common mistake for neophyte form designers. As you will see, however, unsynchronized forms can cause all kinds of trouble when it comes time to enter data. *Always* verify your links when creating a subform.

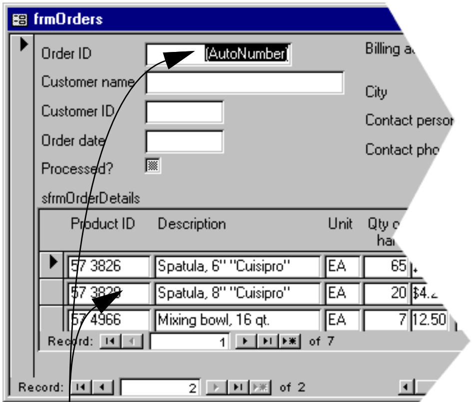
Show me (lesson14-6.avi)

18 Return to form design mode, bring up the subform’s property sheet. Delete the link fields (highlight the text and press the **Delete** key).

19 View the form. When you attempt to add a new order (see Figure 14.10) all order details (not just those associated with a particular order) still show in the subform.

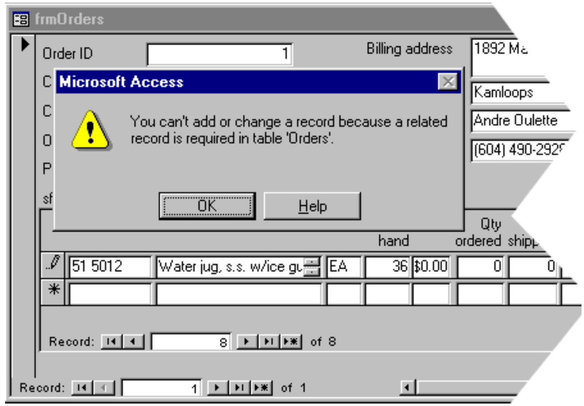
20 Use the record selector buttons at the bottom of the form to return to the first order and attempt to add a new order detail (use `ProductID` = “51 5012”) to the bottom of the list. This is shown in Figure 14.11.

FIGURE 14.10: Adding a new order using an unsynchronized form/subform.



Although the order is new (the AutoNumber has not even been set yet) order details associated with other orders show in the subform.

FIGURE 14.11: Adding a new order using an unsynchronized form/subform.



Because the forms are not synchronized, ACCESS cannot automatically set the `OrderID` of the new order detail to that of the current order (in this case 1). By default, `OrderDetails.OrderID` is equal to zero, hence the referential integrity error in Figure 14.11.

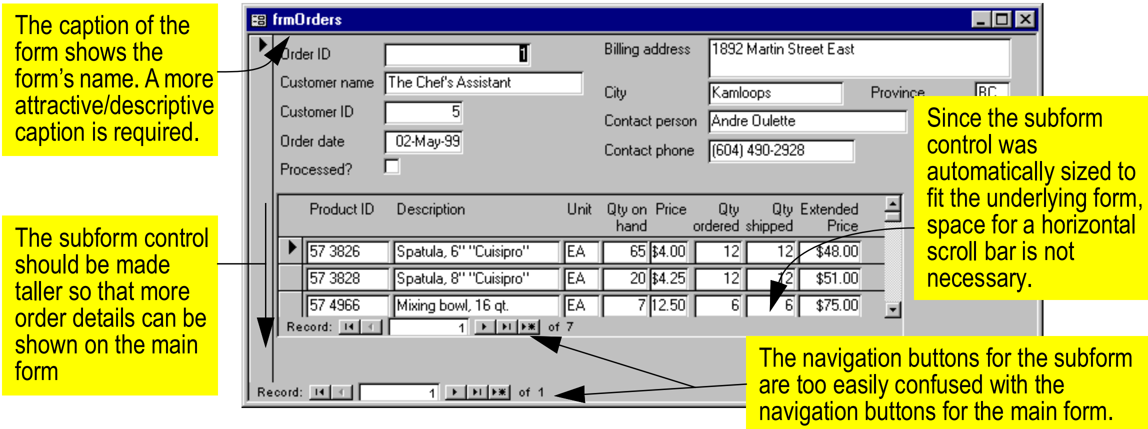
21 Return to design view, re-establish the correct link fields, and save the form.

14.3.6 Aesthetic refinements

In this section, you will modify the properties of several form objects (including the properties of the form itself) to make your form more attractive and easier to use.

In Figure 14.12, the basic form created in the previous sections is shown and a number of shortcomings are identified.

FIGURE 14.12: A form/subform in need of some basic aesthetic refinements.



14.3.6.1 Changing the form's caption

22 In design mode, click on the point at which the horizontal and vertical rulers intersect, as shown in Figure 14.13. This selects the form object.

23 Change form's **Caption** property to "Order Entry" or something similar.

14.3.6.2 Eliminating unwanted scroll bars and navigation buttons

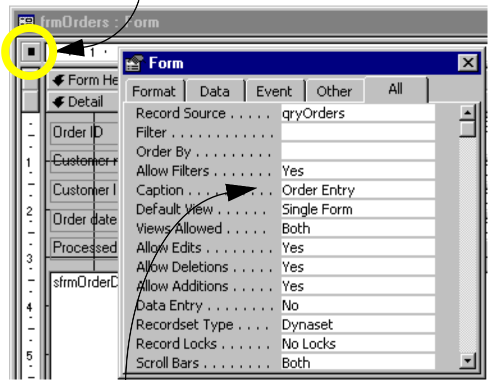
Scroll bars and navigation buttons are also form-level properties. However, in this case, you need to modify the properties of the subform rather than the main form.

Show me (lesson14-7.avi)

24 To quickly open the subform in design mode, double-click the subform control

FIGURE 14.13: Change the form's caption.

1 Click on the square where the vertical and horizontal rulers intersect and right-click to bring up the property sheet for the form.



2 Set the **Caption** property to "Order Entry" (or whatever text you want showing in the title bar of your form).

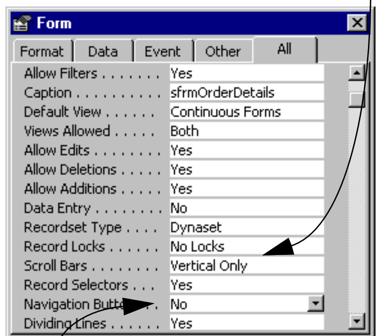
when viewing the main form in design mode (this takes some practice).

In ACCESS 2000, you can edit the subform directly through the subform control. However, this assumes the subform control's window is large enough that you can see what you are doing. If you prefer to work in a full-sized window, you can use the database window and open the subform in design view in the normal manner.

25 Bring up the property sheet for the form and scroll down to change its **Scroll Bars** and **Navigation Buttons** properties, as shown in Figure 14.14.

FIGURE 14.14: Change the scroll bars and navigation buttons of the subform.

1 Set the **Scroll Bar** property to "Vertical Only".



2 Set the **Navigation Buttons** property to "No".

The net result, as shown in Figure 14.1, is a more attractive, less cluttered form.

14.4 Discussion

14.4.1 Add, edit, and view modes

One problem with the order form you have created is that there is nothing to stop users from making changes to the order information once the order has been entered and processed. Clearly, the ability to change the order once the physical goods have been shipped and a paper copy of the invoice has been sent to the customer is dangerous.

This is a common problem: you need to give users the ability to add and edit information when entering transactions. But once the transaction is processed, the information should be cast in stone.[1] There are two basic approaches to achieving this type of functionality:

1. **Two forms** — You can create two forms for your users: one for adding new orders and one for viewing orders that have already been placed. Clearly, the latter form should be read-only.
2. **A single "smart" form** — If you know something about event-driven programming, you can create a single form that behaves differently depending on whether the transaction showing on the screen has been processed. The advantage of this approach is that you do not have to worry about maintaining two forms.

In either approach, control over the users' interaction with the data can be controlled by setting some simple form properties.

14.4.2 Form properties for controlling user access

If you examine the property sheet in Figure 14.14, you will see four form-level properties that can be used to control what the user can and cannot do using the form:

1. **Allow Edits**: this is similar to the **Locked** property for fields except that it applies to the form as a whole.

[1] This is especially true for financial systems in which every transaction must be stored "as is". If a mistake is made in such systems, the transaction cannot be deleted or changed. Instead, the incorrect transaction must be flagged as void, an offsetting transaction must be made to "undo" the error, and the correct transaction must be re-entered. Any other approach would be impossible to audit.

2. **Allow Deletions**: when this property is set to No, a user cannot use the form to delete a record once it is saved.
3. **Allow Additions**: when this property is set to No, users cannot use the form to add new records.
4. **Data Entry**: when this property is set to Yes, the form opens to a new record and prevents the user from moving back to previously saved records. That is, the user can add and edit records created in that particular session. But once the form is closed, all saved records are inaccessible.

Like most properties in ACCESS, these properties can be set via a programming language while the form is in use. As such, it is easy to envision an event-driven programming scenario in which the user presses a button to process an order and, at the same time, the **Allow Edits** property is set to No. You will learn more about event-driven programming in Lesson 19.

14.5 Application to the project

26 Complete the order form you started in this lesson. Since the form is the foundation of your application, you should ensure that basic features (such as form/subform synchronization) are working properly before you continue.

The form you created in this lesson is not quite ready for entering orders. In the remaining tutorials, you will be enhancing its functionality with bound controls and adding a feature to automatically update inventory when the order is complete. For now, the form is only really useful for viewing any orders you have added manually.

15.1 Introduction: What is a combo box?

So far, the only kind of user interface control you have used on your forms has been the text box. However, ACCESS provides other controls (such as combo boxes, list boxes, check boxes, radio buttons, and so on) that can be used to improve the attractiveness and functionality of your forms. These interface elements are called bound controls because they are bound to fields in the underlying table. When you change data in a bound control, you are changing the data in the underlying table.

In this lesson, we are going to focus on a particularly useful bound control: the combo box. A combo box is list of values from which users can select a single value. Not only does selecting from a list save typing, it can be used as a means of enforcing referential integrity since the user's choices are (typically) limited to the values in the list.

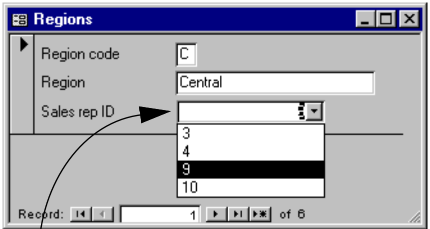
The term "combo box" is a MICROSOFT-ism that is slowly working its way into standard parlance. Synonyms include drop-down list, select list, pick list, and so on.

Figure 15.1 shows a combo box used to assign sales reps to regions. Since users are limited to the choices in the combo box, there is no danger of entering a non-existent employee ID or the employee ID of someone who is not in sales (assuming that the list provided by the combo box is correct).

Although advanced controls such as combo boxes and list boxes look and behave very differently than simple text boxes, their function is ultimately the same. For example, the very basic combo box in Figure 15.1 is bound to the `Regions.RepID` field. When a value in the combo box is selected, it is copied into the underlying field exactly as if the user had typed the value (3, 9, 10, etc.) into a text box.

It is important to realize that combo boxes have no intrinsic *search* capability. Combo boxes change values; they do not automatically move to the record with the value you select. For example, selecting `RepID` = 9 does not move to a region serviced by that employee (after all, there may be more than one region). Of course, it is possible to use combo boxes for search, but implementing this

FIGURE 15.1: A combo box for filling in the ***RepID*** field.



Instead of relying on the user to select a valid value for **`RepID`**, a combo box is used to show the **`EMP_ID`** values of all the employees in the sales group. When the user selects "9", the value is inserted into the **`RepID`** field of the **`Regions`** table.

functionality requires a small amount of programming.

15.2 Learning objectives

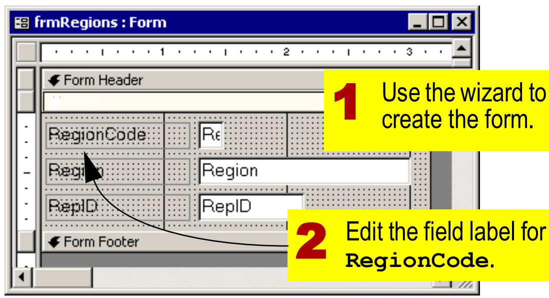
- create a bound combo box
- create a combo box that displays values from a different table
- show additional information in a combo box
- prevent certain information from showing in the combo box
- change the order in which the items appear in a combo box
- understand and change tab order
- know whether to put a combo box on a key field

15.3 Exercises

In the following exercises, you will create a basic "regions" form using the wizard and replace the **`RepID`** textbox with combo boxes of increasing complexity.

1 Use the form wizard to create a new columnar form called **`frmRegions`**, as shown in Figure 15.2.

FIGURE 15.2: Create a simple columnar form for the ***Regions*** table.



Since you did not set a suitable caption for the **RegionCode** field when you created in Section 5.3.5, the wizard uses the field name at the label for the textbox.

2 Change the field label for the **RegionCode** field to “Region code” (or whatever you think is appropriate).

3 Set the **Caption** property of the form to “Regions” (review Section 14.3.6.1 as required).

4 Ensure the toolbox and field list are visible (review Figure 13.3 as required).

15.3.1 Creating a combo box manually

Although ACCESS has a wizard that simplifies the process of creating combo boxes, you will start by building a simple combo box (similar to the one shown in Figure 15.1) with the wizard turned off. This will give you a better appreciation for what the wizard does and provide you with the skills to make refinements to wizard-created controls.

15.3.1.1 Adding the control to the form

Show me (lesson15-1.avi)

5 Delete the existing **RepID** text box by selecting it and pressing the **Delete** key.

The wizard toggle button ( ) in the toolbox allows you to turn wizard support on and off.

6 Ensure the button is out (wizards are turned off).

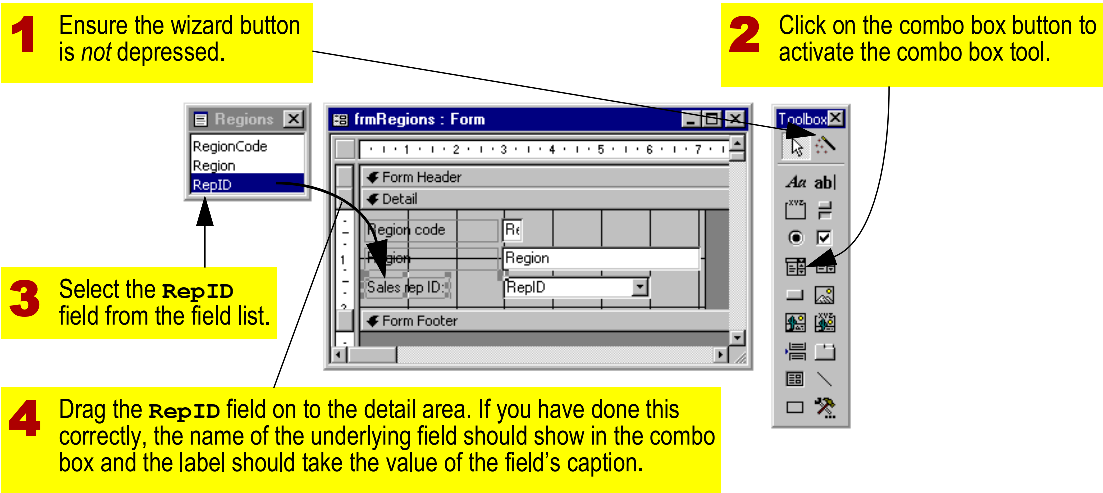
7 Click on the combo box tool ( ). The cursor turns into a small combo box.

8 With the combo box tool selected, drag the **RepID** field from the field list to the desired location on the form’s detail section, as shown in Figure 15.3.

The process of selecting a tool from the toolbox, and then using the tool to drag a field from the field list ensures that the control you create is bound to a field in the underlying table or query.

If you forget to drag the field in from the field list, you will create an unbound combo box, as shown in Figure 15.4. If you accidently create an unbound combo box, the easiest thing to do is to delete it and try again.

FIGURE 15.3: Create a bound combo box without using the wizard.



15.3.1.2 Filling in the combo box properties

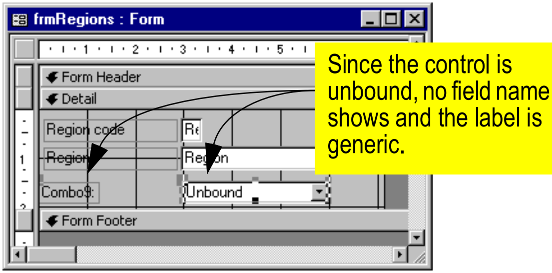
9 Switch to form view and test the combo box that you just created.

In this section, you will tell ACCESS what you want to appear in the rows of the new combo box.

At this point, the combo box does not show any list items because we have not specified what the list items should be. There are three methods of specifying what shows up in the combo box list:

1. Enter a list of values into the combo box's **Row Source** property;
2. Tell ACCESS to get the value from an existing table or query;
3. Tell ACCESS to use the names of fields in an existing table (you will not use this approach).

FIGURE 15.4: An unbound combo box (not what you want).



Although the second method is the most powerful and flexible, we will start with the first.

10 Switch to form view and test the combo box that you just created.

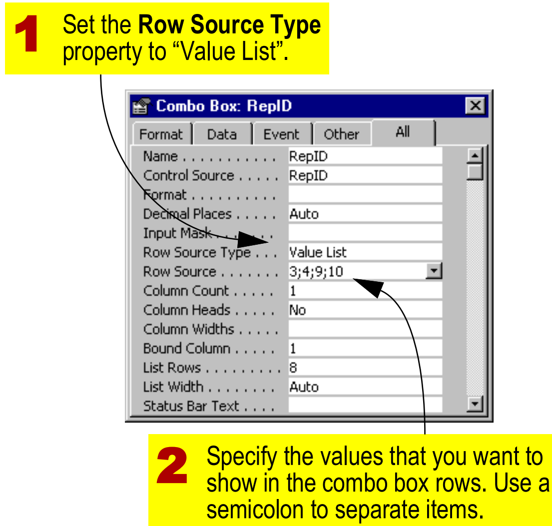
Show me (lesson15-2.avi)

11 Switch to form view and test the combo box that you just created.

12 Bring up the property sheet for the `RepID` combo box.

13 Change the **Row Source Type** property to Value List. This tells ACCESS to expect a list of values in its **Row Source** property.

FIGURE 15.5: Set the **Row Source Type** and **Row Source** properties for the combo box.



14 Enter the following values into the **Row Source** property, as shown in Figure 15.5: `3;4;9;10`. These values correspond to the employee IDs of employees in the sales group.

15 Set the **Limit To List** property to Yes.

If the **Limit To List** property is set to No, the user can ignore the choices in the combo box and simply type in a value (e.g., “53”). In this particular situation, you want to limit the user to the four choices given.

16 Switch to form view and experiment with the combo box.

Notice that the combo box has some useful built-in features. For example, if you choose to type values rather than select them with a mouse, the combo box anticipates your choice based on the letters you type. Thus, to select “10”, you need only type “1”.

15.3.2 A combo box based on another table or query

An obvious limitation of the value-list method of creating combo boxes is that it is impossible to change or update the items that appear in the list without finding and modifying the **Row Source** property. If you have many forms that use combo boxes based on `EMP_ID`, the result is a maintenance nightmare.

A more elegant and flexible method of populating the rows of a combo box is to have ACCESS create the list of items in the combo box dynamically using an existing table or query.

15.3.2.1 Preparing the source data

Since you only want sales employees to show up in this combo box, you need to create a query that provides this information before continuing with the combo box.

If you saved the `qryEmployees` query you created in Section 11.3.1, you can use it as the basis for the `qrySalespeople` query. The procedure below assumes that you are creating the query from scratch.

17 Switch to the database window and create a new query called `qrySalespeople`.

18 Project the asterisk and the `EMP_JOB_CL` field. Uncheck the **Show** box for `EMP_JOB_CL`, as shown in Figure 15.6.

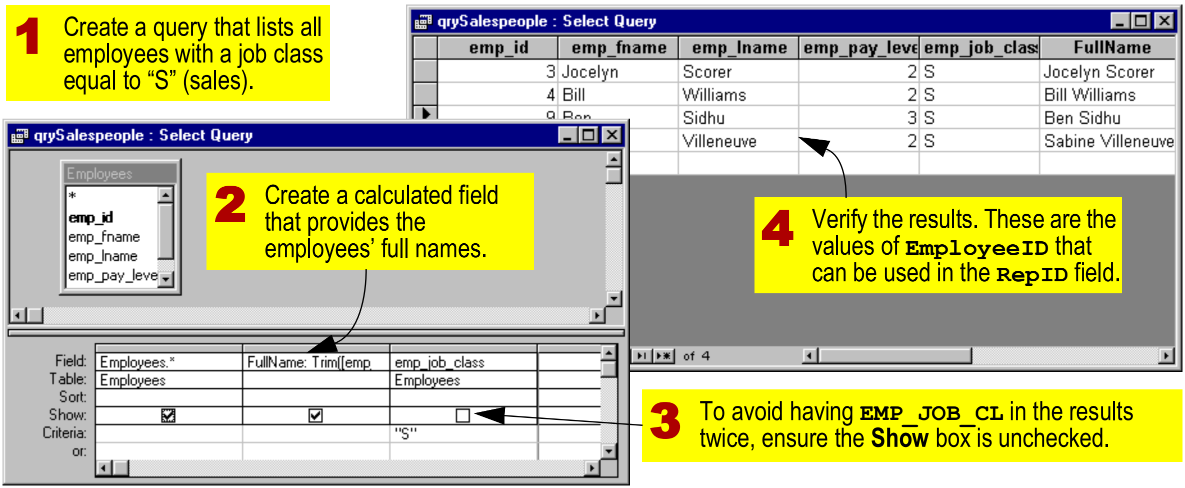
You project the `EMP_JOB_CL` field into the query so that you can filter out non-sales employees. However, since you have also projected the asterisk, you must ensure that the **Show** box under the `EMP_JOB_CL` column is *unchecked*. Otherwise, `EMP_JOB_CL` will appear in the query’s results set twice.

19 As a criteria in the **EMP_JOB_CL** field, enter “S” (“S” is the job class for salespeople).

20 Create a calculated field called **FullName** that concatenates the employees’ first and last names (review Section 11.3.1 as required).

The resulting query and results are shown in Figure 15.6.

FIGURE 15.6: Create a query that provides information about employees in the sales group.



15.3.2.2 Using the combo box wizard

Although the basic process of setting the combo box properties remains the same regardless of whether its properties are set manually or using the wizard, the wizard is far more efficient when building a combo box based on a table or a query.

Show me (lesson15-3.avi)

21 Delete the existing `RepID` combo box.

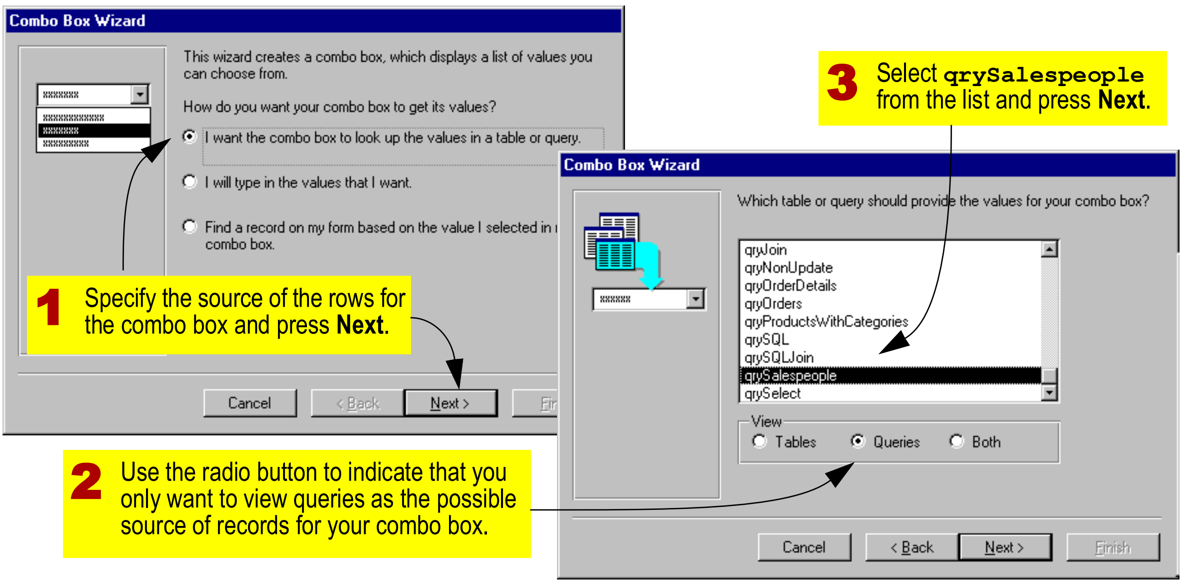
22 Ensure the wizard button ( ) in the toolbox is depressed (wizards are activated).

23 Repeat the steps for creating a bound combo box (i.e., select the combo box tool and drag the `RepID` field from the field list on to the detail section). Since the control wizard is now activated, you should get the dialog shown in Figure 15.7.

The wizard asks you to specify a number of things about the list of values that appears in the combo box:

1. the table (or query) from which values are taken;

FIGURE 15.7: Create a combo box using the combo box wizard.



2. the field (or fields) that you would like to show up as columns in the list;
3. the width of the field(s) in the list;
4. the column from the list (if more than one column is visible) that is bound to the underlying field; and,
5. the label that accompanies the combo box.

To complete your combo box, follow the instructions given by the wizard:

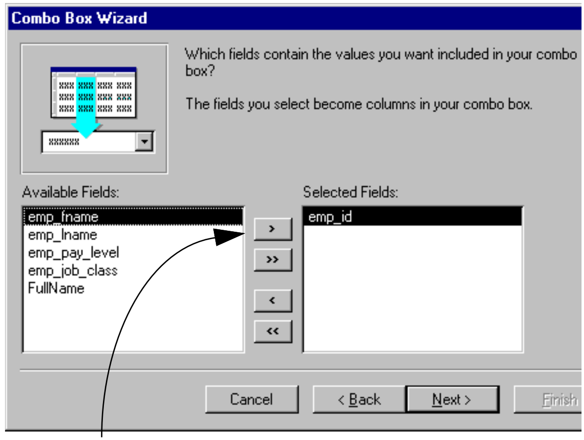
24 Indicate that the rows in the combo box should come from a table or query, and select the correct query, as shown in Figure 15.7.

25 Since `RepID` has to be a valid value of `Employees.EMP_ID`, select this field for the combo box as shown in Figure 15.8.

In the second last step of the wizard, you are asked whether to store the value picked from the combo box in a field. Since the drag-and-drop procedure you used to create the combo box binds it to the `RepID` field, you should not have to make any changes in the last two steps of the wizard.

Normally, in the last step of the wizard, you would enter a descriptive label to be shown on the left of the combo box. However, since you will shortly delete this

FIGURE 15.8: Select the field to show in the combo box.



1 Select `EMP_ID` as the field to show. No other choices are appropriate since `RepID` is a foreign key that references `EMP_ID`.

combo box and replace it with a better one, you should not waste any time formatting or tidying up these early efforts.

26 Switch to form view and test your combo box. It should look similar to that shown in Figure 15.1.

Clearly, updating or changing the values in the combo box is much easier when the combo box is based on a table. Since you have used the linked **`Employees`** table as the basis for **`qrySalespeople`**, then your combo box will always show the current members of the sales group (according to the human resources information system). This could be very helpful in a company with significant employee turnover.[1]

15.3.2.3 Showing more than one column in the combo box

One problem with the combo boxes created to this point is that they are not of much use to a user who is not familiar with the employee IDs of the sales group. In this section, you will use the **`FullName`** field of the **`qrySalespeople`** query to make the combo box easier to use.

Show me (lesson15-4.avi)

27 Delete the existing combo box and start again.

[1] The downside of using shared data is that an employee cannot be assigned to a region until he or she is added to the HR system. If the HR system is updated slowly or in batches, the use of shared data becomes a liability.

28 Fill in the wizard dialog sheets as in Section 15.3.2, but create a multi-column combo box, as shown in Figure 15.9 and Figure 15.10.

29 Verify that your combo box resembles the one shown in Figure 15.11.

15.3.2.4 Hiding the bound column

Assume that the **`EMP_ID`** values do not have any business meaning for users of this system. In such a case, you might be tempted to include only the **`FullName`** field in the combo box. However, this would not work because the target **`RepID`** field expects a long integer corresponding to a valid value of **`EMP_ID`**. If you try to stuff the text “Ben Sidhu” into a long integer field, you will get an error.

In this section, you will create a combo box identical to that shown in Figure 15.11 except that the key column (**`EMP_ID`**) will be hidden from view. Despite its invisibility, however, the **`EMP_ID`** column will still be bound to the **`RepID`** field of the underlying table and thus the combo box will work as it should.

Show me (lesson15-5.avi)

30 Delete the existing combo box and start again using the combo box wizard.

31 Include both the **`EMP_ID`** and **`FullName`** fields in the combo box.

32 Resize the **`EMP_ID`** column to a width of zero, as shown in Figure 15.12.

If you base your combo box on a table with a non-concatenated primary key, then ACCESS (version 7.0 and greater) provides a check box to hide the key, as shown in Figure 15.13. However, since the **`RepID`** combo box in the current example is based on a query, this feature is not available.

33 Complete the combo box as in Section 15.3.2.3.

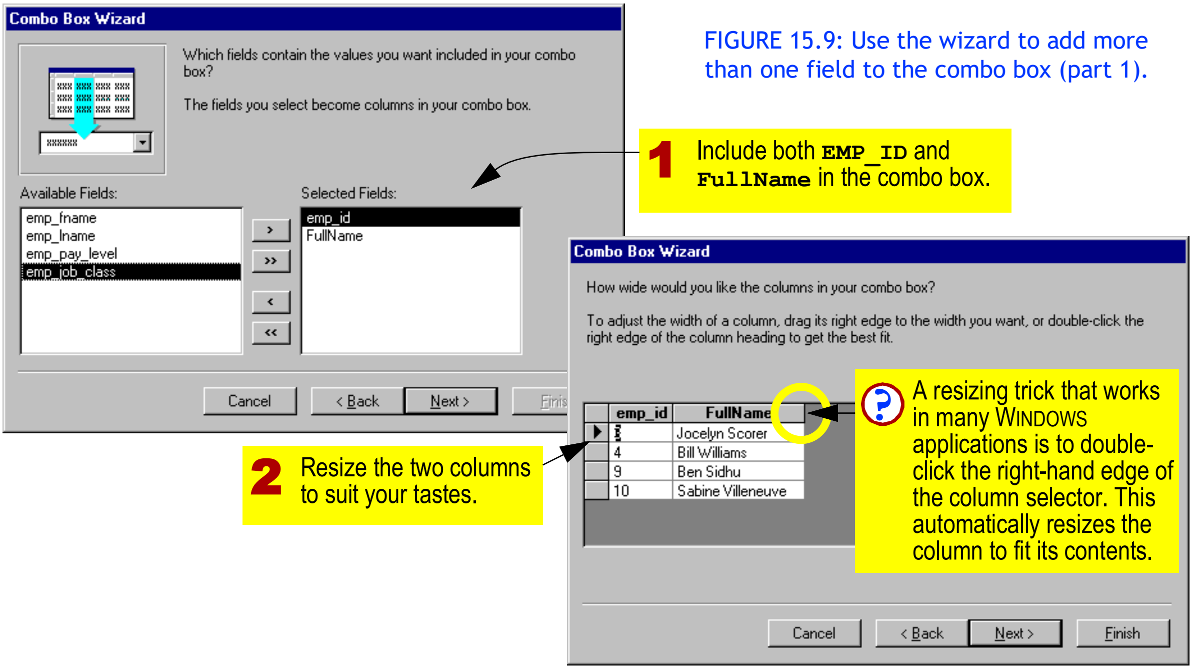
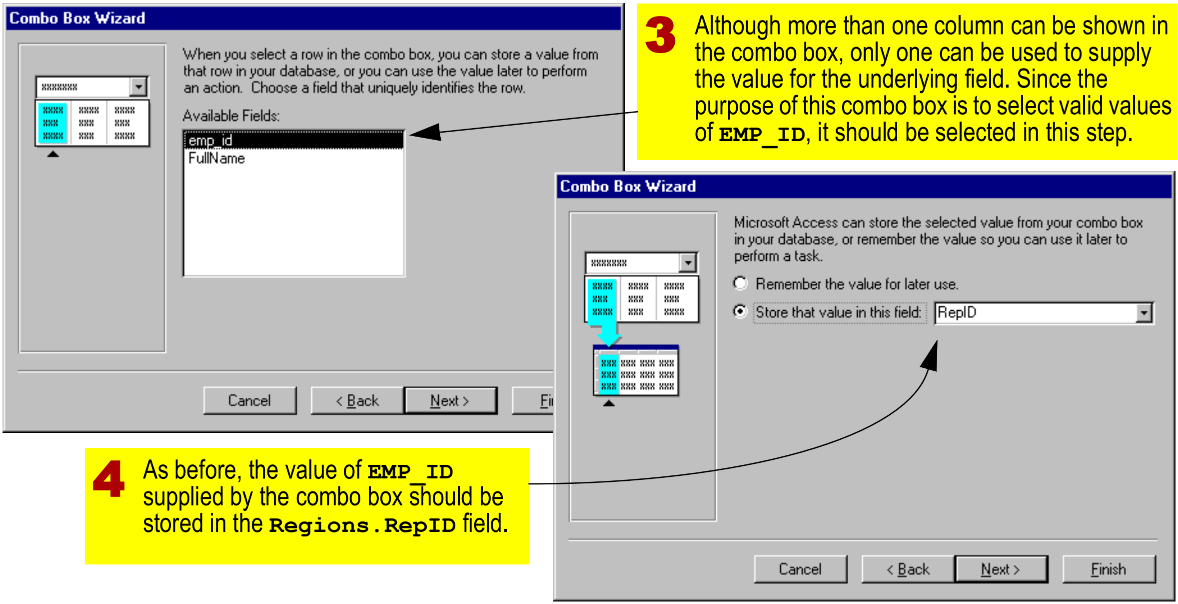


FIGURE 15.9: Use the wizard to add more than one field to the combo box (part 1).

FIGURE 15.10: Use the wizard to add more than one field to the combo box (part 2).



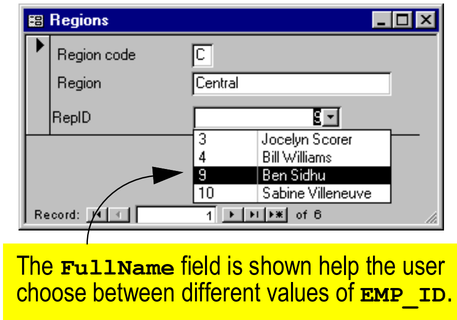
34 Ensure that the **Input Mask** property for the combo box (which is inherited from the `RepID` field's **Input Mask** property) is blank.

If you create an input mask for a field and then use a multi-column combo box to display an entirely different data type in the control, you must remove the input mask for the control.

35 Change the label for the `RepID` field from "RepID" to "Sales rep".

Although the combo box really contains a `RepID`, it looks to the user like the combo

FIGURE 15.11: A combo box showing multiple columns from the source query.



box is bound to the employee's name. You should change the label to make this illusion complete.

36 Verify that the resulting combo box resembles that shown in Figure 15.14.

15.3.2.5 Changing the order of rows in the combo box

In some cases, you may want to make minor modifications to the appearance of the combo box without altering the source table or query. For example, in the combo box in Figure 15.14, you may wish to show the members of the sales group in alphabetical order. You can do this by changing the embedded SQL statement in the **Row Source** property of the combo box.

Show me (lesson15-6.avi)

37 Bring up the property sheet for the **RepID** combo box.

38 Put the cursor in the **Row Source** property and press the builder button (...). This shows the embedded SQL statement in the QBE editor.

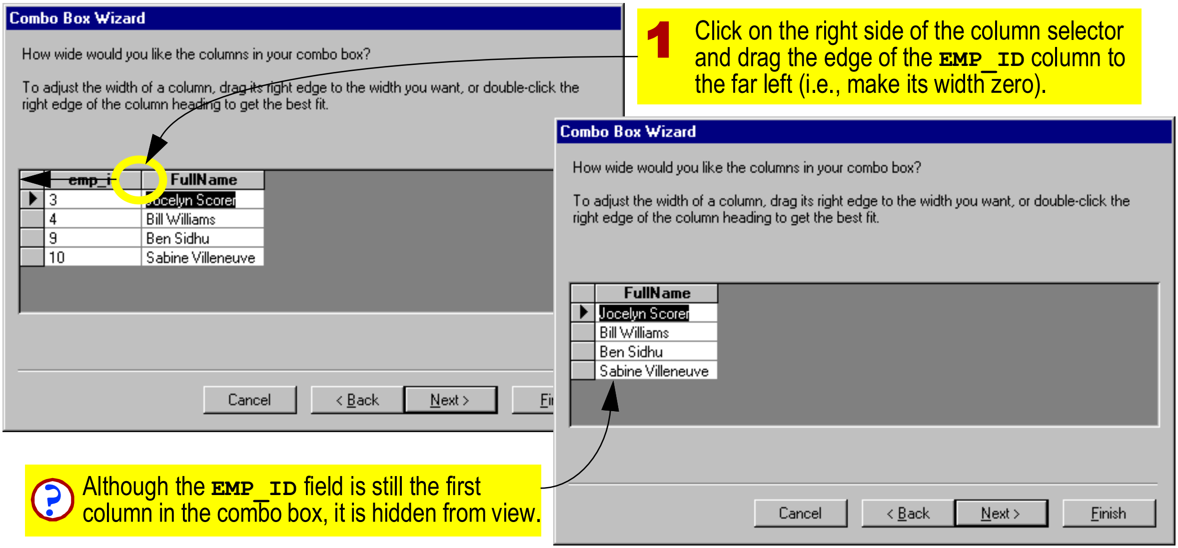
39 Modify the query to sort by **FullName**, as shown in Figure 15.15.

40 Instead of saving the query in the normal way, simply close the QBE box using the close button (☒).

15.3.3 Changing a form's tab order

A form's tab order determines the order in which the objects on a form are visited when the **Tab** or **Enter** (or **Return**) keys are pressed. ACCESS sets the tab order according to the sequence in which controls are added to the form. As a result, when you delete a text box and replace it with a combo box or some other control, the new control becomes the last item in the tab order regardless of its position on the form.

FIGURE 15.12: Resize the columns to hide the key.



To illustrate the problem, you are going to create a `CustID` combo box on the order form that you created in Lesson 14.

Show me (lesson15-7.avi)

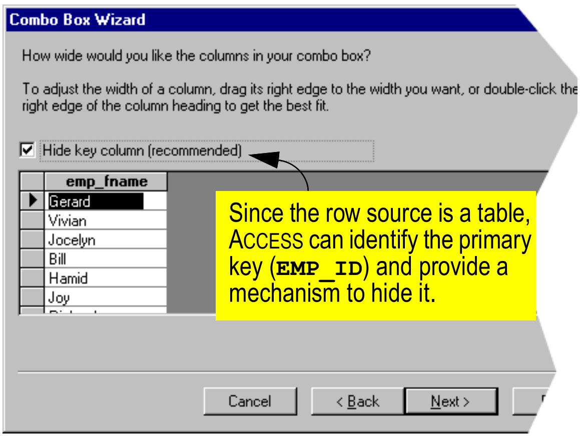
41 Open `frmOrders` in design mode and delete the `CustID` textbox.

42 Create a combo box that uses values from the `Customers` table.

43 Hide `CustID` in the combo box.

Anytime you use an AutoNumber to automatically generate unique keys for a table, it probably means that the key has no business meaning and should be hidden from users whenever possible.

FIGURE 15.13: The “hide key” option is available when using a table for a row source.

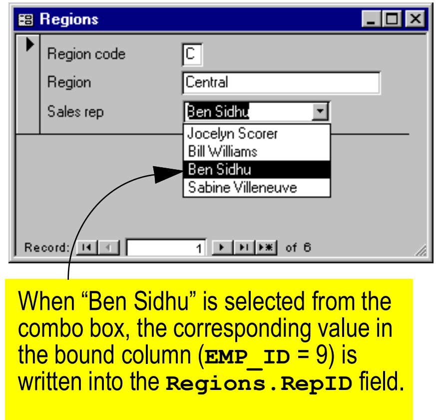


44 Use “Customer name” for the combo box’s label.

45 Switch to form view and use the **Tab** key to move from field to field.

Notice that the focus seems to skip the combo box when tabbing from field to field. If you put the cursor on the combo box and press the tab key, you will move to the next record because `CustID` is now the last control on the form.

FIGURE 15.14: An multi-column combo box with the bound column hidden.



46 To fix the problem, return to form design mode and select **View** → **Tab Order** from the main menu.

In ACCESS version 2.0, the menu structure is slightly different: Select **Edit** → **Tab Order** instead.

47 Move the `CustID` field from the end of the list to an appropriate location, as shown in Figure 15.16.

FIGURE 15.15: An multi-column combo box with the bound column hidden.

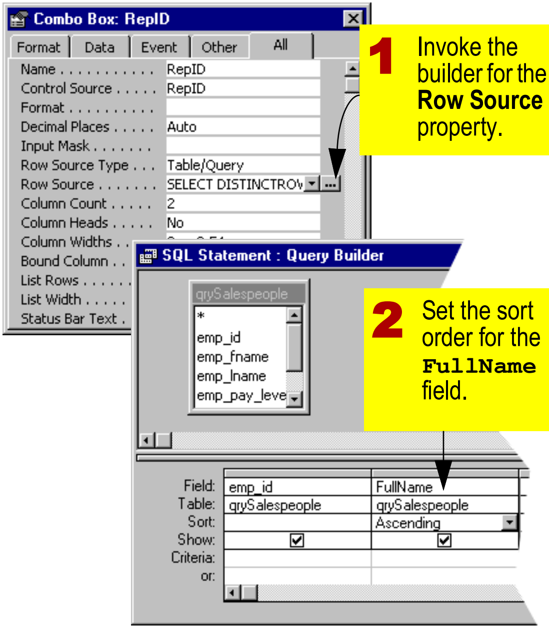
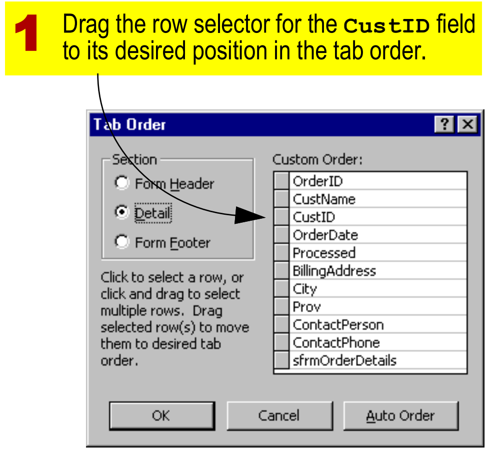


FIGURE 15.16: Set the tab order for the new combo box.



48 Since the `CustID` combo box now shows the customers name, delete the `CustName` textbox from the form.

15.4 Discussion

15.4.1 Why you should never bind a combo box to a primary key.

Once new users learn how to create combo boxes, a mistake they often make is to put a combo box on everything. There are certain situations, however, in which the use of a combo box is simply incorrect.

For example, it never makes sense to put a combo box on a non-concatenated primary key. To illustrate, consider the **Employees** form shown in Figure 15.17. On this form, the **EMP_ID** text box has been replaced with a combo box that draws its values from the same **Employees** table to which the form is bound.

This combo box appears to work. However, if you think about it, it makes no sense: The form in Figure 15.17 is a window on the **Employees** table. As such, when the **EMP_ID** combo box is used, one of two things can occur depending on whether a new record is being created or an existing record is being edited:

1. **A new record is being created** — If a new employee is being added to the database, a unique value of **EMP_ID** must be created to distinguish the new employee from existing employees. However, the combo box only shows **EMP_ID** values of *existing* employees. If the **Limit To List** property is set to Yes, then the combo box actually prevents the user from entering a unique **EMP_ID** value.
2. **An existing record is being edited** — It is important to remember that a combo box has no intrinsic search capability. As such, selecting "Bill Williams" in the combo box does not take you to the employee record belonging to Bill Williams. Rather, selecting Bill Williams from the combo box is identical to typing "4" over whatever is currently in the **EMP_ID** field. For example, in Figure 15.17, Gerard Huff's employee ID is overwritten by Bill William's employee ID. Obviously, this generates an error.

A combo box may make sense when the key is concatenated. For example, in your **OrderDetails** subform, a combo box should be used to select values for **ProductID** even through **ProductID** and **OrderID** form a concatenated key.

15.4.2 Controls and widgets

Predefined controls are becoming increasingly popular in software development. Although MICROSOFT includes several predefined controls with ACCESS (such as combo boxes, check boxes, radio buttons, etc.), a large number of more complex or specialized controls are available from MICROSOFT and other vendors such as WWW.COMPONENTSOURCE.COM. In addition, you can write your own custom controls using a language like VISUAL C++ or VISUAL BASIC and incorporate them into many different forms and applications.

An example of a more complex control is the calendar control shown in Figure 15.18. A calendar control can be added to a form to simplify the entry of dates. MICROSOFT calls such components ACTIVEX controls (formerly known as OLE controls). Non-WINDOWS environments also support components, but tend to favor the more generic terms “interface components” or “widgets”. In JAVA, SUN’s SWING library provides a large collection of interface components that conform to the JAVA BEANS specification.

There are two main advantages of using pre-packaged controls. First, they cut down on the time it takes to develop and test an application. Second, they are standardized so that users encounter the same basic behavior in all applications.

Non-interface (invisible) controls for doing chores like credit card processing, communicating over the internet, and encryption are also widely available.

15.5 Application to the project

There are a number of forms in your assignment that can be greatly enhanced by combo boxes.

49 Ensure that you have created a combo box on your order form to allow users to select customers by name rather than `CustID`.

FIGURE 15.18: A calendar control on a form.

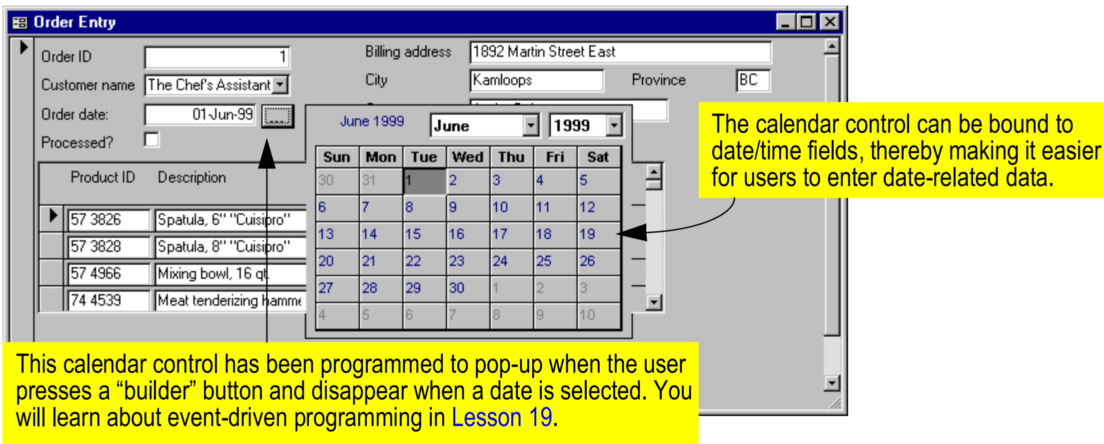
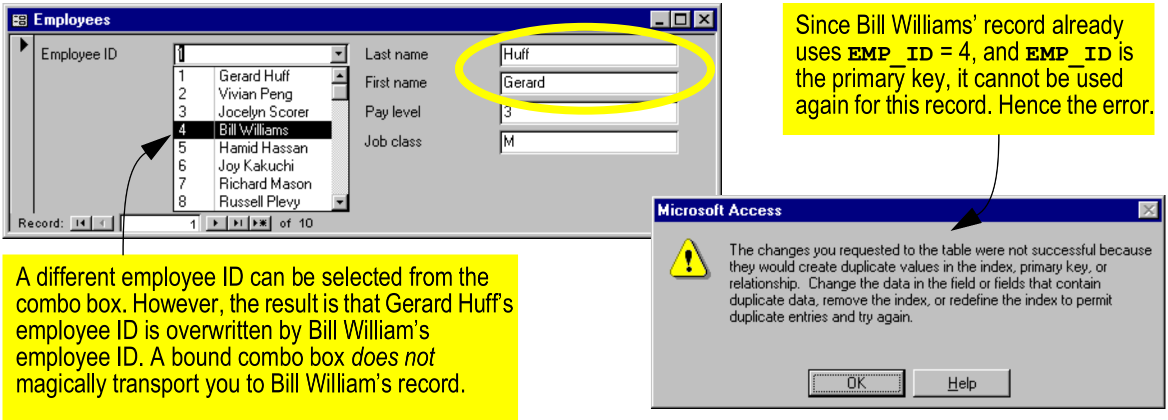


FIGURE 15.17: A combo box on a primary key field never makes sense.



50 Disable all the fields on the order form that you do not want users to change during order entry.

51 Create a combo box in your order details subform to allow the user to select products.

HINT: The `ProductID` values are used by both you and your customers to identify products. In other words, the field has meaning outside of the information system and should not be hidden in combo boxes bound to the `ProductID` field. In addition, the items in the product list should be sorted by `ProductID` to make it easier to select a product by typing the first few numbers.

HINT: It is very easy for users to confuse two similar `ProductID`s. To minimize the possibility of entering the wrong product number, you should show the product description in the combo box. In this way, users can confirm that the `ProductID` is correct before they make a selection.

Lesson 16: Parameter queries

16.1 Introduction: Dynamic queries using parameters

The last few lessons have been primarily concerned with interface issues. In the next several lessons, the focus shifts to transaction processing—making changes to data in response to business events.

A **parameter query** is a query in which the criteria for selecting records are determined when the query is executed rather than when the query is designed. For example, recall the select query shown in Figure 10.5. The criteria in the query were set so that the resulting record set consists exclusively of records that have a `UnitPrice` greater than $20.

If you wanted a different set of results (say products that cost more than $50), you would have to open the query in design view, change the criterion, and rerun the query. However, if a parameter is used for the criterion, ACCESS will determine the value of the parameter when the query is executed (for example, by prompting the user or pulling the value off an open form). The result is a flexible query.

When the concepts from this tutorial are combined with action queries and event handlers, you will have all the tools required to create a basic transaction processing system without writing a single line of programming code.

16.2 Learning objectives

- understand the way in which parameters can be used to create flexible queries
- prompt the user to enter parameter values
- create a query whose results depend on a value on a form

16.3 Exercises

16.3.1 Simple parameter queries

1. Create a query like `qryBasics` (recall Figure 10.5) and save it under the name `pqryExpensiveProducts`.

2. Replace the criterion in the `UnitPrice` column with a variable criterion: `> [X]` as shown in Figure 16.1.

Show me (lesson16-1.avi)

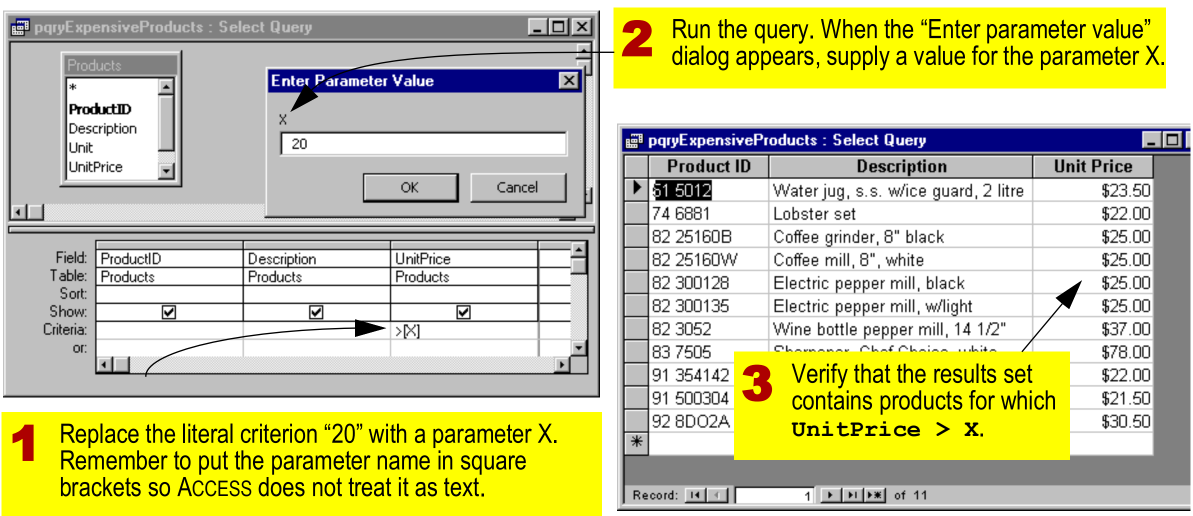
ACCESS expects criteria to be literal strings of text and automatically adds quotation marks to everything in the criteria row. Since the expression **`UnitPrice > "X"`** is not what we want, we have to tell ACCESS that **`x`** is the *name* of a parameter (not the letter "X") by enclosing the parameter name in square brackets.

3 Execute the query as shown in Figure 16.1.

When ACCESS encounters a variable (i.e., something that is not a literal string) during execution, it attempts to bind the variable to some value. To do this, the program does the following:

1. ACCESS checks whether the variable is the name of a field or a calculated field in the query. If it is, the variable is bound to the current value of the field. For example, if

FIGURE 16.1: Convert a select query into a parameter query.



the parameter is named **[ProductID]**, ACCESS replaces the parameter with the current value of the **ProductID** field (e.g., "51 5012"). Since **x** is not the name of a field or a calculated field in this particular query, ACCESS continues looking.

2. ACCESS attempts to resolve the parameter as a reference to something within the current environment (e.g., the value on an open form). Since there is nothing called **x** in the current environment, ACCESS continues looking.
3. As a last resort, ACCESS asks the user for the value of the parameter via the "Enter Parameter Value" dialog box.

Note that the spelling mistakes discussed in Section 11.3.2 are processed by ACCESS as parameters.

4 Press the requery button (**Shift-F9**) to re-execute the query. This time, enter a different value for **x** (e.g., 50).

The power of the parameter query in Figure 16.1 is that the dollar value denoting "expensive" can be defined at query execution time rather than at query design time.

16.3.2 Using parameters to generate prompts

Since the name of the parameter can be anything (as long as it is enclosed in square brackets), you can exploit this feature to create quick and easy dialog boxes.

5 Change the name of your **UnitPrice** parameter from **[X]** to **[Show products with a unit price greater than:]**.

6 Run the query, as shown in Figure 16.2.

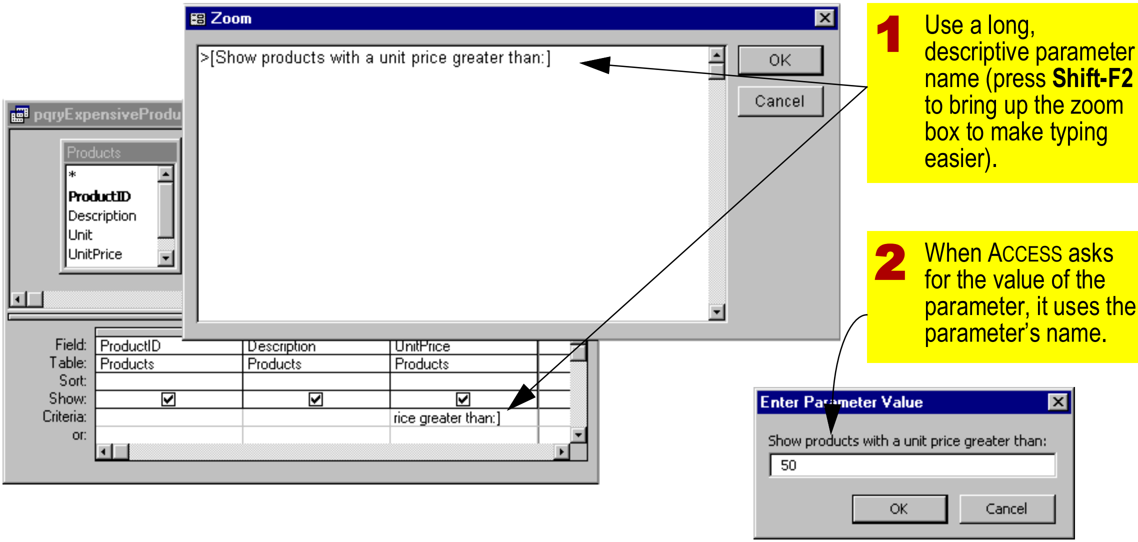
To create a *real* dialog box, you would use an unbound form and provide additional options such as a **Cancel** button.

16.3.3 Using values on forms as parameters

A common requirement is to use the value on a form to influence the outcome of a query. For instance, if the user is viewing information about regions, it may be useful to be able to generate a list of customers in the region currently being viewed without the user having to enter any additional information.

Of course, if all you want to do is view the customers, then a synchronized form/subform works well. However, if you want to do more than view the customer data, you will need a parameter query that pulls the value of a

FIGURE 16.2: Select a parameter name that generates a quick-and-easy dialog box.



parameter directly from the open form. The basic idea is shown in Figure 16.3.

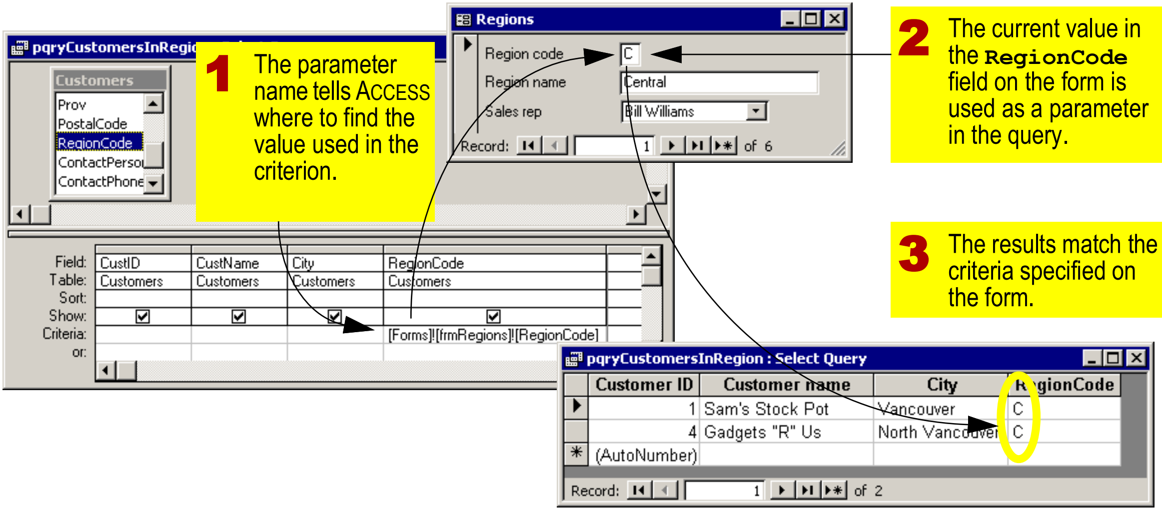
7 If you do not already have a `frmRegions` form, create a very simple one.

8 Leave the form open (in form view or design view, it does not matter).

The key to making this parameter query work is to provide a parameter name that correctly references the form object containing the value of interest. In order to avoid having to remember ACCESS' convoluted syntax for naming objects on forms, you can invoke the **expression builder** to select the correct name from the hierarchy of database objects.

Show me (lesson16-2.avi)

FIGURE 16.3: Using the value on an open form as a parameter in a query.



9 Create a new query called **`pqryCustomersInRegion`**. Project a few customer fields including **`RegionCode`**.

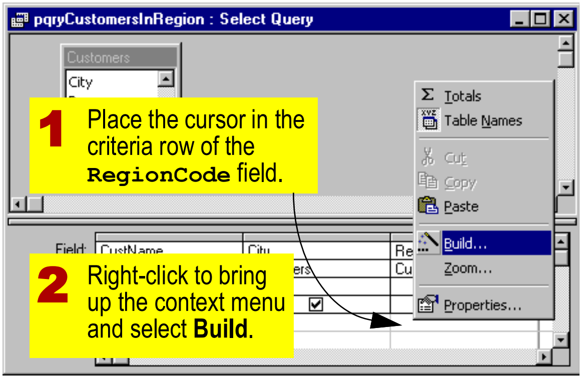
10 Move to the criteria row for **`RegionCode`** and invoke the expression builder, as shown in Figure 16.4.

11 Perform the steps shown in Figure 16.5 to create a parameter that references the **`RegionCode`** field on the **`frmRegions`** form.

12 View the query in datasheet mode. The results should correspond to the region showing on the regions form.

13 Move to a new record on the form. Notice that you have to requery the form

FIGURE 16.4: Invoke the builder to build a parameter.



(**Shift-F9**) in order for the new parameter value to be used (see Figure 16.6).

ACCESS does not continuously monitor the parameter for changes. When the query is opened, ACCESS evaluates the parameter and selects according to that value. If you want to force the query to re-evaluate, you have two choices: close and reopen the query or execute the requery action.

If the form is closed, ACCESS is unable to resolve the name of the criterion and has to prompt the user.

14 Close **`frmRegions`** and requery the parameter query. You should get the "enter parameter" dialog box shown in Figure 16.7.

16.4 Application to the project

16.4.1 Selecting the current order

Parameter queries provide a convenient way to control what appears on a form or a report based on what the user is currently viewing on screen. For example, once an order is entered into the order entry system, a user may want to print or fax an invoice to the customer of that order. To implement this feature, an invoice report is created and bound to a parameter query. The parameter query uses the techniques described in Section 16.3.3 to pull the value of the **`OrderID`** parameter off the currently visible order.

15 Create a parameter query called **`pqryInvoice`** that pulls its **`OrderID`** criteria from the order form. In addition to order information, your invoice should show information about the customer to whom the invoice is being sent.

HINT: Remember when testing the query that the order form must be open in order for ACCESS to find the parameter value.

FIGURE 16.5: Use the builder to select the name of the object you want to use as a parameter.

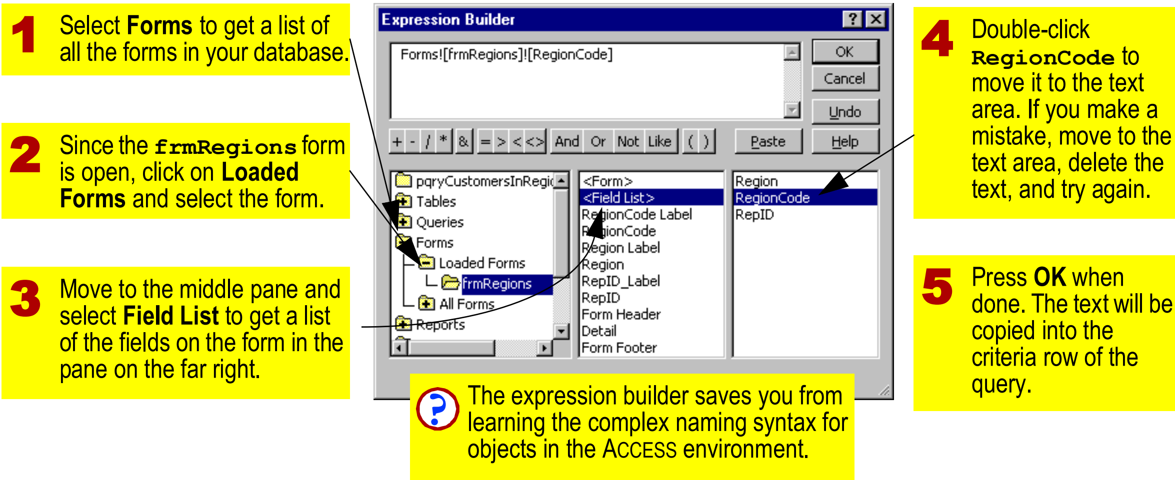
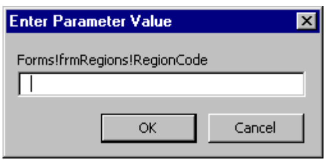


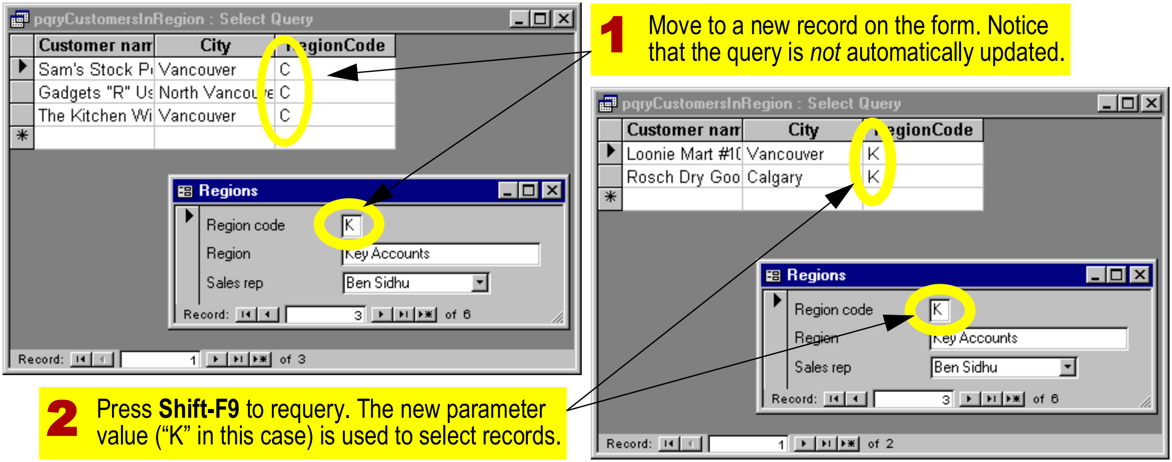
FIGURE 16.7: When the form is closed, ACCESS cannot resolve the parameter's value.



16.4.2 Using the report writer

Since invoices are meant to be printed, they should be created with the report writer. Creating reports is mostly a mechanical task and contributes little to your understanding of the relational database model. As a consequence, report creation is not covered in these lessons. However, you are not entirely on your own. ACCESS provides a graphical report writer and a report wizard to simplify the process of creating reports.

FIGURE 16.6: Requery the results set to reflect changes on the form.



The structure of an invoice is similar to the structure of the order form you created in Lesson 14: information about the order is shown at the top and information about the order details are shown in the invoice body. The main difference between an order form and an invoice report is that you only need to generate the invoice for the currently visible order. You do not want to generate invoices for all the orders in the `Orders` table because chances are that these invoices have already been created.

As was the case with the order form, you should create a report for the order, a second report for the order details, and combine them into a report/subreport structure linked on `OrderID`.

16 Use the report wizard to create a columnar main report based on `pqryInvoice`. Test it to make sure it gives the correct results and save it as `rptInvoice`.

17 Use the report wizard to create a tabular subreport that shows order details. You should be able to use your `qryOrderDetails` query as the record source for the subreport.

Since the invoice is not used for decision making, it can show less information than the order subform. For example, there is no requirement for the invoice to show confidential information such as quantity on hand.

18 Use the drag-and-drop process described in Section 14.3.3 to create a subreport control on the `rptInvoice` form.

Report/subreport synchronization is very important. Make sure you verify the link fields for the subreport control in the same way that you verified the link fields for your subform controls (review Section 14.3.4 as required).

Lesson 17: Action queries

17.1 Introduction: Queries that change data

All of the queries that you have created to this point have been variations of "select" queries. Select queries are used to display data, but do not actually change the data in any way. In this lesson, you are going to learn about action queries.

17.1.1 What is an action query?

Action queries are used to change the data in existing tables or make new tables based on the query's results set. The primary advantage of action queries is that they allow you to modify a large number of records without having to resort to writing VISUAL BASIC programs.

ACCESS provides four different types of action queries:

1. **Make table** — creates a new table based on the results set of the query;
2. **Append** — similar to a make-table query, except that the results set of the query is appended to an existing table;
3. **Update** — allows the values of one or more fields in the results set to be modified; and,
4. **Delete** — deletes all the records in the results set from the underlying table.

Since the operation of all four types of action queries is similar, we will focus on update and make-table queries in this tutorial.

17.1.2 Why use action queries?

The `Products` table includes a field called `QtyOnHand` that stores the inventory level for each product. Note, however, that this information already exists in the database. In principle, one could calculate the quantity on hand for each product by summing all the input transactions (shipments from suppliers) and subtracting all the output transactions (shipments to customers).

Given the emphasis to this point on minimizing the amount of redundancy and dependency in our databases, it may seem odd to store `QtyOnHand` when it can (in principle) be calculated whenever it is needed. The problem is that as the number of transactions grows large, it may become infeasible from a performance point of view to continuously recompute the inventory level for each product.[1]

Instead, we use "master" fields such as `QtyOnHand` to store status information about transactions. The advantage of storing this type of summary information is that it is available immediately. The disadvantage is that great care must be taken to insure that it is consistent with the information in the transactions that it summarizes.

17.1.3 Rolling back updates

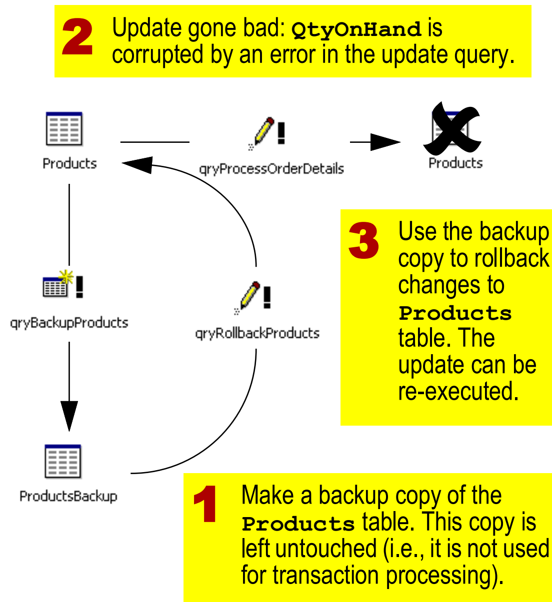
Since action queries permanently modify the data in tables, and since there is no undo feature for action queries, it is a good idea to create a mechanism to undo the effects of the query before executing it. Figure 17.1 shows the basic elements of the simple rollback feature that you are going to implement for your project.

The rollback in Figure 17.1 is “simple” because it only allows you to restore the **`Products.QtyOnHand`** field to its state when the **`ProductsBackup`** table was created. To implement a more realistic rollback feature, you would have to update the backup copy periodically.

---

1 In practice, one must also account for changes to inventory levels (e.g., theft, breakage, spoilage) that do not appear as transactions. Hence the need to periodically count the inventory and reconcile any discrepancies.

FIGURE 17.1: Using action queries to enable a simple rollback feature.



This rollback feature will allow you to test your action queries without having to worry about corrupting the data in the **`Products`** table.

Rolling back transactions is so important that industrial strength databases (and even ACCESS) have a built-in infrastructure for automatic commit and rollback (see the on-line help system). The exercises in this section are intended as an introduction to action queries, not database recovery.

17.2 Learning objectives

- understand the difference between action queries and select queries
- make a backup copy of a table using an action query
- undo (rollback) an action query once it has been executed
- update only certain records in a table
- create a button on a form that executes an action query when pressed

17.3 Exercises

17.3.1 Using a make-table query to create a backup

One way to make a backup copy of the `Products` table is to use a make-table query.

Show me (lesson17-1.avi)

1. Create a select query based on the `Products` table and save it as `qryBackupProducts`.

2. Project the asterisk (*) into the query definition so that all the fields are included in the results set.

3. While still in query design mode, select **Query** → **Make Table** from the main menu and provide a name for the target table (e.g., `ProductsBackup`) as shown in Figure 17.2.

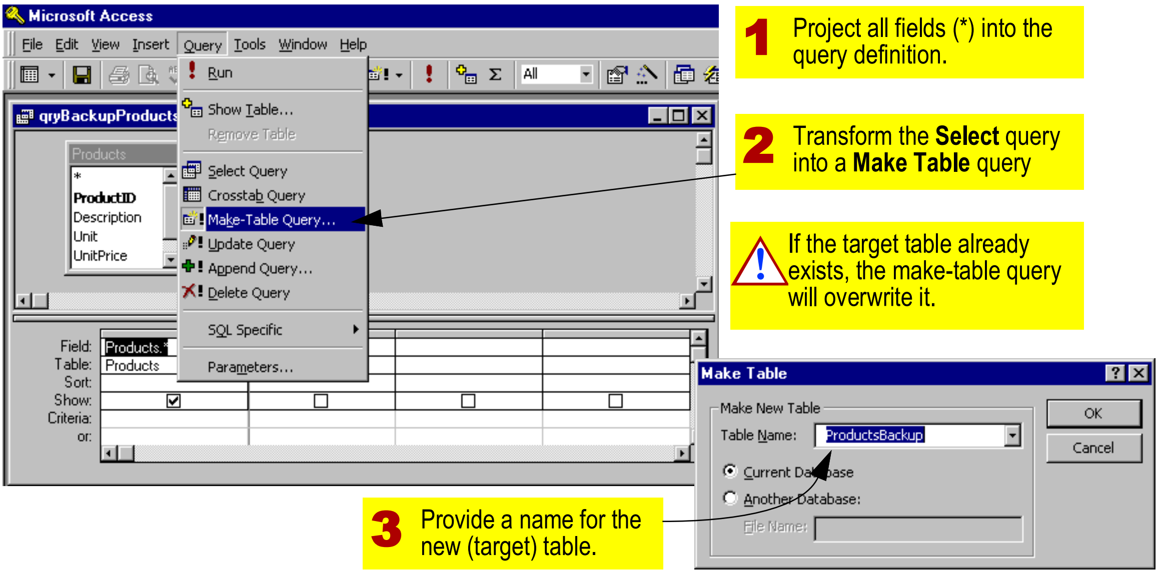
4. Switch to datasheet mode to preview the action.

Switching to datasheet mode does not execute the action query. Instead, it shows you the records that will be used when the action is run. In the case of a make-table query, the datasheet shows you the records that will be used to make the new table.

5. Select **Query** → **Run** from the main menu to execute the action query, as shown in Figure 17.3. Respond to warning boxes as required.

6. If you switch to the database window, you will notice that the new make-table query has a different icon than the select queries. If you click on the **Tables** tab, you will notice the new `ProductsBackup` table.

FIGURE 17.2: Use a make-table query to back up an existing table



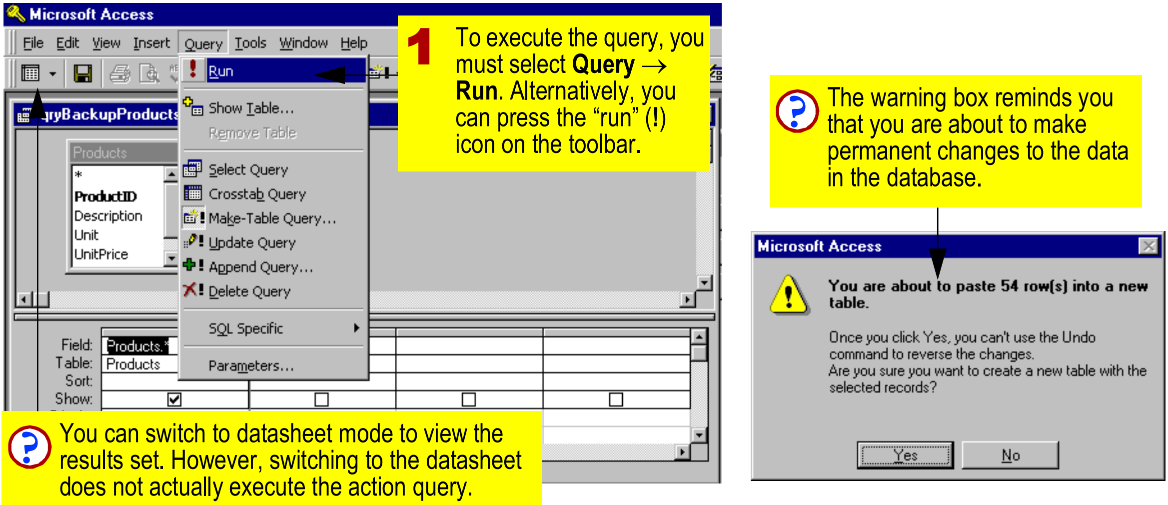
17.3.2 Using an update query to rollback changes

Having a backup table is not much use without a means of using it to restore the data in your original table. In this section, you will use an update query to replace the `QtyOnHand` values in your `Products` table with pristine values from your `ProductsBackup` table.

Show me (lesson17-2.avi)

7 Create a new query and add both the `Products` and the `ProductsBackup` tables. Save it as `qryRollbackProducts`.

FIGURE 17.3: Run the make-table query.



8 Since no relationship exists between these tables, create an *ad hoc* relationship within the query as shown in Figure 17.4.

9 Select **Query** → **Update** from the main menu. Note that this results in the addition of an **Update To** row in the query definition grid.

10 Project `QtyOnHand` into the query definition and fill in the **Update To** row as shown in Figure 17.5.

Now is a good point to stop and interpret what you have done so far:

1. By creating a relationship between the `Products` table and its backup, you are joining together records from both tables that share the same `ProductID`.

FIGURE 17.4: Create an *ad hoc* relationship between the table and its backup copy.

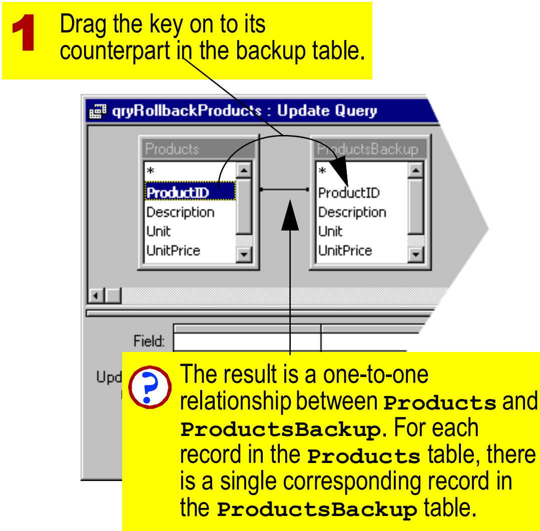
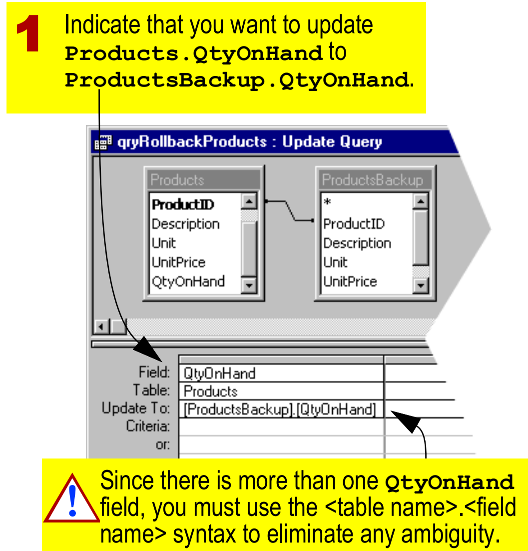


FIGURE 17.5: Fill in the **Update To** field.



2. By projecting **`Products.QtyOnHand`** into the query, you are making it the target for the update. No other field is modified by the action query.
3. By setting the **Update To** field to **`ProductsBackup.QtyOnHand`**, you are telling ACCESS to replace the contents of **`Products.QtyOnHand`** with the contents of **`ProductsBackup.QtyOnHand`**.

This update query can be used at any time to restore the inventory levels in the **`Products`** table to the values they contained when the **`ProductsBackup`** table was created.

Although it would be a simple matter to use the backup to replace the contents of all the fields in the **`Products`** table (e.g., **`Description`**, **`UnitPrice`**, and so on), `QtyOnHand` is the only field that can be corrupted by the update query you will create to process orders.

17.3.3 Using an update query to process the order

Now that you have an infrastructure for rolling back any errors, you can continue with the task of creating an action query to process orders.

Show me (lesson17-3.avi)

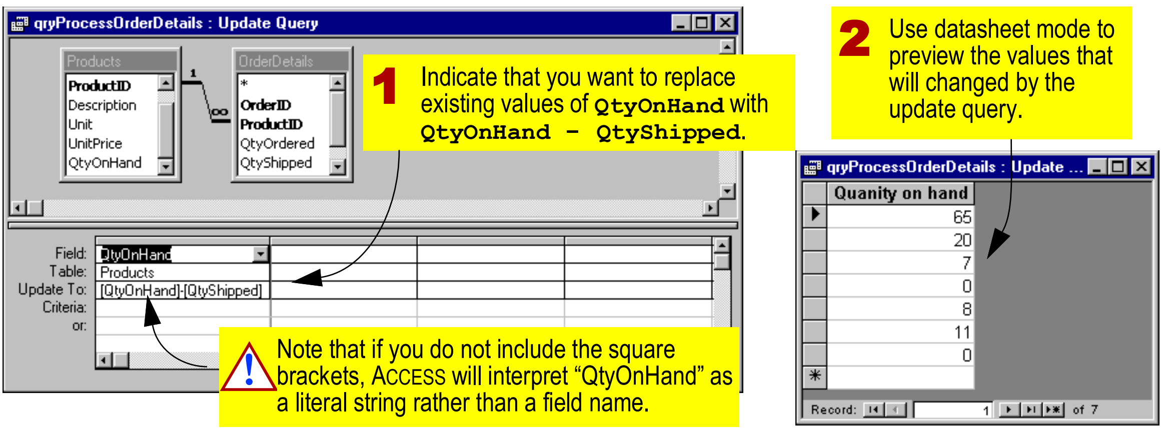
11 Create an update query based on the `OrderDetails` and `Products` tables and save it as `qryProcessOrderDetails`.

12 Set the **Update To** field to `[QtyOnHand]-[QtyShipped]`, as shown in Figure 17.6.

13 Execute the query.

Note that the query in Figure 17.6 updates the `QtyOnHand` field once for every value in the `OrderDetails` table. Consequently, if you enter a new order and run the query, you will end up subtracting all the items shipped in the new

FIGURE 17.6: Create an action query to process *all* orders.



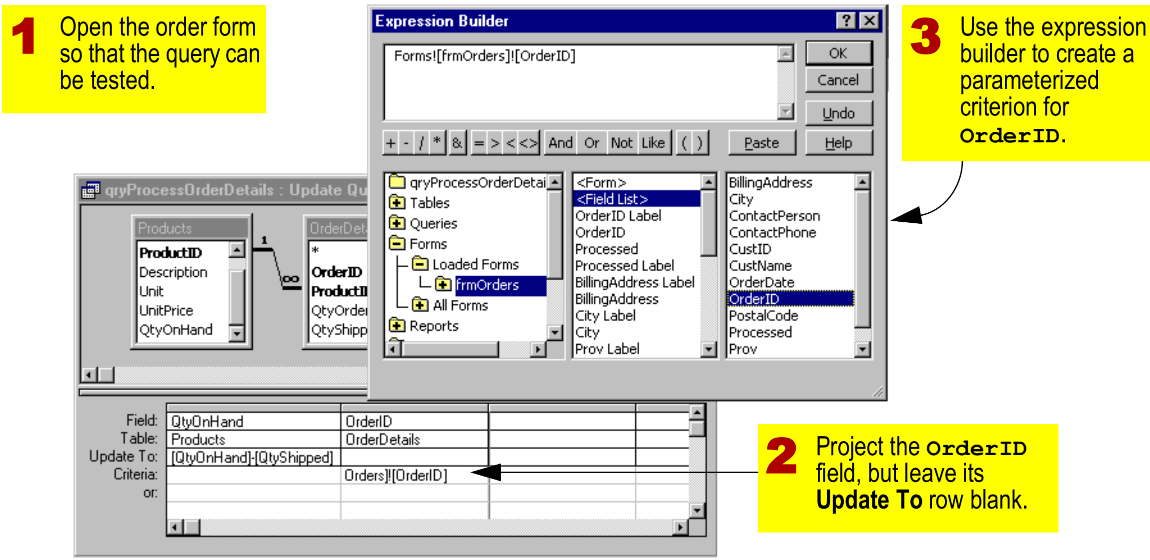
order plus all items shipped in previous orders. Clearly, this is incorrect.

A convenient means of avoiding this problem is to have the action query process items from one order only. To do this, add a parameter to your action query.

14 Open your `frmOrders` form so that it appears in the expression builder under “loaded forms”.

15 Switch back to the query and use the builder as shown in Figure 17.7 to enter a parameterized criterion for the `OrderID` field. The criterion should pull its value from the order form.

FIGURE 17.7: Create an action parameter query to process a single order only.



16 On the order form, navigate to an order that contains some order details.

17 Run the query and verify that the update has been performed successfully. Since the action query is making changes to your data, you will see a number of warning dialogs, as shown in Figure 17.8.

Once an action query is created, it has more in common with programs written in VISUAL BASIC than standard select queries. Double-clicking an action query executes it and, apart from the warning messages, there is no visible indication that the action has been performed.

Since the action query is also a parameter query, you should rename it to be consistent with the parameter query naming convention used in Lesson 16.

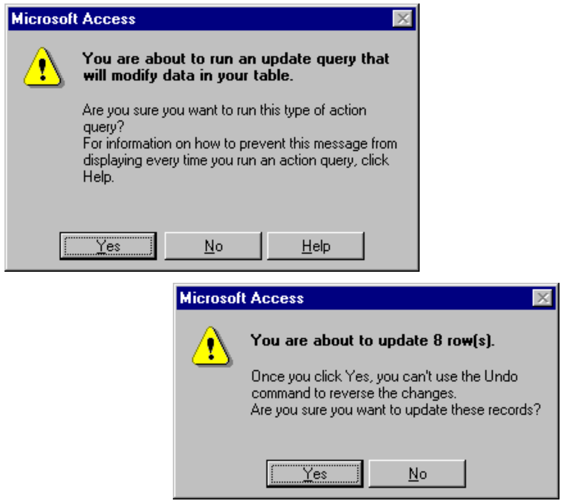
18 In the database window, right-click on `qryProcessOrderDetails`, select **Rename**, and change the prefix to “pqry”.

You cannot rename a database object if it is open. If your query is open, close it before attempting to rename it.

17.3.4 Rolling back changes

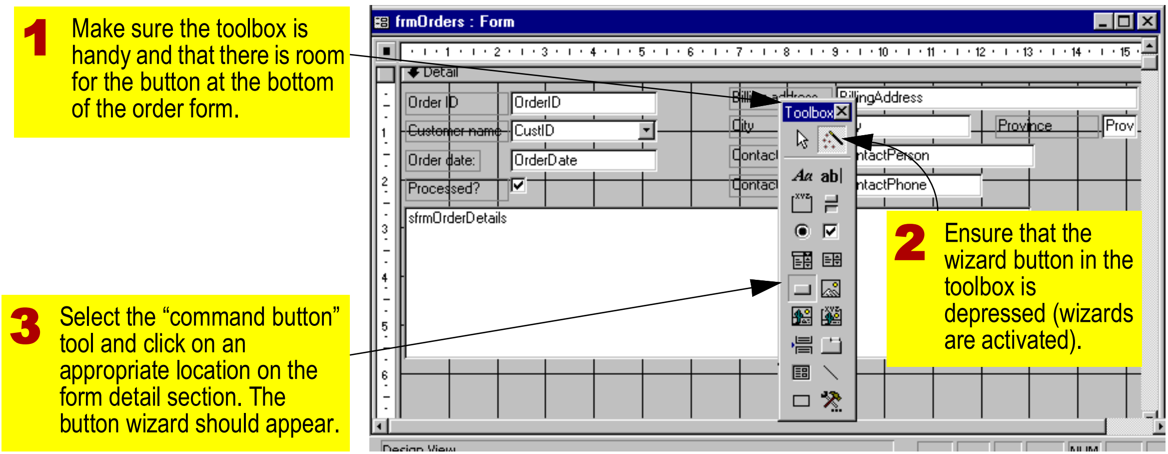
While testing the `pqryProcessOrderDetails` query, your exuberance may lead you to execute it more than once. To return the `Products` table to its state before any updates, all you need to do it run your rollback query.

FIGURE 17.8: Running an action query generates warning messages.



The number of records to be processed should correspond to the number of order details in the currently visible order.

FIGURE 17.9: Add a button to the form using the button wizard (part 1).



19 Run `qryRollbackProducts` by double-clicking its icon in the database window.

17.3.5 Attaching action queries to buttons

As a designer, you should not expect your users to understand your query naming convention, rummage through the queries listed in the database window, and execute the queries that need to be executed.

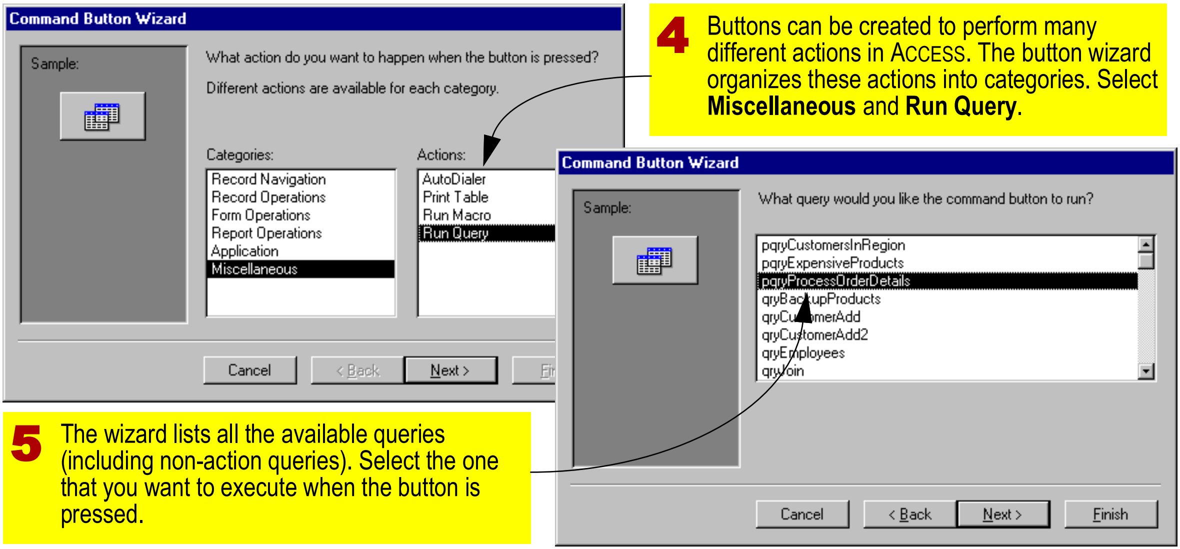
A better approach is to create buttons on forms and “attach” the action queries to the buttons. When the button is pressed, the query is executed. Although we have not yet discussed buttons (or events in general), the button wizard makes the creation of this type of form object straightforward.

Show me (lesson17-4.avi)

20 Switch to the design view of `frmOrders` and add a button as shown in Figure 17.9.

21 Attach the `pqryProcessOrderDetails` query to the button as shown in Figure 17.10.

FIGURE 17.10: Add a button to the form using the button wizard (part 2)



22 Provide a caption and a name for the button as shown in Figure 17.11.

23 Switch to form view. Press the button to run the query (alternatively, use the shortcut key by pressing **Alt-P**) as shown in Figure 17.12.

17.4 Application to the project

24 Ensure you have implemented and tested the action queries described in this lesson.

FIGURE 17.11: Add a button to the form using the button wizard (part 3)

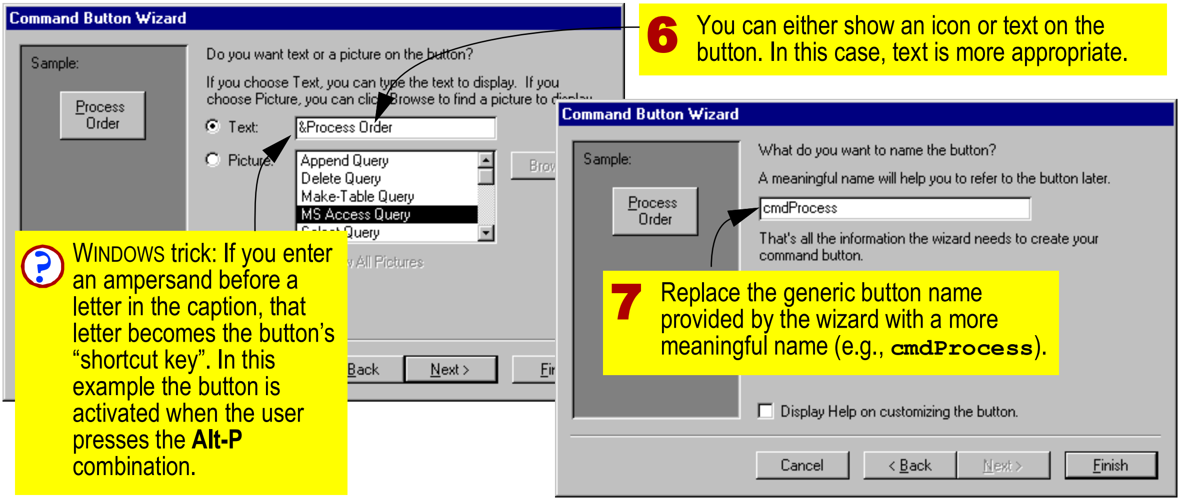
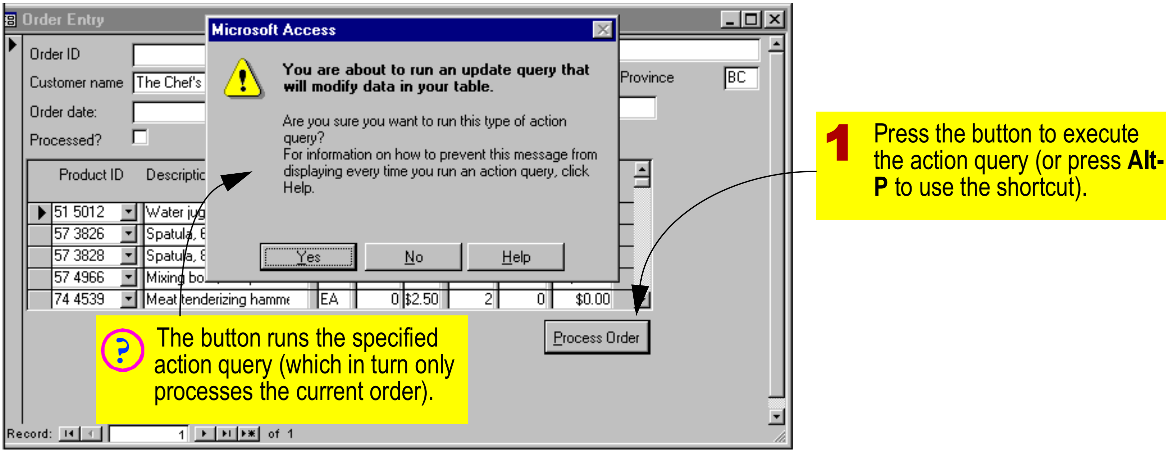


FIGURE 17.12: Execute the action query by pressing the button.



18.1 Introduction: Learning the basics of programming

Programming can be enormously complex and difficult, or it can be easy and straightforward. In many cases, the difference lies in the choice of tools used for writing the program. In other cases, it is a question of scale—large projects are unavoidably complex.

In this project, we are going to focus on solving small problems using an easy-to-use programming language. It is important to recognize, however, that basic programming concepts remain the same regardless of language or project scale. The programs you create here are trivial, but they introduce a handful of programming constructs that can be found in any "third generation" language, not just VISUAL BASIC.

Strictly speaking, the language that is included with ACCESS is not VISUAL BASIC—it is a subset of the full, stand-alone VISUAL BASIC language (which MICROSOFT sells separately). In ACCESS version 2.0, the subset is called "ACCESS BASIC". In version 7.0 and above, it is slightly enlarged subset called "VISUAL BASIC FOR APPLICATIONS" (VBA). In the context of the simple programs we are writing here, these terms are interchangeable.

ACCESS provides two ways of interacting with the VBA language. The most useful of the two is saved **modules** that contain VBA **procedures**. Procedures are batches of programming commands that can be executed to do interesting things like process transactions against master tables, provide sophisticated error checking, and so on.

The second way to interact with VBA is directly through the interpreter. When you type a statement into the interpreter, it is executed immediately. Although there is little business use for this feature, it does making learning the language and testing new statements easier.

In the first part of this lesson, you are going to invoke ACCESS' VBA interpreter and execute some very simple statements. In the second part of the tutorial, you are going to create VBA modules to explore generic programming constructs such as looping, conditional branching, and parameter passing.

18.2 Learning objectives

- invoke and use the debug/immediate window
- understand the difference between fundamental program constructs (statements, variables, the assignment operator, and predefined functions)
- understand the difference between subroutines and functions
- create a module containing VBA code
- gain experience with looping and conditional branching constructs
- use the VBA debugger in ACCESS
- understand the difference between an interpreted and compiled programming language

18.3 Exercises

18.3.1 Invoking the interpreter

1 Click on the **Modules** tab in the database window and press **New**.

This opens the module window which we will return to in Section 18.3.3.

You have to have a module window open in order for the debug window to be available from the menu.

2 Select **View** → **Debug Window** from the main menu. Note that **Ctrl-G** can be used in version 7.0 and above as a shortcut to bring up the debug window.

In version 2.0, the "debug" window is called the "immediate" window and you use **View** → **Immediate Window** to activate it.

18.3.2 Basic programming constructs

In this section, you will use the VBA interpreter to explore some fundamental programming constructs and VISUAL BASIC syntax.

18.3.2.1 Statements

Statements are built around special keywords in a programming language that do something when executed. For example, the `Print` statement in VBA "prints" an expression on the screen.

Show me (lesson18-1.avi)

3 In the debug window, type the following:

NL `Print "Hello world!"↵`

The ↵ symbol at the end of a line means "press the **Return** or **Enter** key". From this point forward, assume that each line is followed by an **Enter**.

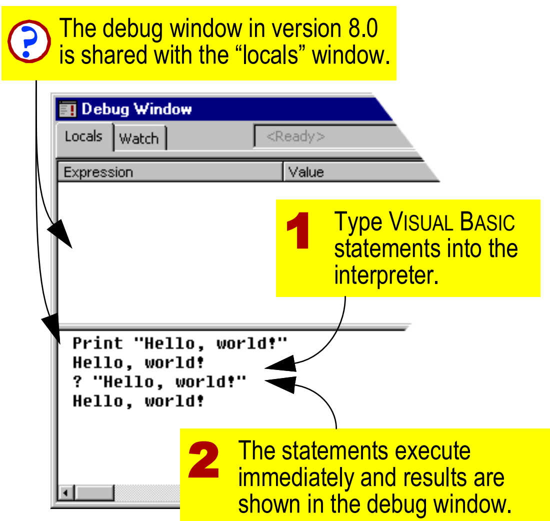
In VBA (as in all dialects of BASIC), the question mark (`?`) is typically used as shorthand for the `Print` statement.

4 Type the following into the debug window:

NL `? "Hello world!"`

As shown in Figure 18.1, the result is identical to first `Print` statement.

FIGURE 18.1: Interacting with the VISUAL BASIC interpreter.



18.3.2.2 Action statements

Actions are special statements that allow programmers to use elements of ACCESS' macro language in VBA programs. Some actions are stand-alone (such as the `MsgBox` action below) and others are actually methods of the `DoCmd` object (see Section 18.4.1 for more information on the evolution of VBA and its increasing use of object-oriented concepts). Although the inclusion of so many different types of statements results in a conceptual mess, it provides programers with a great deal of flexibility.

At this point, you should ignore theoretical language issues and accept VBA as a powerful-but-inconsistent friend.

5 Type the following into the debug window:

NL `MsgBox "Hello, world!"`

Show me (lesson18-2.avi)

The `MsgBox` macro action provides an easy way to create custom messages using the standard WINDOWS message box format.

The `DoCmd` object is just a kludge that brings VBA, object-orientation, and the macro language together. Ugly as it is, the `DoCmd` object has many methods that are worth learning about.

6 Type the following into the debug window:

```
NL  DoCmd.OpenForm "frmOrders"
```

You will notice that your order form opens in the background behind the debug window.

If you do not have a form called `frmOrders`, this method will result in an error.

18.3.2.3 Variables and assignment

A variable is space in memory to which you assign a name and a value. When you use the variable name in expressions, the programming language replaces the variable name with its assigned value at that particular instant.

Show me (lesson18-3.avi)

7 Type the following:

```
NL  s = "Hello"
NL  ? s & " world"
NL  ? "s" & " world"
```

In the first statement, the variable named `s` is created and the string "Hello" is assigned to it. Recall the function of the concatenation operator from Section 11.4.1. When the second statement is executed, VBA recognizes that `s` is a variable, not a string (since it is not in quotations marks). The interpreter replaces `s` with its value ("Hello") before executing the `Print` command. In the final statement, `s` is in quotation marks so it is interpreted as a **literal string**.

Contrary to the practice in languages like C and PASCAL, the equals sign (=) is used to **assign** values to variables. It is also used as the **equivalence operator** (e.g., does x = y?).

Within the debug window, any string of characters in quotations marks (e.g., "Hello") is interpreted as a literal string. Any string without quotation marks (e.g., `strName`) is interpreted as a variable or some other objects (such as a field) that is defined within the ACCESS environment.

18.3.2.4 Predefined functions

You were introduced to predefined functions in Section 11.4.2. In this section, you are going to explore some basic predefined functions for working with numbers and text. The results of these exercises are shown in Figure 18.2.

8 Print the cosine of 2π radians and then nest the cosine function within a conversion function to round to the nearest integer:

```
NL  pi = 3.14159
NL  ? cos(2*pi)
NL  ? CInt(cos(2*pi))
```

9 Convert a string of characters to uppercase:

```
NL  s = "basic or cobol"
NL  ? UCase(s)
```

10 Extract the middle six characters from a string starting at the fifth character:

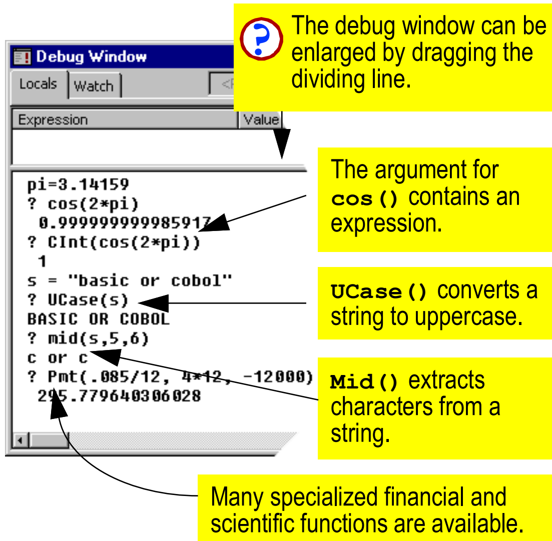
```
NL  ? mid (s,5,6)
```

11 Calculate the monthly payment for a $12,000 loan over four years with 8.5% interest (nominal) compounded monthly:

```
NL  ? Pmt(0.085/12, 4*12, -12000)
```

In VBA, as in ACCESS, case is ignored. As such `MID(s,5,6)` is identical to `mid(s,5,6)`. In addition, the amount of "whitespace" (space between elements of the statement) is irrelevant.

FIGURE 18.2: Using the VISUAL BASIC interpreter to test predefined functions.



18.3.2.5 Remark statements

When creating large programs, it is considered good programming practice to include adequate internal documentation. In other words, you should include comments throughout your code to explain to others and your future self what the program is doing.

Comment lines are ignored by the interpreter when the program is run. To designate a comment in VBA, use an apostrophe to start the comment, e.g.:

```
NL  'This is a comment line!
NL  Print "Hello" 'the comment starts
    here
```

The original REM (remark) statement from BASIC can also be used, but is less common.

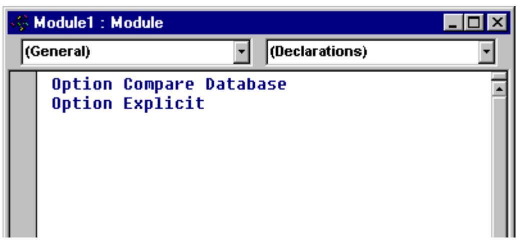
```
NL  REM This is also a comment (remark)
```

18.3.3 Creating a module

So far, you have written and executed VBA statements one at a time. A more useful technique is to bundle a number of VBA commands together in a procedure and run them in sequence. A module is a collection of statements that is saved with the database (like a form or a query).

12 Close the debug window so that the declaration page of the new module created in Section 18.3.3 is visible (see Figure 18.3).

FIGURE 18.3: The declarations page of a VISUAL BASIC module.



Show me (lesson18-4.avi)

The two lines:

```
NL  Option Compare Database
NL  Option Explicit
```

are included in the module by default. The `Option Compare` statement specifies the way in which strings are compared (e.g., are “wire whisk” and “WIRE WHISK” considered identical?). The `Option Explicit` statement forces you to declare all your variables before using them (variable declaration is discussed in Section 18.3.4.1).

In version 2.0, ACCESS does not automatically add the `Option Explicit` statement. You should add it yourself to the declarations section.

A module contains a declaration page and one or more pages containing procedures. In VBA, there are two types of procedures: subroutines and functions. The primary difference between the two is that subroutines simply execute whereas functions execute and return a value (e.g., `cos()`).

In version 2.0, only one subroutine or function shows in the window at a time. You must use the **Page Up** and **Page Down** keys to navigate the module.

18.3.4 Creating subroutines with looping and branching

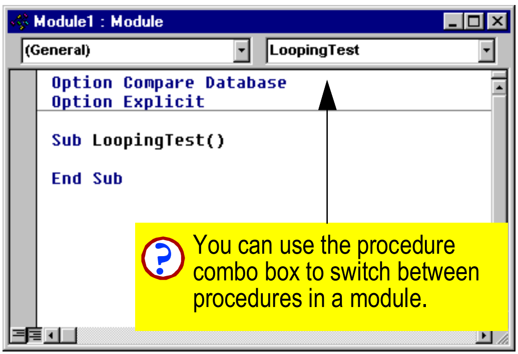
In this section, you will explore two of the most powerful constructs in computer programming: looping and conditional branching.

13 Create a new subroutine by typing the following anywhere on the declarations page of the open module:

```
NL  Sub LoopingTest()
```

Notice that ACCESS creates a new section/page in the module for the subroutine, as shown in Figure 18.4.

FIGURE 18.4: Create a new subroutine.



18.3.4.1 Declaring variables

When you declare a variable, you tell the programming environment to reserve some space in memory for the variable. Since the amount of space that is required depends on the type of data the variable is expected to contain (e.g., string, integer, Boolean, double-precision floating-point, etc.), you have to include data type information in the declaration statement.

In VBA, you use the `Dim` statement to declare variables.

14 Type the following into the space between the `Sub… End Sub` pair:

```
NL  Sub LoopingTest()
NL    Dim i as integer
NL    Dim s as string
NL  End Sub
```

15 Save the module as `basTesting`.

One of the most useful looping constructs is `For <condition>… Next`. All statements between the `For` and `Next` parts are repeated as long as the `<condition>` part is true. The index `i` is automatically incremented after each iteration.

16 Enter the remainder of the `LoopingTest` program:

```
NL  Sub LoopingTest()
NL    Dim i as integer
NL    Dim s as string
NL    s = "Loop number: "
NL    For i = 1 To 10
NL      Debug.Print s & i
NL    Next i
NL  End Sub
```

17 Save the module.

It is customary in most programming languages to use the **Tab** key to indent the elements within a loop slightly. This makes the program more readable.

Note that the `Print` statement in Figure 18.5 is prefaced by `Debug`. You can get away with the old style BASIC `Print` statement in the debug window. But within modules, VBA enforces a form of object-orientation. All stand-alone statements are replaced by methods of objects. In this case, the `Print` method window belongs to the `Debug` object.

18.3.4.2 Running the subroutine

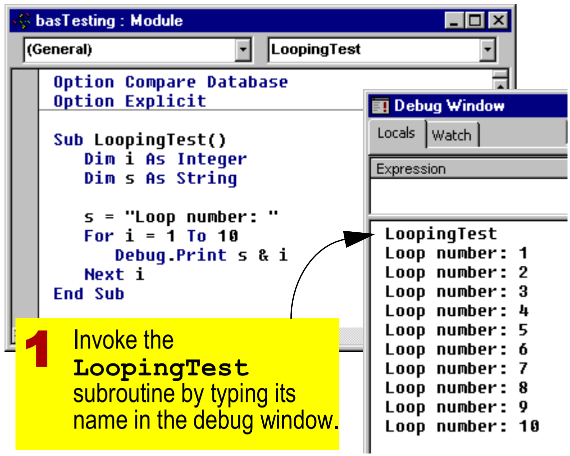
Now that you have created a subroutine, you need to run it to see that it works. To invoke a subroutine, you simply use its name like you would any statement.

Show me (lesson18-5.avi)

18 Select **View** → **Debug Window** from the menu (or press **Ctrl-G** in version 7.0 and above).

19 Type: `LoopingTest` in the debug window, as shown in Figure 18.5.

FIGURE 18.5: Run the ***LoopingTest*** subroutine in the debug window.



18.3.4.3 Conditional branching

We can use a different looping construct, **`Do Until <condition>… Loop`**, and the conditional branching construct, **`If <condition> Then… Else`**, to achieve the same result.

20 Type the following anywhere under the **`End Sub`** statement in order to create a new page in the module:

```
NL  Sub BranchingTest
```

21 Enter the following program:

```
NL  Sub BranchingTest
NL    Dim i As Integer, s As String
NL    Dim blDone As Boolean
NL    s = "Loop number: "
NL    i = 1 'initialize counter
NL    blDone = False
NL    Do Until blDone
NL      If i > 10 Then
NL        Debug.Print "All done"
NL        blDone = True
NL      Else
NL        Debug.Print s & i
NL        i = i + 1
NL      End If
NL    Loop
NL  End Sub
```

22 Run the program

18.3.5 Using the debugger

ACCESS provides a very good debugger to help you step through your programs and understand how they are executing. The two basic debugging constructs explored here are breakpoints and stepping (line-by-line execution).

Show me (lesson18-6.avi)

23 Move to the "**`Do Until blDone`**" line in the **`BranchingTest`** subroutine and select **Run → Toggle Breakpoint** from the menu (you can also press **F9** to toggle the breakpoint on a particular line of code).

Note that the line becomes highlighted, indicating the presence of an active breakpoint. When the program runs, the interpreter will suspend execution at this breakpoint and pass control of the program back to you.

24 Run the subroutine from the debug window. Execution should halt at the breakpoint.

25 Step through a couple of lines in the program line-by-line by pressing **F8**, as shown in Figure 18.6.

By stepping through a program line by line, you can usually find any program bugs. In addition,

FIGURE 18.6: Step through the each line in the program individually.

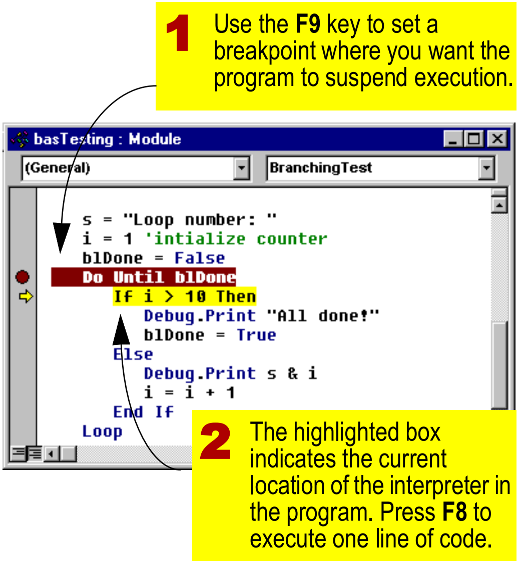
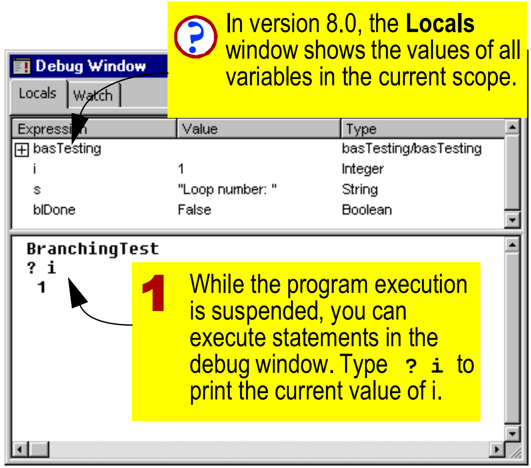


FIGURE 18.7: Use the debug window to print variable values while the program is running.



you can use the debug window to examine the value of variables while the program's execution is suspended.

26 Click on the debug window and type `? i` to see the current value of the variable `i`, as shown in Figure 18.7.

18.3.6 Passing parameters

In the `BranchingTest` subroutine, the loop starts at 1 and repeats until the counter `i` reaches 10. It may be preferable, however, to set the start and finish quantities when the subroutine is executed. To achieve this, you pass **parameters** (or **arguments**) to the subroutine.

The main difference between passed parameters and other variables in a procedure is that passed parameters are declared in the first line of the subroutine definition. For example, following subroutine declaration

```
NL  Sub BranchingTest(intStart as
    Integer, intStop as Integer)
```

not only declares the variables **intStart** and **intStop** as integers, it also tells the subroutine to expect these two numbers to be passed as parameters.

To see how this works, create a new subroutine called **ParameterTest** based on **BranchingTest**.

27 Type the declaration statement above to create the **ParameterTest** subroutine.

28 Switch back to **BranchingTest** and highlight all the code except the **Sub** and **End Sub** statements

29 Cut and paste the code into the **ParameterTest** procedure.

To incorporate the parameters into **ParameterTest**, you will have to make the following modifications to the pasted code:

30 Replace `i = 1` with `i = intStart`.

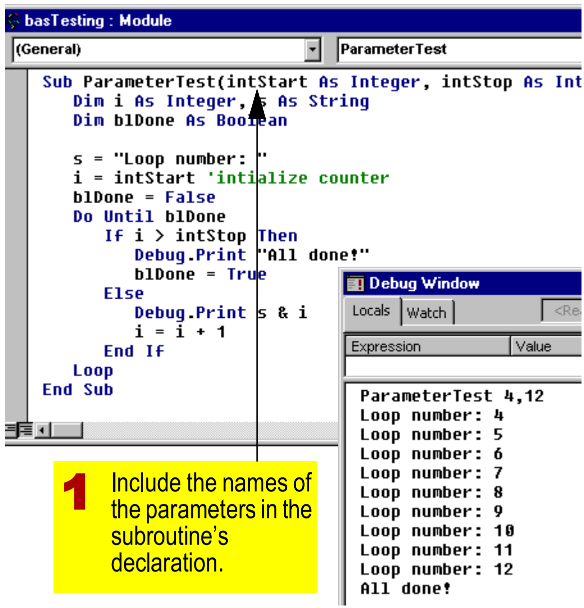
31 Replace `i > 10` with `i > intStop`.

32 Call the subroutine from the debug window by typing:

```
NL  ParameterTest 4, 12↵
```

The results are shown in Figure 18.8.

FIGURE 18.8: Create a parameterized subroutine.



If you prefer enclosing parameters in brackets, you have to use the

`Call <sub name>(parameter1, ..., parametern)` syntax, e.g.: `Call ParameterTest(4, 12)`. See Section 18.4.3 for more information on the use of brackets in VISUAL BASIC.

18.3.7 Creating a MinValue() function

In this section, you are going to create a user-defined function that returns the minimum of two numbers. Although most languages supply such a function, ACCESS does not (the **Min()** and **Max()** function in ACCESS are for use within SQL statements only).

33 Create a new module called **basUtilities**.

34 Type the following (on one line) to create a new function:

NL **Function MinValue(n1 as Single, n2 as Single) as Single**

The statement above defines a function called **MinValue** that returns a single-precision number. The function requires two single-precision numbers as parameters.

Since a function returns a value, the data type of the return value should be specified in the function declaration. Accordingly, the basic syntax of a function declaration is:

NL **Function** ***<function name>*(*parameter1* As *<data type>*, …, *parametern* As *<data type>*) As *<data type>***

The function returns a variable named ***<function name>***.

35 Type the following as the body of the function:

```
NL Function MinValue(n1 as Single, n2 as Single) as Single
NL   If n1 <= n2 Then
NL     MinValue = n1
NL   Else
NL     MinValue = n2
NL   End If
NL End Function
```

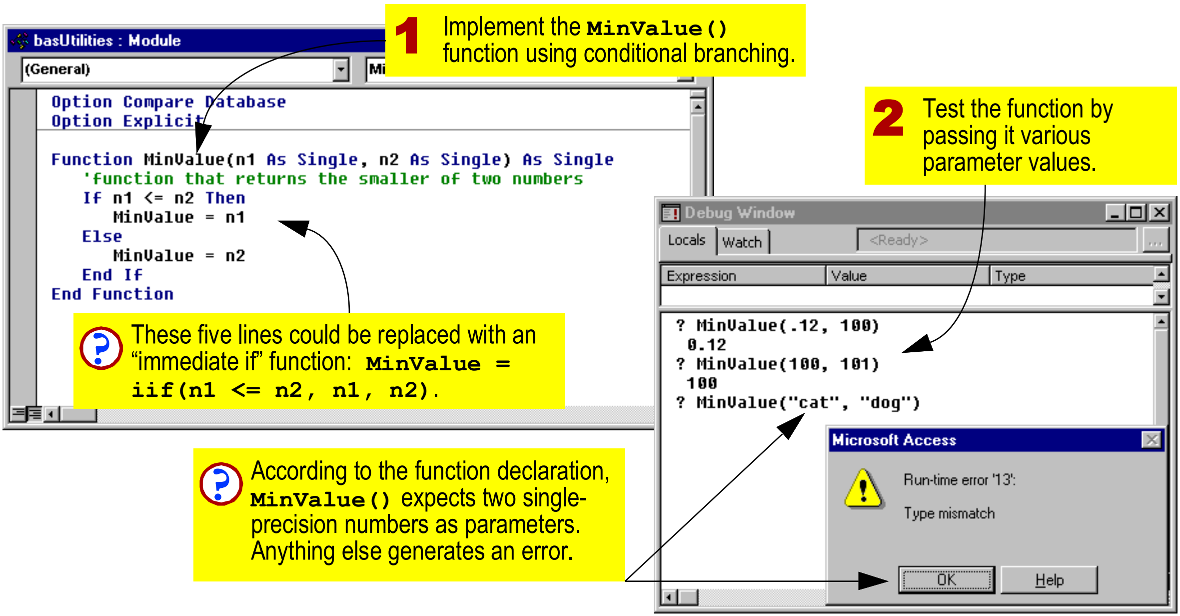
36 Test the function, as shown in Figure 18.9.

18.4 Discussion

18.4.1 The evolution of BASIC

An important thing to keep in mind when using VBA is that the BASIC language has been around since the mid 1960s and has evolved in a relatively uncontrolled, organic matter. Because of this, VISUAL BASIC lacks the purity and simplicity of a teaching language like PASCAL or a newcomer like JAVA.

FIGURE 18.9: Testing the ***MinValue()*** function.



For new programmers, the result is often frustration. There are multiple conflicting ways to accomplish the same task and some of the language constructs (e.g., **Dim**, **Goto**) are only understandable in their historical context.

The latest transformation in the language's evolution is quasi-object-orientation. Stand-alone statements are being de-emphasized in favor or methods that belong to objects. For example, recall the **Debug.Print** method in Section 18.3.4.

18.4.2 Interpreted and compiled languages

VBA is an **interpreted language**. In interpreted languages, each line of the program is interpreted (converted into machine language) and executed when the program is run. Other languages (such as C, PASCAL, FORTRAN, etc.) are **compiled**, meaning that the original (source) program is translated and saved into a file of machine language commands. The resulting executable file is run instead of the source code.

Predictably, compiled languages run much faster than interpreted languages (e.g., compiled C++ is generally ten times faster than interpreted JAVA). However, interpreted languages are generally easier to debug since program execution can be stopped at any point and the current line of source code viewed and manipulated.

18.4.3 Brackets and parameters

Recall that subroutines in VISUAL BASIC execute a batch of commands, whereas functions execute a batch of commands and return a single value. The distinction is important because, when it comes to passing parameters to procedures, VISUAL BASIC adopts an odd convention:

- If the procedure returns a value (i.e., it is a function), enclose the parameters in brackets, e.g., `MinValue(2, 7)`.
- If the procedure does not return a value (i.e., it is a subroutine), do not enclose the parameters in brackets, e.g.: `ParameterTest 4, 12`.

18.5 Application to the project

You will need a `MinValue()` function in Section 19.5 when you have to determine the default quantity to ship. Ensure you have created and tested the function in Section 18.3.7.

Lesson 19: Event-driven programming

19.1 Introduction:

In conventional programming, the sequence of operations for an application is determined by a central controlling program (e.g., a main procedure). In **event-driven** programming, the sequence of operations for an application is determined by the user's interaction with the application's interface (forms, menus, buttons, etc.).

The code for an event-driven application remains in the background until certain events happen, for example:

- when a value in a field is modified, a small data verification program is executed;
- when the user presses a button to indicate that the order entry is complete, the inventory update procedure is executed.
- when the user switches to a different record, the properties of certain controls on a form are set based on values in the new record.

Event-driven programming, graphical user interfaces (GUIs), and object-orientation are all related since forms (like those created in Lesson 13) and the graphical interface objects on the forms (like those created in Lesson 15) serve as the skeleton for the entire application.

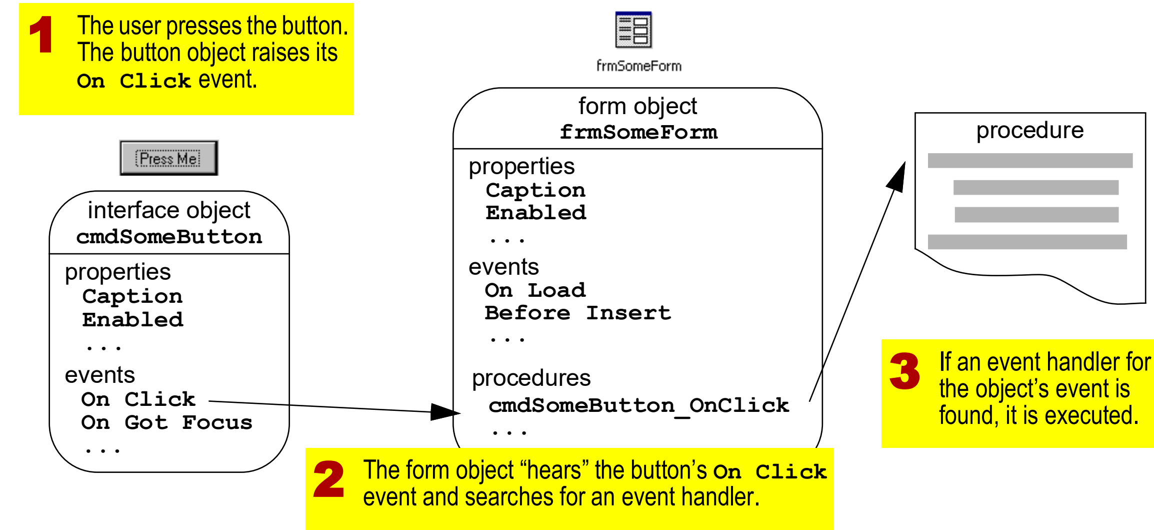
19.1.1 Listening and handling events

To create an event-driven application, the programmer creates small programs—called **event handlers**—and associates them with specific events raised by specific objects. In ACCESS, events are typically raised by objects on a form (or by the form itself) and handled by procedures defined within the **form module**.

ACCESS has different types of modules including stand-alone and form modules. In Lesson 18, you created a stand-alone module called `basUtilities`. For event-driven programming, you will use modules embedded within forms. The distinction is important because when you go looking for your event handlers, you will not find them in the **Modules** pane of the database window. Instead, the VISUAL BASIC code you write to handle events is saved with the form.

The relationship between interface objects, the form on which the interface objects reside, and procedures is shown in Figure 19.1.

FIGURE 19.1: In ACCESS, all events for objects on the form are handled by the form object.



19.1.2 Creating event handlers

The nice thing about event-driven programming is that the event handlers—that is, the procedures that are executed when the event is raised—can be very simple. In fact, you have already created an event-driven program without writing a single line of code: When the button you created in Section 17.3.5 is pressed, an action query is executed to update inventory levels. Event-driven programming can be that simple.

In practice, however, you will seldom rely on action queries alone to implement your event handlers. Since an action query in ACCESS can only perform one type of action, and since you typically have a number of actions that need to be performed, macros or VISUAL BASIC procedures are far more useful. But as you will see in this lesson, it is possible to use a combination of action queries and simple VISUAL BASIC statements to accomplish most of what you need to do.

19.1.3 The event-driven design cycle

To create an event-driven procedure, you need to answer two questions:

1. What has to happen?
2. When should it happen?

Once you have answered the first question ("what?"), you can create a procedure to execute the necessary steps. Once you know the answer to the second question ("when?"), you can associate the procedure with the correct event of the correct object.

Selecting the correct object and the correct event are often the most difficult part of creating an event-driven application. It is best to think about this carefully before you get too caught up in implementing the procedure.

19.1.4 VBA versus macros

There are two ways to create procedures within ACCESS:

- VISUAL BASIC FOR APPLICATIONS (VBA) code, or
- the proprietary macro language included with ACCESS.

The primary difference between VBA programming and macro programming is that the macro language consists of a handful of simple commands to accomplish many common tasks. In contrast, VBA is a general purpose programming language that is considerably harder to use, but far more flexible. Indeed, if you want your application to do something, chances are you can do it using VBA. Another nice thing about learning VBA is that most MICROSOFT applications (e.g., EXCEL, OUTLOOK) and some non-MICROSOFT products support VBA as a scripting or macro language.

A macro language is simply a language that consists of high-level (or "macro") commands. The term causes some confusion because macros in many older applications and some new ones (including MICROSOFT EXCEL) are programmed by recording keystrokes. As such, a common mistake it to consider the terms "macro" and "keystroke recorder" to be synonymous. However, the macro language in ACCESS provides no keystroke recording functionality.

19.1.5 Event-driven programming versus triggers

A trigger is usually defined as a procedure that is executed when a specific event in a table occurs (such as an insert, delete, or update). Triggers are useful for enforcing business rules throughout a database and applications based on the database.

In a client/server database product such as MICROSOFT SQL SERVER and ORACLE, triggers are SQL statements (similar to action queries in QBE) saved at the table level. ACCESS, in contrast, does not support table-level triggers. Instead, business rules in ACCESS applications must be enforced by event-driven procedures attached to form objects.

19.2 Learning objectives

- understand the basic concepts of event-driven programming
- create a button that executes several actions when pressed
- understand the difference between the ACCESS macro language and VBA
- use input from users to execute multi-step procedures
- understand how objects within ACCESS are named

19.3 Exercises

In these exercises, you will create event-driven programs using both the ACCESS macro language and VBA.

19.3.1 More flexible buttons

In Section 17.3.5, you used the button wizard to associate an action query with the `On Click` event of a button. This works fine, except that the execution of an action query results in two cryptic warning messages that users should not see. In this section, you are going to create an event-driven procedure that shows a single user-oriented message before running the query.

The answer to the "what?" question is:

1. Turn off the warnings so the dialog boxes do not pop up when the action query is executed.
2. Run the action query.
3. Display a custom message box.
4. Turn the warnings back on.

It is generally good programming practice to return the environment to its original state. Thus, if you turn warnings off in a procedure, you should turn them back on before leaving the procedure.

19.3.1.1 Using a macro to run an action query

You will start by implementing these four steps using ACCESS' macro language.

1 Select the **Macros** tab from the database window and press **New**. This brings up the macro editor shown in Figure 19.2.

2 Add the four macro actions shown in Figure 19.3. Note that the `OpenQuery` command is used to execute the action query, not “open” it.

3 Save the macro as `mcrProcessOrderDetails` and close it.

The answer to the “when?” question is straightforward: the macro should execute when the user presses the button on the order form. What you have to do at this point is tell ACCESS that the macro you just created should handle all `On Click` events raised by the button.

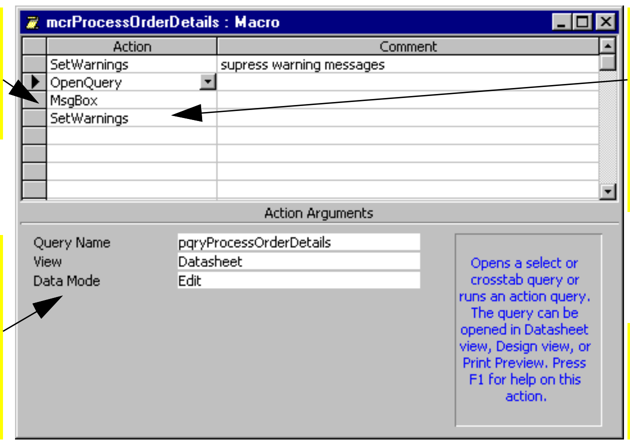
4 Open the order form in design mode and bring up the properties sheet for the command button you created in Section 17.3.5.

5 Although you will shortly delete the VBA procedure created by the wizard, press the

FIGURE 19.3: Create a macro that answers the “what?” question.

1 The message box should display a message such as, “The order has been processed.”

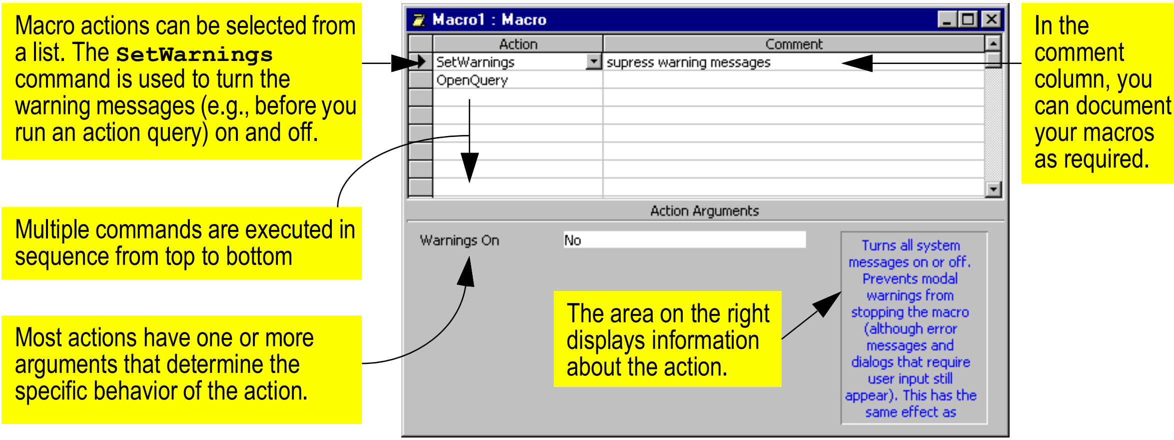
2 The `OpenQuery` method can be used to open a select query or run an action query. If an action query is used, the **View** and **Data Mode** properties are irrelevant.



3 Use the `SetWarnings` action a second time to turn warnings back on in the ACCESS environment.

4 Save the macro using the “mcr” prefix.

FIGURE 19.2: The macro editor in ACCESS.



builder to view the code, as shown in Figure 19.4.

As Figure 19.5 shows, the **On Click** event for the button is currently handled by an event procedure (VBA code) created by the button wizard.

The wizard tends to generate needlessly complex VBA code. The same procedure (without the error handling code) could be written using a single VBA statement.

6 Highlight the entire subroutine and delete it. You will replace the event procedure with a reference to your macro.

7 Close the module window. Note that the **On Click** event for the button is now empty.

8 Click the combo box arrow in the **On Click** property and select the macro you created to update orders. The procedure is shown in Figure 19.6.

FIGURE 19.4: The wizard creates a procedure to handle the button's ***On Click*** event.

1 Bring up the properties sheet for the button.

2 Click on the **On Click** event and press the builder to view the VBA code.

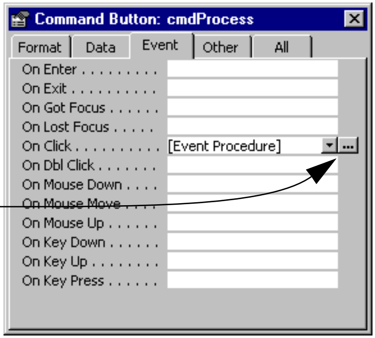
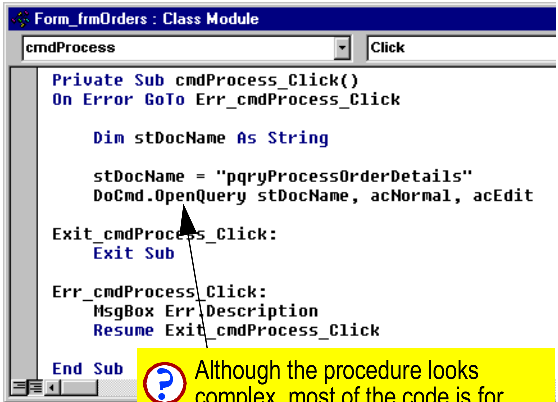


FIGURE 19.5: The VBA event handler created by the wizard.



Although the procedure looks complex, most of the code is for handling errors. The one line that does anything is a macro action called (surprise) **OpenQuery**.

9 Switch to form view and press the button. The message box you defined earlier should inform you that the order has been updated.

19.3.1.2 Using VBA to run an action query

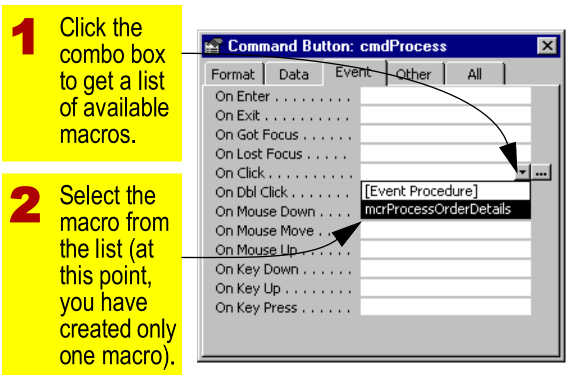
It is possible to achieve the same result using VBA.

10 Switch to design view and bring up the properties sheet for the **cmdProcess** button.

11 Click the combo box, but instead of selecting a macro as you did in Figure 19.6, select "event procedure".

12 Click the builder. The module associated with the order form will open. In it, you will find an empty subroutine called **cmdProcess_Click**, as shown in Figure 19.7.

FIGURE 19.6: Tell Access which macro to use to handle the button's ***On Click*** event.

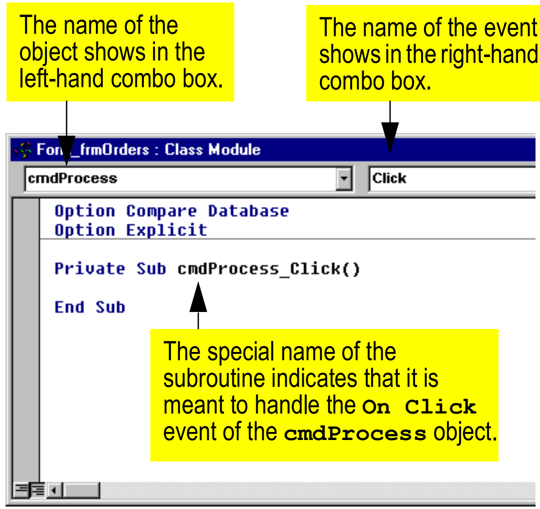


13 Define the subroutine as follows (see Figure 19.8):

```
NL  Private Sub cmdProcess_Click()
NL    DoCmd.SetWarnings False
NL    DoCmd.OpenQuery "pqryProcessOrderDetails"
NL    MsgBox "The order has been processed"
NL    DoCmd.SetWarnings True
NL  End Sub
```

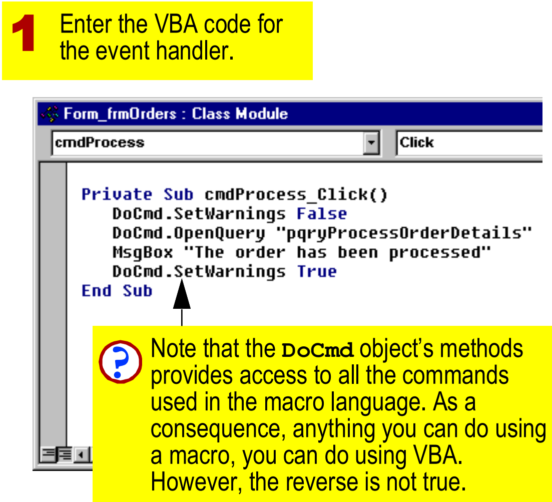
14 Close the module and switch to form view. When you press the button, you should get the same result as in Section 19.3.1.1.

FIGURE 19.7: The form module with an empty event handler.



While testing your procedures, you will probably process the same order multiple times and corrupted the `QtyOnHand` values in the `Products` table. You may wish to periodically run the rollback query you created in Section 17.3.2 to restore the correct inventory values.

FIGURE 19.8: Define the subroutine using VBA action statements.



19.3.2 Conditional procedures

For simple procedures such as the one you just created, the advantages of using VBA may not be obvious. After all, three of the four commands in Figure 19.8 are actually macro language commands. When it comes to looping and conditional branching, however, it is generally much easier to use VBA than to fiddle with macro language's looping and conditional branching constructs.

19.3.2.1 Motivation for a conditional procedure

As it now stands, there is nothing to prevent users from pressing the "process orders" button multiple times. However, since each order is physically shipped only once, it is clear that each order should be subtracted from `Products` table only once.

19.3.2.2 Updating processed status

The processed/not processed status of an order can be stored as a Boolean (yes/no) variable in the `Orders` table. However, it is not realistic to rely on the user to remember to change the `Processed` check box after updating an order.

However, the following line of code can be added to the subroutine to update the field automatically:

```
NL   Me.Processed.Value = True
```

In this statement, `Me` refers to the current form. Accordingly, `Me.Processed` refers to the checkbox control bound to the `Processed` field on the current form. `Value` is a property of the `Processed` control; however, since `Value` is also the default property, its inclusion is optional.

Strictly speaking, the syntax of this statement should be:

Me!Processed.Value = True. However, using the dot operator (.) instead of the bang operator (!) allows you to exploit MICROSOFT's "intellisense technology." See Section 19.4 for more information on techniques for naming objects in ACCESS.

15 While in form design view, bring up the properties sheet for the command button

16 Find the `On Click` event and press the builder to enter the VBA editor for the form module.

17 Modify your code to update the value of `Processed` once the update has occurred.

19.3.2.3 Using the status information

It is possible to use the `Processed` field for a particular order to skip the action query if the update has already been processed.

18 Add the following condition to your code, as shown in Figure 19.9:

```
Private Sub cmdProcess_Click()
  If Me!Processed Then
    MsgBox "This order has already been processed"
  Else
    DoCmd.SetWarnings False
    DoCmd.OpenQuery "pqryProcessOrderDetails"
    MsgBox "The order has been processed"
    DoCmd.SetWarnings True
  End If
End Sub
```

Notice that the condition part of the `If` statement does not contain an equals sign. With Boolean variables, the equals sign is implied. Thus, `If Me!Processed` is identical to `If Me!Processed = True` and `If` *`Not`* `Me!Processed` is the same as `If Me!Processed = False`.

19 Test the button to make sure the procedure is working correctly. The message box the user sees should depend on the value of the `Processed` field of the current record.

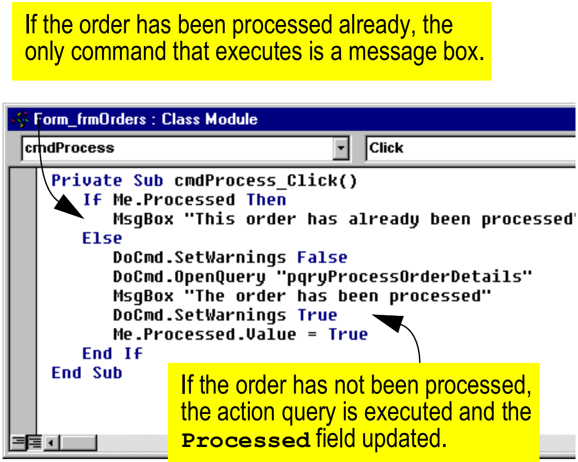
19.3.2.4 Protecting the status information

On final refinement: At this point, it is possible for the user to manually change the value of the `Processed` checkbox.

20 Set the Enabled property of `Processed` to No.

It is possible to use VBA to change the value of a disabled control. However, the ACCESS macro language cannot change the value of a control when it is disabled.

FIGURE 19.9: The completed subroutine for processing orders.



Consequently, to change the value of a disabled field with a macro, you must first enable the field, make the change, and then re-disable it. All the changes can be made using the `SetValue` macro action.

19.3.3 Using the AfterUpdate event

Consider the process of entering a single order detail:

1. A product is selected using a combo box that shows `ProductID` and a product description.
2. The quantity ordered for that particular product is entered. This information is supplied on the sales order sent by the customer.
3. The actual price of the product is entered. The `UnitPrice` for each product is stored in the `Products` table; however, the `ActualPrice` may be different (e.g., a promotion is being run or the customer in question qualifies for a discount).
4. The quantity to ship is determined. Since stockouts are possible, `QtyShipped` may be different than `QtyOrdered`. If `QtyOrdered` is greater than `QtyOnHand`, then the maximum quantity that can be shipped is `QtyOnHand`.

Once these four items of information are specified, the order detail is complete. To reduce the time the user spends entering each order detail, we should automate as much of the process as possible.

19.3.3.1 Getting the default price

When a new order detail record is created, the value of all numerical fields (**QtyOrdered**, **QtyShipped**, and **ActualPrice**) is set to zero by default. However, we do not want the default value for **ActualPrice** to be zero; we want it to be equal to the **UnitPrice** stored in the Products table. What is required, therefore, is a procedure that copies the default price into the **ActualPrice** field. To implement the procedure, we must answer two questions:

- What?: Set the value of **ActualPrice** to the correct value of **UnitPrice**.
- When? As soon as the **UnitPrice** of the product is known (i.e., as soon as **ProductID** is specified).

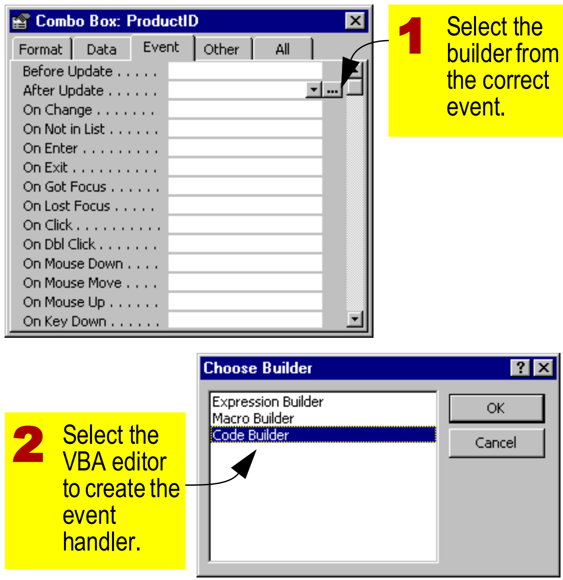
Note the choice of object and event. One might make the assumption that the **On Enter** event of the **ActualPrice** field is the best place for this procedure. However, there is no guarantee that the user will enter the **ActualPrice** field to raise the event. We do know for certain, however, that the user will have to enter a **ProductID**.

21 Open **sfrmOrderDetails** in design mode and bring up the properties sheet for the **ProductID** combo box.

22 Find the **After Update** event and click the builder button (...).

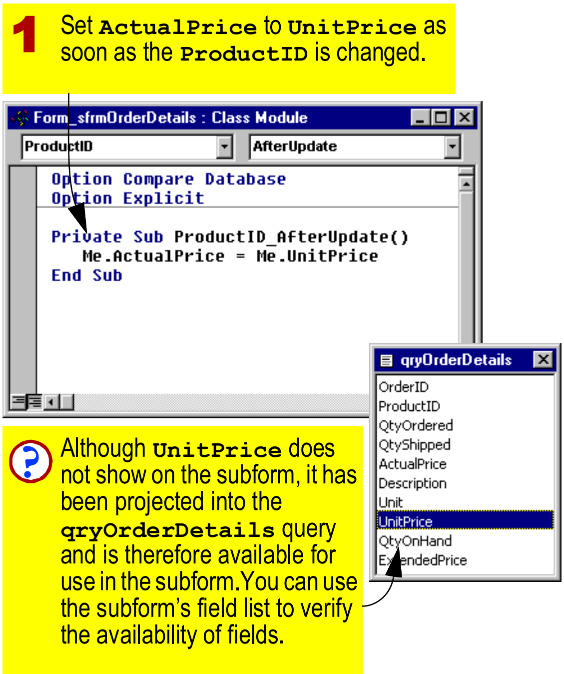
23 You will be given a choice of builders, as shown in Figure 19.10. Select **Code Builder** to get the VBA editor.

FIGURE 19.10: Invoke the VBA editor for the ***After Update*** event.



24 Enter the following assignment statement, as shown in Figure 19.11.

FIGURE 19.11: Create an event-driven procedure to set the default value of ***ActualPrice***.



NL `Me.ActualPrice = Me.UnitPrice`

25 Test the procedure. Whenever a `ProductID` is selected for a new or existing order detail, the `ActualPrice` field should be set to show the product's default price.

Although all fields have a **Default Value** property at the table level, there is no way to use this feature to set a default price. ACCESS permits only simple expressions such as constants (e.g., 0, "ea.") or predefined functions (e.g., `Now()`) to be used in the **Default Value** property.

19.4 Discussion: Object naming in ACCESS

Because the objects in ACCESS are organized into a hierarchy (know as DATA ACCESS OBJECTS, or DAO), a naming scheme is required to allow programmers to refer to specific objects. In previous lessons, you have used the expression builder (e.g., Figure 17.7) to navigate the DAO hierarchy graphically and select objects by clicking.

Unfortunately, VBA does not provide an expression builder. Consequently, anyone who wishes to write VBA needs to learn about the DAO hierarchy and the naming conventions used to navigate it. This section provides a brief overview of naming issues for top-level collections, control collections, and properties.

19.4.1 Top-level collections

A collection is like an egg carton: it is an object that exists solely to store and organize groups of similar objects (eggs). DAO has a number of top-level collections that contain one or more database objects of the same type. For example, there is a `Forms` collection that contains all the forms in a database. Top-level collections correspond (roughly) to the panes of the database window.

To refer to an item in a collection, you can use its `Item` property in combination with its numerical index. For example, ensure at least one form is open and type the following into the debug window:

NL `? Forms.Item(0).Name`

This statement prints the `Name` property of the "first" open form in the database. Of course, you seldom know the index numbers of items in a collection so using the `Item` property in concert with the object's name is more convenient (but redundant, at least in this example):

NL `? Forms.Item("frmOrders").Name`

To make the naming scheme even more confusing, VBA allows you to drop the `Item` part (it is assumed by default) and use the bang operator (!) as a syntactical shortcut. As a result, there are many different ways to refer to the same object:

NL `Forms.Item("frmOrders") 'long version`

NL `Forms("frmOrders") 'shortcut`

NL `Forms!frmOrders 'another shortcut`

19.4.2 Embedded controls collections

An object, like a form, can both belong to a collection (`Forms`) and contain other collections. For example, each form object in ACCESS contains a `Controls` collection that contains all the textboxes, combo boxes, checkboxes, subforms, and so on.

Since the `Controls` collection is the default "property" for `Form` objects, shortcuts are possible. For example, to refer to the `CustID` combo box on your order form, you could use any of the following:

NL `Forms.Item("frmOrders").Controls.Item("CustID") 'long version`

NL `Forms!frmOrders("CustID") 'shortcut`

NL `Forms!frmOrders!CustID 'another shortcut`

Since the `Forms` collection contains only open forms, you order form has to be open for you to test these examples.

19.4.3 Properties

Each object in ACCESS has predefined properties. Typically, properties are accessed using the dot

(.) operator. For example, each **Form** object has a **Name** property (e.g., **Forms!frmOrders.Name**) and each control has a **Value** property (e.g., **Forms!frmOrders!CustID.Value**). In addition, each object has a single default property; when no property is specified, the default property is used.

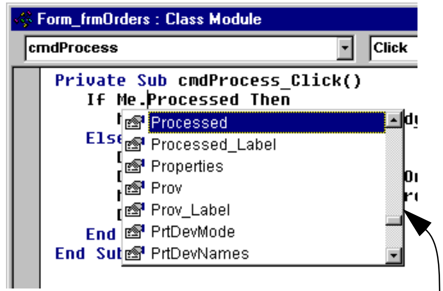
The on-line help system provides a summary of the properties, methods, and collections for each object.

19.4.4 Dot or bang?

According to the ACCESS help system, the bang operator (!) should be used to indicate that what follows is a user-defined item (i.e., an element of a collection). The dot operator (.) should be used to indicate that what follows is a property or something that is not user-defined.

Unfortunately, the "intellisense technology" that appeared in ACCESS version 8.0 does not do a very good job of recognizing the bang operator. For example, if you type "**Me.**" while editing an event handler, the editor will provide you with a list of form properties and objects on the form, as shown in Figure 19.12. However, if you type "**Me!**" (to indicate that you are only interested in objects on the form), intellisense provides no help. As such, it is often easier to use the dot operator, even though MICROSOFT discourages it.

FIGURE 19.12: Using "intellisense technology" to finish the VBA statement.



"Intellisense technology" is a feature introduced in version 8.0 that helps complete VBA statements. In this example, a list is shown for the **Me** object (the current form). It shows all the properties for the form as well as all the objects in the control collection.

19.5 Application to the project

Now that you have an event-driven procedure to set the default price for products, you know all you need to know to implement a similar procedure for suggesting a quantity to ship:

- What? Set the **QtyShipped** field for an order detail to the minimum of **QtyOrdered** and **QtyOnHand**.

- When? As soon as `QtyOrdered` and `QtyOnHand` are known.

26 Create an event-driven procedure for setting the default quantity to ship. The user should be able to override this value.

HINT: You may want to consider using the `MinValue()` function you created in Section 18.3.7 to implement this feature.

27 Add a second button to your order form to show the invoice you created in Section 16.4 in “print preview” view.

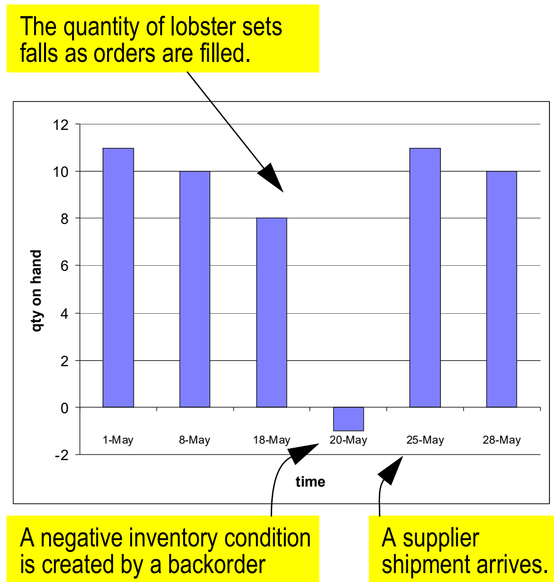
20.1 Introduction: Inflows and outflows of product over time

Your system—as it currently stands—is capable of subtracting items from inventory as you make shipments to your customers. However, it is clear that at some point, you must replenish your inventory with shipments from your suppliers. In order to have an accurate picture of your inventory level, you need to account for both incoming and outgoing flows of products.[1]

Consider the inventory level of a product over time. For example, the `QtyOnHand` value for ProductID = 74 6881 (“lobster set”) is shown in Figure 20.1. The initial inventory is 11 items; however, as customer orders are filled, the inventory drops and as supplier shipments are received, the inventory rises.

In this lesson, you are going to enhance your system to deal with inflows of product, such as the shipment from one of your suppliers on May 25th. The issue of what to do when inventory drops below zero (i.e., backorders) is addressed in Lesson 21.

FIGURE 20.1: Changes in inventory level over time for product 74 6881



1 In the real world, you would also have to account for “shrinkage”—loss of product due to breakage, theft, and so on. In the face of inventory shrinkage, the only way to keep your inventory record accurate is to periodically count the physical items in your warehouse and reconcile the physical count with the inventory levels in the database. To keep things simple in this lesson, we ignore shrinkage.

20.2 Learning objectives

- gain more experience creating tables, relationships, and queries
- gain more experience building multi-part forms
- create an event-driven procedure to update inventory levels on receipt of a shipment
- understand the difference between inventory tracking and inventory management

20.3 Exercises

A supplier shipment is an event that causes changes in a status field (`QtyOnHand` in the `Products` table). As such, you will want to record the details of "shipment" events in the same way that you record details of "customer order" events.

Given the fixed costs associated with shipping physical goods (e.g., a truck, a driver, and so on), a shipment is typically a nested transaction—that is, each shipment transaction consists of many shipment detail transactions.

Since you already have experience with nested transactions from `Orders` and `OrderDetails`, implementing the tables, forms, and event-driven procedures for recording supplier shipments uses skills you already possess.

20.3.1 Creating tables

1 Create a `Shipments` table to record the critical attributes of a shipment event. The table should include the date of the shipment, the ID of the supplier, and whether the shipment has already been processed.

Think carefully about an appropriate primary key for the `Shipments` table. Although you could use a concatenated primary key such as `ShipDate` + `ShipTime` + `SupplierName`, it is probably a better idea to use a **surrogate primary key**, such as `ShipmentID`.

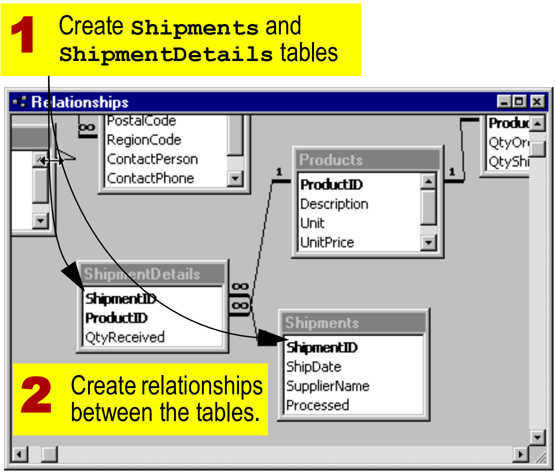
2 Add a field named `ShipmentID` to your `Shipments` table and designate it as the primary key. It should be an AutoNumber data type.

Since each supplier shipment is likely to contain more than one item, you need a `ShipmentDetails` table.

3 Create a `ShipmentDetails` table. The foreign key corresponds to the `ShipmentID` field created above (recall Section 5.4.2 when deciding on the correct data type for `ShipmentDetails.ShipmentID`).

4 Create relationships between the new tables and existing tables, as shown in Figure 20.2.

FIGURE 20.2: Relationships between the new shipment tables and the existing tables.



5 Ensure field properties such as **Caption** and **Default Value** are set appropriately for both tables.

6 Populate the `Suppliers` table you created in Section 12.3.4 with a small number of records. You can either make up the supplier names or use the names shown on the in the list of backorders and shipments (`BackShip.pdf`) in the project package.

20.3.2 Getting the right information for the shipment details subform

Your form for recording shipments is going to be very similar to the form shown in Figure 14.1 for recording orders. To make the details of the shipment more readable, you will want to base the subform on a join query.

7 Create a new query called `qryShipmentDetails` based on the `ShipmentDetails` and `Products` tables.

8 Project the asterisk from the `ShipmentDetails` table and the `Description` field from the `Products` table.

9 Save and close the query.

20.3.3 Creating a form for recording shipments

10 Use the form wizard to create a new columnar form based on the `Shipments` table and save the form as `frmShipments`. If you are having difficulty remembering how to create a main form, review Section 14.3.1.

11 Create a new tabular form based on your `qryShipmentDetails` query and save the form as `sfrmShipmentDetails`. If you are having difficulty remembering how to create a subform, review Section 14.3.2.

Since the form is based on a query, you should bring up the properties sheet for the form and ensure that the **RecordSource** property is `qryShipmentDetails`, not an *ad hoc* SQL statement.

12 Replace the `SupplierID` textbox on the `frmShipments` form with a bound combo box. The combo box should only show the suppliers' names (review Section 15.3.2 as required).

13 Replace the `ProductID` textbox on the `sfrmShipmentDetails` form with a bound combo box. The combo box should show the `ProductID` and `ProductName` fields.

14 Drag the subform onto the main form (review Section 14.3.3 as required).

Ensure the master and child link properties for the subform control are set properly. Otherwise, the form and subform will not be synchronized.

20.3.4 Processing the shipment

Recall the procedure implemented in Lesson 19 for processing an order:

1. The user enters the order information and the order details from the faxed-in order.
2. When the user is satisfied that the order has been entered correctly, she presses a button on the order form to process the order.
3. Pressing the button raises an **On Click** event, which causes a few lines of VBA code to be executed.
4. The VBA code executes a parameter action query that decrements the inventory the appropriate amount for each item in the order.

The procedure for processing a shipment is identical, except that items are *added* to inventory.

15 Create a parameter action query called `pqryProcessShipmentDetails` to add the amount stored in `QtyReceived` to `QtyOnHand` (review Section 17.3.3 as required).

Remember to use the value of `ShipmentID` on the main shipment form as a parameter. If you forget to do this, all shipment details (including those associated with other shipments) will be processed against the **`Products`** table.

16 Open **`frmShipments`** in edit mode and turn the control wizard off (recall Figure 17.9). This permits you to create the VBA code for the button from scratch.

Alternatively, you can leave the control wizard on and press the **Cancel** button as soon as the wizard appears.

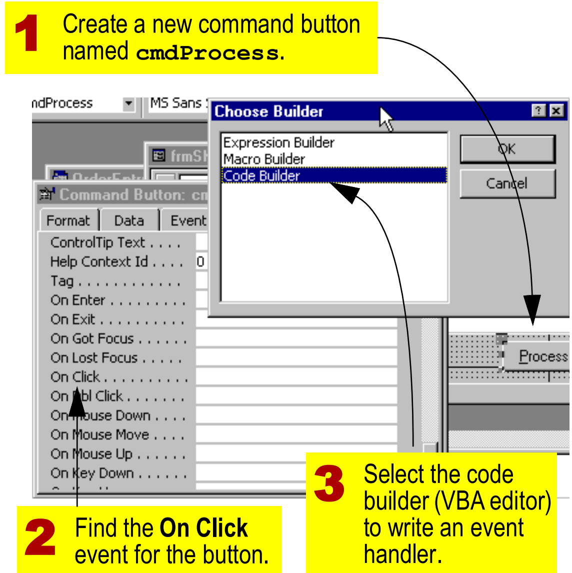
17 Add a button to the main shipment form and change its **Name** property to **`cmdProcess`**.

18 Bring up the properties sheet for the button, move to the **On Click** event, and invoke the VBA editor by selecting "event procedure", as shown in Figure 20.3.

19 Enter the following code into the VBA editor (this is similar to the code in Figure 19.9):

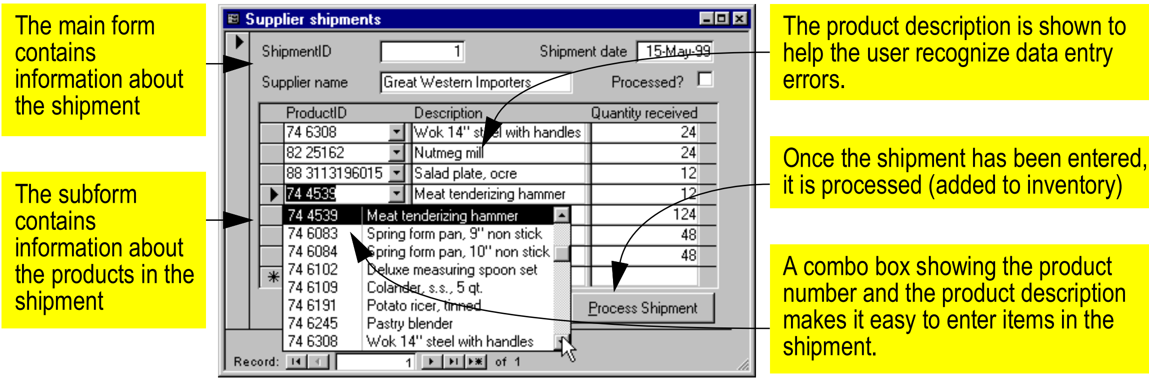
```
NL  If Me!Processed Then
NL    MsgBox "This shipment has already
      been processed"
NL  Else
NL    DoCmd.SetWarnings False
NL    DoCmd.OpenQuery
      "pqryProcessShipmentDetails"
NL    MsgBox "The shipment has been
      processed"
NL    DoCmd.SetWarnings True
NL    Me!Processed = True
NL  End If
```

FIGURE 20.3: Create an event handler for the **On Click** event of the new button.



20 Close the module and test the update button. The final shipment form is shown in Figure 20.4.

FIGURE 20.4: A form for recording supplier shipments.



Remember, you can use your rollback feature to return the `QtyOnHand` values for all products to their original values.

20.4 Discussion: Tracking versus optimizing

Determining how much to order (the reorder quantity) and when to order (the reorder point) requires additional information not explicitly stored in your database:

- the expected lead time from your suppliers (i.e., how long after placing an order do you expect to receive the shipment)
- the fixed cost of placing an order
- the cost of holding inventory
- the cost of stocking out

In addition, you need to be able to accurately forecast the demand for each product during its lead time. Although you do not store demand forecasts in your database explicitly, information from past orders (which are stored in the database) could be a starting point for forecasting methods such as seasonally-adjusted moving averages, and so on.

In principal, the inventory levels for each product can be monitored and a supplier order can be automatically generated whenever the reorder points are reached. The mathematical techniques for determining the optimal inventory management policy can be found in any production management text and the data requirements are not particularly onerous. However, you are not expected to implement this functionality in this lesson.

20.5 Application to the assignment

Once the shipment form is working, you should take a few moments to refine it.

21 Lock and disable the `Processed` check box.

22 Lock and disable the product description in the subform.

23 Ensure the tab order is correct for both the main form and the subform.

Lesson 21: Managing backorders

21.1 Introduction: When customers do not know your stock levels

An important issue that we have not addressed to this point is backorders. When customers fax you an order, they have no idea what your stock levels are. As such, it is possible that you will receive orders for product that you do not have. In this lesson, you will implement a simple strategy for managing backorders.

21.2 Learning objectives

- identify different strategies for dealing with backorders
- write a more complex VBA procedure that uses the DATA ACCESS OBJECTS object model
- gain experience with the `DLookUp()` function
- understand why you cannot create a relationship between the `OrderDetails` and the `BackOrders` tables

21.3 Exercises

21.3.1 Updating backorders

There are at least three basic strategies for dealing with the stockout problem:

1. ignore backordered items (this makes the least business sense);
2. keep a simple list of which customers requires which items and ship the items when you have them in stock;
3. build a more elaborate backorder fulfillment system to allocate scarce products according to some priority scheme (e.g., best customers first, oldest outstanding backorder first, etc.)

In this section, you are going to implement the simple list-based approach in (2) above: when a customer orders a product that is not in stock, the missing items are added to a list. You attempt to ship any items that you “owe” the customer on the customer’s next order. To keep it simple, we are assuming that customers do not cancel backorders.

21.3.1.1 A simple list of backorders

1 Create a new table called `BackOrders`. It should consists of three fields: `CustID`, `ProductID`, and `QtyOnBO`. The data type for `QtyOnBO` should be Numeric (Integer).

2 Create a concatenated primary key using the `CustID` and `ProductID` fields.

3 Populate the `BackOrders` table using the information in a file called `BackShip.pdf` in the project package. The list of backorders in the file is the initial state of the `BackOrders` table at the start of the development project.

The `BackOrders` table used here is simply a running list of customers and products. When a customer orders a product that is not in stock, one of two things must occur:

1. If the particular customer-product combination is already in the `BackOrders` table, then the `QtyOnBO` is incremented the appropriate amount.
2. If the particular customer-product combination is not already in the `BackOrders` table, then a new record is added.

To illustrate, consider the following scenario: THE CHEF'S CHOICE orders 6 meat hammers and you have no meat hammers in stock. Since the customer already has an outstanding backorder for the product in question, you must change the `QtyOnBO` for (`CustID` = 5, `ProductID` = "74 4539") from 5 to 11. Conversely, if a different customer without an outstanding backorder for meat hammers (say SAM'S STOCK POT) orders the item, then a new record must be added to the `BackOrders` table.

To manage backorders, you must also have a procedure for decrementing `QtyOnBO` when a backorder is filled or partially filled. In the case in which the `QtyOnBO` field reaches zero, you have two choices:

1. write a routine to remove the record, or
2. leave the zero-valued customer-product combination in the table.

In the interest of simplicity, we are going to opt for the latter approach and leave zero-valued entries in the `BackOrders` table.

In the worst case, the `BackOrders` table will eventually contain one record for every unique combination of `CustID` and `ProductID`. In our small example, this is $54 \times 5 = 270$ records.

21.3.1.2 The add or edit problem

Updating the `Products` table when processing an order is straightforward: each order detail corresponds to a particular product and thus a simple update query can be used. However, the method we are using for managing backorder requires an update procedure that is slightly more complex. Specifically, an action query cannot be used since, in some circumstances, records have to be changed and in others, records have to be added. VBA code must be used to implement the necessary functionality because action queries are not designed for this type of decision logic.

This is not to suggest that we are going to abandon queries altogether. Since queries are much easy to write than code, you will use a parameter select query to do as much work as possible.

21.3.1.3 Determining backorder changes

4 Create a new select query called `pqryBackOrderChanges` based on the `OrderDetails` table.

5 Project the `ProductID` field and create a calculated field called `BOChange` to show the change to the `BackOrders` table implied by the `QtyOrdered` and `QtyShipped` fields.

This requires some thought: if a customer orders 15 of a certain product (`QtyOrdered` = 15) and you only ship 11 (`QtyShipped` = 11) then this implies that you owe the customer 4. That is, the change to `QtyOnBO` for that particular customer and that particular product should be +4.

In contrast, if the customer orders 15 and you ship 30, then this implies that an outstanding backorder has been filled or partially filled (i.e., `BOChange` = -15).[1]

6 Add a parameter to the query so that it shows a single order only. Name the parameter `[pOrderID]`.

This query differs in two respects from the parameter queries you have created so far. First, it is a *select* query, not an *action* query. The query will not update the `BackOrders` table by itself; instead, it will be used by a VBA procedure to perform the update. Second, the parameter does not refer to a field on a form. As you will see in a moment, the parameter is set by VBA code before the query is used.

7 Add a condition to the `BOChange` field to ensure that non-changes are not included in the results set. That is, if `QtyShipped`

[1] This logic assumes that the only reason that you ever overship is to fill a backorder (i.e, you do not make shipping mistakes). To keep things simple, let us assume that this assumption is valid.

exactly equals `QtyOrdered`, then no change to the `BackOrders` table is required for that particular order detail.

8 Ensure your query is similar to the one shown in Figure 21.1 and that you understand what the query does.

9 Save and close the query.

21.3.1.4 Writing a backorder update routine

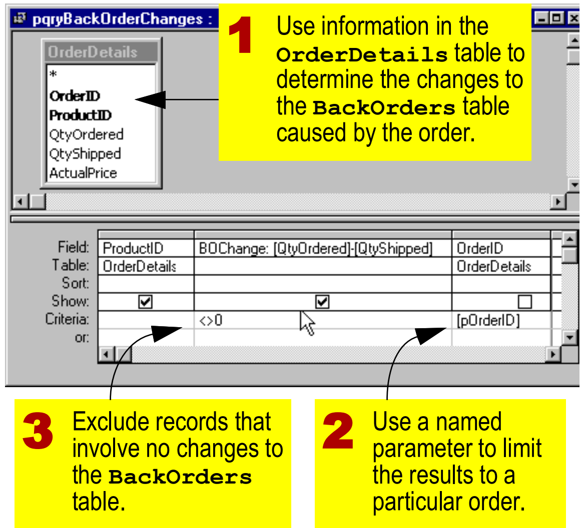
In this section, you will write a VBA procedure for updating the `BackOrders` table. At this point you are not expected to fully understand the code. In the discussion in Section 21.4.2, elements of the DATA ACCESS OBJECT object model are described in greater detail and the trickier parts of the code are explained.

10 Switch to the modules pane of the database window and create a new module. Save the module as `basBackOrders`.

11 Create a new subroutine called `UpdateBackOrders`. The subroutine should accept two parameters: the `CustomerID` of the customer who placed the order and the `OrderID` of the order.

12 Write a subroutine that loops through the backorder changes and decides how to

FIGURE 21.1: A parameterized select query for updating the ***BackOrders*** table.



update the `BackOrders` table. The code for the subroutine is shown below.

2K If you are using ACCESS 2000, you will have to create a reference to the DATA ACCESS OBJECTS (DAO) library before the code will run. See Section 21.4.3 for additional information.

```
NL  Public Sub
    UpdateBackOrders(lngCustID As Long,
    lngOrderID As Long)
NL    'routine to process changes to
    backorders associated with an order
NL    Dim dbCurr As Database
NL    Dim rsChanges As Recordset,
    rsBackOrders As Recordset
NL    Dim qdf As QueryDef
NL    Set dbCurr =
    DBEngine.Workspaces(0).Databases(0)
NL    'create a recordset of existing
    backorders
NL    Set rsBackOrders =
    dbCurr.OpenRecordset(“BackOrders”,
    dbOpenDynaset)
NL    'create a recordset containing
    the changes for the order
NL    Set qdf =
    dbCurr.QueryDefs!pqryBackOrderChang
    es
NL    With qdf
NL      .Parameters!pOrderID =
    lngOrderID
NL      Set rsChanges
    = .OpenRecordset(dbOpenSnapshot)
NL    End With
NL    Set qdf = Nothing
NL    'loop through the changes and
    decide whether to edit or append
    backorder records
NL    Do Until rsChanges.EOF
NL      rsBackOrders.FindFirst “CustID
    = ” &
    lngCustID & “ AND ProductID = ‘” &
    rsChanges!ProductID
    & “’”
NL      If rsBackOrders.NoMatch Then
NL        'customer-product combination
    does not exist in the backorders
    list
NL        With rsBackOrders
NL          .AddNew
NL          !CustID = lngCustID
NL          !ProductID =
    rsChanges!ProductID
NL          !QtyOnBO =
    rsChanges!BOChange
NL          .Update
NL        End With
NL      Else
NL        'customer-product combination
    already exists in the backorders
    list
NL        With rsBackOrders
NL          .Edit
NL          !QtyOnBO = !QtyOnBO +
    rsChanges!BOChange
NL          .Update
NL        End With
NL      End If
NL      rsChanges.MoveNext
NL    Loop
NL    rsChanges.Close
NL    rsBackOrders.Close
```

```
NL End Sub
```

13 Save the subroutine and exit the VBA editor.

21.3.1.5 Updating the trigger

14 Open **frmOrders** in edit mode and bring up the properties sheet for the **cmdProcess** button.

15 Move to the **On Click** event and edit the VBA procedure you created in Section 19.3.1.2 to handle the event.

16 Add code to call the new subroutine when the order is processed:

```
NL Private Sub cmdProcess_Click()
NL   If Me!Processed Then
NL     MsgBox "This order has already
   been processed"
NL   Else
NL     DoCmd.SetWarnings False
NL     DoCmd.OpenQuery
   "pqryProcessOrderDetails"
NL     UpdateBackOrders Me!CustID,
   Me!OrderID
NL     MsgBox "The order has been
   processed"
NL     DoCmd.SetWarnings True
NL   End If
```

17 Save and exit the module.

Now, whenever the button is pressed to process an order, the **UpdateBackOrders** subroutine is called. The current values of **CustomerID** and **OrderID** are passed as parameters to the subroutine.

The subroutine created here depends only on the **OrderDetails** and **BackOrders** tables. It is completely independent of the form (except that a button on the form is used to call the subroutine).

21.3.2 An introduction to the DLookUp() function

At this point, you have a means of keeping the **BackOrders** table up to date. As orders are processed, items are added or subtracted from the list as required. However, you have not yet incorporated the information from the **BackOrders** table into the decision making logic of the order entry system.

In Section 19.5, you were asked to create a feature to automatically suggest a quantity to ship based on the quantity ordered by the customer and the quantity on hand. In this section, you are going to extend this feature to include backorders.

To illustrate the basic issues, assume the **BackOrders** table contains an outstanding backorder from SAM'S STOCK POT for 5 lobster sets. If, in his next order, Sam orders a dozen more lobster sets, then you should ship him 5 + 12 = 17 (if you have that many in stock).

Even if Sam does not order lobster sets in his current order, the system should automatically remind you that you owe him certain items. Implementation of this feature is left as an exercise in Section 21.5.

21.3.2.1 DLookUp() basics

For each item that a customer orders, you need to determine whether there is an outstanding backorder for that particular customer-product combination. Although you might be tempted to simply join the `BackOrders` table with the other tables in your `qryOrderDetails` query to retrieve this information, this approach does not work and will never work (see Section 21.4.5).

Thus, you are left with the following situation: you know the `CustID` of the customer placing the order and the `ProductID` of the product being added to the order. What you want to do is "look up" the quantity on backorder (if any) for the customer-product combination. However, you cannot accomplish this using a join query. In such situations, a built-in function called `DLookUp()` is very useful.

Many ACCESS neophytes find use of the `DLookUp()` function to be a bit tricky at first. As such, we are going to step through its use incrementally. The most important thing to keep in mind is that the `DLookUp()` function is simply a stand-alone SQL query that returns a single field from a single record.

To illustrate, consider the following scenario: you wish to find the postal code of the company named "ROSCH DRY GOODS INC." in your `Customers` table. To get this information, you could create a simple SQL query, much like those you created in Lesson 12:

```
NL  SELECT PostalCode
NL  FROM Customers
NL  WHERE CustName = "Rosch Dry Goods
    Inc."
```

Note that the query returns a single field (`PostalCode`) from a single record (the customer with the unique name "Rosch Dry Goods Inc."). In addition, note that the value for `CustName` in the WHERE clause must be in quotation marks, since it is a literal string. If a non-string field such as `CustID` is used in the WHERE clause, quotation marks are not used:

```
NL  SELECT PostalCode
NL  FROM Customers
NL  WHERE CustID = 3
```

The `DLookUp()` function simply allows you to execute queries such as these from anywhere in

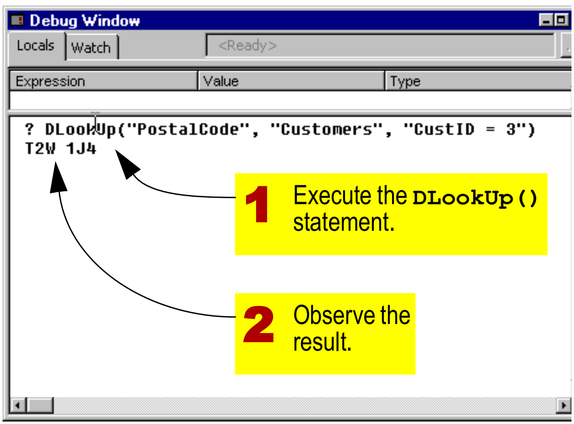
ACCESS, such as within a VBA program or a calculated field.

In the following exercises, you are going to experiment with the `DLookUp()` function within the debug window.

18 Press **Ctrl-G** to bring up the debug window.

19 Execute the following `DLookUp()` function call, as shown in Figure 21.2:

FIGURE 21.2: Execute a simple *`DLookUp()`* function from within the debug window.



NL `? DLookUp("PostalCode", "Customers", "CustID = 3")`

The `DLookUp()` function requires three arguments:

1. **Field name (or "expression"):** the first argument contains the name of the field containing the desired data. This argument corresponds to the field name after the SELECT keyword in the SQL statements above.

Note that like all arguments in the `DLookUp()` function, the name of the field is passed as a string (within quotation marks).

2. **"Domain" name** — the domain is simply the name of the table or query containing the desired data. This argument corresponds to the data source following the FROM keyword in the SQL statements above.
3. **Criteria** — the condition in the criteria argument narrows the search to a single record in the domain.This argument corresponds to the WHERE clause in the SQL statements above.

The first two arguments of the `DLookUp()` function are straightforward. Most of the difficulty with the function lies with the criteria argument (which is called the WHERE clause throughout the remainder of this lesson). Specifically, trouble occurs in three situations:

- when the WHERE clause contains a literal string and quotation marks have to nested;
- when the WHERE clause contains one or more variables; and,
- when the two situations above are combined.

Each of these situations is addressed in the sections below.

21.3.2.2 Nested quotation marks

The WHERE clause for the `DLookUp()` function must be a string—i.e., it must be enclosed in quotation marks. However, what happens when the WHERE clause already contains quotation marks? For example:

```
NL SELECT … FROM …
NL WHERE CustName = "Rosch Dry Goods
   Inc."
```

20 In the debug window, type the following (incorrect) `DLookUp()` statement:

```
NL ? DLookUp("PostalCode",
   "Customers", "CustName = "Rosch Dry
   Goods Inc."")
```

The reason you get an error is because the VBA interpreter cannot make sense of the nested quotation marks. To get around the problem, you use the same convention that is used in written English—single quotation marks are used to denote the quotation inside of the quotation.

To illustrate, consider the following sentence:

"When I asked him, he said, 'I do not know.'"

If you understand the use of quotation marks in the sentence above, you should have no problem understanding their use in the `DLookUp()` function.

21 In the debug window, type the corrected version of the `DLookUp()` statement:

```
NL ? DLookUp("PostalCode",
   "Customers", "CustName = 'Rosch Dry
   Goods Inc.'")
```

Unlike many other languages, you cannot arbitrarily split a line of code in VBA—i.e., VBA requires one statement per line and one line per statement. However, you can use the "continuation character" to tell the interpreter to treat two or more lines as one. In VBA, the continuation character is the underscore (_) and it must be the last character on the split line. For example, the following code is treated as a single line:

```
NL ? DLookUp("PostalCode", _
NL     "Customers", _
NL     "CustName = _
NL     'Rosch Dry Goods Inc.'")
```

21.3.2.3 DLookUp() functions with variables in the WHERE clause

In the same way that parameter queries greatly enhance the flexibility of queries, variables in the WHERE clause (or any other argument) greatly enhance the flexibility of the **DLookUp()** function.

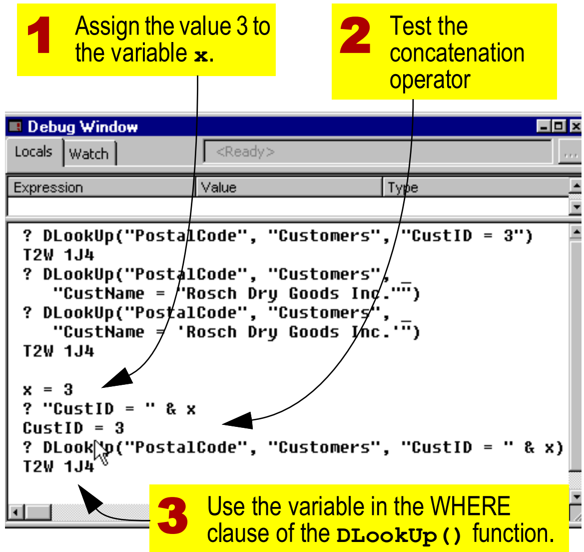
22 Type the following VBA code into the debug window, as shown in Figure 21.3:

```
NL  x = 3
NL  ? "CustID = " & 3
NL  ? DLookUp("PostalCode",
    "Customers", "CustID = " & x)
```

In the first line of code, you assign the value 3 to a variable **x**. The second line simply reminds you of how an expression containing the concatenation operator (&) is executed. In this case, the *value* of **x** is simply appended to the end of the "**CustID** = " string. Finally, in the third line, you use an expression containing a variable in the WHERE clause.

Since **x** falls outside of the quotation marks in the WHERE clause, it is treated as a variable, not a character of text. The VBA interpreter replaces variables with their current values before calling the function.

FIGURE 21.3: Use a variable to replace a literal value for ***CustID*** in a ***DLookUp()*** function.



21.3.2.4 Using text variables in the WHERE clause

In this section, you are going to put all the pieces of the **DLookUp()** function together.

23 Type the following into the debug window and observe the outcome, as shown in Figure 21.4.

NL `x = "Rosch Dry Goods Inc."`

NL `? "CustName = " & x`

NL `? "CustName = '" & x "'"`

NL `? DLookUp("PostalCode", "Customers", "CustName = '" & x & "'")`

Remember that what you ultimately want to pass to the **`DLookUp()`** function is the WHERE argument you created in Section 21.3.2.2: **`"CustName = 'Rosch Dry Goods Inc.'"`**

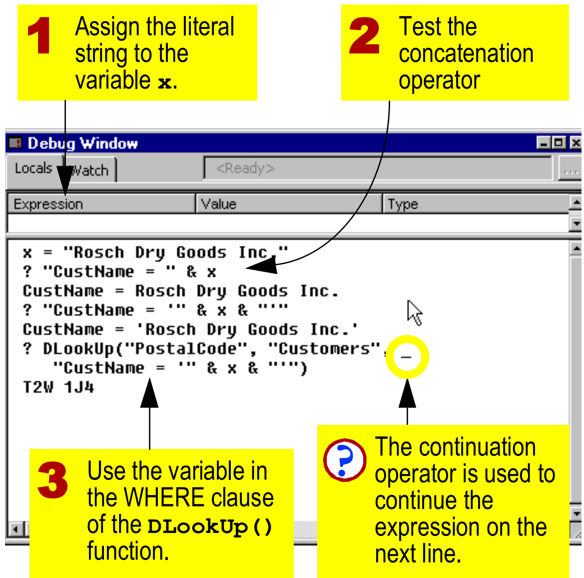
To get this result, you must include the single quotation marks in the string expression.

It is very important that you understand the outcomes in Figure 21.4 before continuing with the remainder of the exercises in this lesson.

21.3.3 Using DLookUp() to get the number of items on backorder

Now that you have honed your **`DLookUp()`** skills in the debug window, it is time to do something useful. In this section, you will create a number of calculated fields in a query to get the backordered quantity associated with each product ordered by the customer.

FIGURE 21.4: Use a variable to replace a literal value for ***`CustName`*** in a ***`DLookUp()`*** function.



The calculations in this section could also be done on the order form instead of within the **`qryOrderDetails`** query. However, it is good policy to push calculations to the lowest level in ACCESS. In this way, the query can be tested independently of the form (tracking down errors on forms and subforms is more complex).

21.3.3.1 Preliminaries

24 Open **qryOrderDetails** in edit mode.

The first thing to recognize when faced with the lookup task is that the primary key of the **BackOrders** table contains **CustID**; however the **qryOrderDetails** query does not include any information on the customer. Fortunately, the query does contain the **OrderID**, which is sufficient to determine the **CustID** using a simple join.

25 Add the **Orders** table to the query and ensure the relationship lines appear correctly, as shown in Figure 21.5.

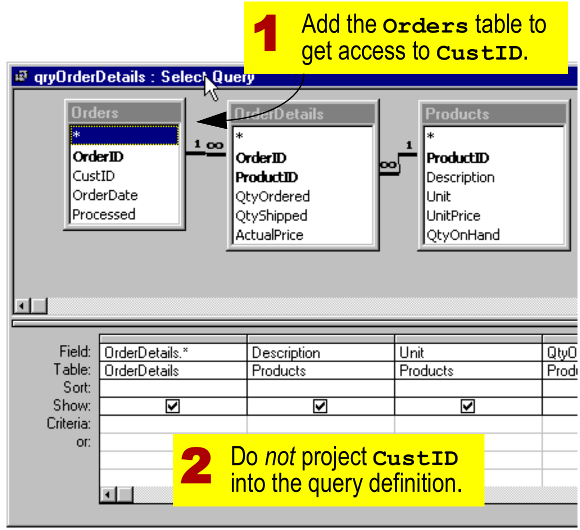
The **CustID** field does not have to be projected into the query definition in order to be "available" within the query.

21.3.3.2 Getting the WHERE clause right

26 Define a new calculated field called **BOWhere**:

```
NL  BOWhere: "CustID = " & [CustID]
```

FIGURE 21.5: Add the ***Orders*** table to the ***qryOrderDetails*** query.



We are going to break the **DLookUp()** function into a number of smaller pieces. This is done to facilitate troubleshooting if things go wrong.

The **BOWhere** calculated field is very similar to the expression you created in Section 21.3.2.3. The difference is that a variable **x** is replaced by the name of a field, **CustID**. Since **CustID** falls outside of the quotation marks, it is replaced by the field's current value when the query executes.

Do not be confused by the use of the square brackets in the definition of `BOWhere`. As always, square brackets are used to denote the name of a field (or some other object within the ACCESS environment).

27 View the query in datasheet mode to ensure that the `BOWhere` field is yielding a result of the form: `CustID = 3` for each order detail.

Of course, `CustID` is only the first half of the WHERE clause. As in SQL, the "AND" and "OR" keywords can be used to create more complex criteria:

28 Edit the `BOWhere` field and add the `ProductID` half of the WHERE clause:

NL
```
BOWhere: "CustID = " & [CustID] &
" AND ProductID = '" &
[Products].[ProductID] & "'"
```

Since there are two fields in the query called `ProductID`, you must use the table name prefix to remove any ambiguity. In this case, since the tables are joined on `ProductID`, the values of `Products.ProductID` and `OrderDetails.ProductID` are (by definition) the same in all records in the results set. As such, it does not matter which `ProductID` field you use.

29 Test the query to ensure the WHERE clause works correctly, as shown in Figure 21.6.

21.3.3.3 Using the WHERE clause in the DLookUp() function

You can use one calculated field within another calculated field. We are going to exploit this capability in this section to keep our `DLookUp()` function simple and modular.

30 Create a new calculated field called `QtyOnBO`:

NL
```
QtyOnBO: DLookUp("QtyOnBO",
"BackOrders", BOWhere)
```

Note that the WHERE clause is this `DLookUp()` is the name of a calculated field, not a literal string. Before the `DLookUp()` is called, the `BOWhere` field is calculated and passed to the function.

31 Test the query and ensure the values contained in `QtyOnBO` are correct.

For a meaningful test, you must have at least one item in your `OrderDetails`

FIGURE 21.6: Ensure the ***BOWhere*** field is providing the correct WHERE clause.

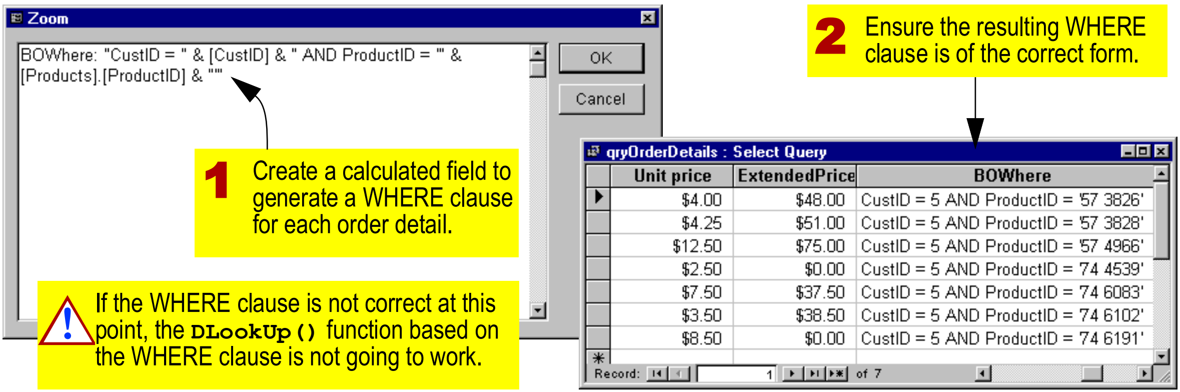


table that is also on backorder for a customer that has placed an order.

21.3.3.4 Creating a "suggested" quantity to ship field

In Section 19.3.3, you used your `MinValue()` function and a simple VBA procedure to set the value of `QtyShipped` to a suggested value once the value of `QtyOrdered` is specified. In this section, you are going to expand the logic to include the quantity on backorder.

Heeding the recommendation above to push calculations to the lowest level, the `SuggestQtyToShip` field is going to be implemented within the query.

32 Create a new calculated field called `SuggestQtyToShip`:

NL
```
SuggestQtyToShip:
MinValue([QtyOnHand], [QtyOrdered]
+ [QtyOnBO])
```

33 Test the query. You will notice that the results are not as expected. One more refinement is required to your backorder look-up.

21.3.3.5 Dealing with NULL values

The problem with the new **QtyOnBO** field is that **DLookUp()** returns a special value (NULL) when it cannot find a record matching the WHERE clause. Ideally, we would prefer never to have to backorder an item, the **BackOrders** table would be empty, and **QtyOnBO** would always be NULL.

NULL is problematic from an arithmetical point of view since any value plus NULL equals NULL.

To use **QtyOnBO** in a calculation, we must first map the NULL to a different value, such as zero.

34 Define another calculated field called **QtyOnBONotNull** using the built-in functions **iif()** and **IsNull()**:

NL
```
QtyOnBONotNull:
iif(IsNull([QtyOnBO]), 0,
[QtyOnBO])
```

Newer versions of ACCESS (since version 7) provide a built-in function called **Nz()** which can be used to map NULL values to some other value. The **Nz()** equivalent of the **iif()** statement above is:

```
QtyOnBONotNull: Nz([QtyOnBO]), 0)
```

35 Change the **SuggestQtyToShip** field so it uses the non-null quantity to ship.

21.3.3.6 Updating the trigger

36 Open **sfrmOrderDetails** in edit mode and update the trigger you created in Section 19.3.3 to set **QtyShipped** to **SuggestQtyToShip**.

As long as **SuggestQtyToShip** is projected into the **qryOrderDetailsQuery**, it is available for use in forms based on the query.

21.4 Discussion

21.4.1 When to fill backorders

The question of *when* to fill an outstanding backorder depends on the business context. In the low-volume kitchen supply scenario considered here, it probably does not make sense to make special backorder shipments. For example, if you owe SAM'S STOCK POT a $3.25 pastry blender, it is probably not worth your time to pack it up, fill out a waybill, arrange a courier pick-up and, and ship it off to him when your pastry blenders arrive. Instead, the assumption is made here that backorders are filled—if possible—on the customer's next order.

21.4.2 Using Data Access Objects (DAO)

The core of MICROSOFT ACCESS and an important part of VISUAL BASIC (the stand-alone application development environment) is the MICROSOFT JET database engine. The relational DBMS functionality of ACCESS comes from the JET engine; ACCESS itself merely provides a convenient interface to the database engine.[1]

Because the application environment and the database engine are implemented as separate components, it is possible to upgrade or improve JET without altering the interface aspects of ACCESS, and vice-versa. Indeed, you can download the latest version of the JET database engine from MICROSOFT's web site.

21.4.2.1 The DAO object hierarchy

MICROSOFT takes the component-based approach further in that the interface to the JET engine consists of a hierarchy of components (or "objects"). You were briefly introduced to the DATA ACCESS OBJECTS (DAO) hierarchy in Lesson 19.

[1] MICROSOFT has started its migration towards the MICROSOFT DATA ENGINE (MSDE), a scaled-down version of its SQL SERVER database engine. ACCESS 2000 still uses the JET engine; however, the MSDE engine is included on the OFFICE 2000 CD-ROM. The MSDE engine may be used in lieu of JET using a special ACCESS 2000 feature called a "data project".

The advantage of DAO is that it is modular and supports easier development and maintenance of applications. The disadvantage is that is you have to understand a large part of the hierarchy before you can write your first line of useful code. This makes using VBA difficult for beginners (even for those with considerable experience writing programs in BASIC or other 3GLs[2]).

21.4.2.2 DAO basics

You already have some familiarity with the DAO hierarchy. For example, you know that a **Database** object (such as `OrderEntry.mdb`) contains other objects such as tables (**TableDef** objects) and queries (**QueryDef** objects). Moving down the hierarchy, you know that TableDef objects contain **Field** objects.

The DAO object model is somewhat more complex than this. However, at this stage it is sufficient to recognize three things about DAO:

1. Each object that you create is an instance of a class of similar objects (e.g., `OrderEntry.mdb` is a particular instance of the class of Database objects).
2. Each object may contain one or more **Collections** of objects. Collections simply keep all objects of a similar type or

[2] Third-generation programming languages.

function under one umbrella. For example, Field objects such as `CustID` and `ProductName` are accessible through a Collection called **Fields**).

3. Objects have properties and methods (see below).

21.4.2.3 Properties and methods

You should already be familiar with the concept of object properties from working with the property sheets of form objects. The idea is much the same in DAO: every object has a number of properties that can be either observed (read-only properties) or set (read/write properties).

For example, each TableDef (table definition) object has a read-only property called **DateCreated** and a read/write property called **Name**. To access an object's properties in VBA, you normally use the ***<object name>.<property name>*** syntax. For example: `Employees.DateCreated`.

To avoid confusion between a property called `DateCreated` and a field (defined by you) called `DateCreated`, ACCESS recommends that you use a bang (!) instead of a period to indicate a field name or some other object created by you as a developer (recall the discussion on naming in Section 19.4). For example, `Employees!DateCreated.Value` identifies the **Value** property of the `DateCreated` field (assuming one exists) in the `Employees` table.

Methods are actions or behaviors that can be executed by objects of a particular class. In a sense, they are like predefined functions that only work in the context of one type of object. For example, all Field objects have a method called `FieldSize` that returns the size of the field. To invoke an object's methods, you use the ***<object name>.<method>*** [***parameter***$_1$, …, ***parameter***$_n$] syntax, e.g.,: `DeptCode.FieldSize`.

A reasonable question at this point might be: Isn't `FieldSize` a *property* of a field, not a *method*? The answer to this is that the implementation of DAO is somewhat inconsistent in this respect. The best policy is to look at the object summaries in the on-line help if you are unsure.

A more obvious example of a method is the `CreateField` method of TableDef objects, e.g.:

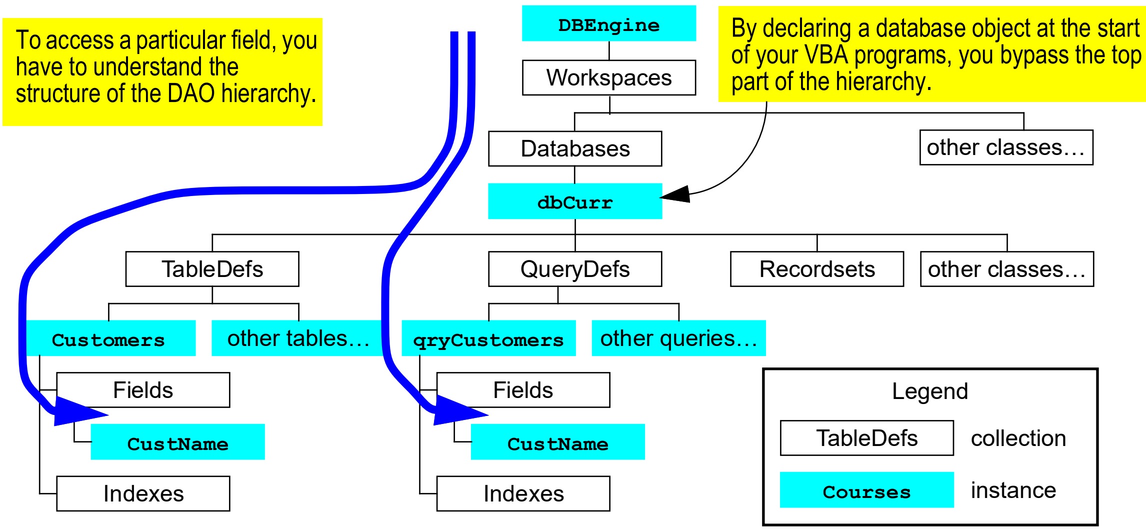
`Employees.CreateField("Phone", dbText, 25)`

This creates a field called `Phone`, of type `dbText` (a predefined constant used to represent text), with a length of 25 characters.

21.4.2.4 Engines, workspaces, and so on

A confusing aspect of the DAO hierarchy is that you cannot simply refer to objects and their properties as done in the examples above. As Figure 21.7 illustrates, you must include the entire path through the hierarchy in order to avoid any ambiguity between, say, the **CustName** field in the **Customers** TableDef object and the **CustName** field in the **qryCustomers** QueryDef object.

FIGURE 21.7: Navigating the DAO hierarchy.



Working down through the hierarchy is especially confusing since the first two levels (**DBEngine** and **Workspaces**) are essentially abstractions that have no physical manifestations in the ACCESS environment. The easiest way to avoid having to continually deal with the Database Engine and Workspace objects is to create a Database object that refers to the currently open database and start from the database level when working down the hierarchy. This is illustrated in Section 21.4.4.2.

21.4.3 Using the DAO object model in ACCESS 2000

The VBA code you wrote in Section 21.3.1.4 will not work in ACCESS 2000 unless you tell ACCESS where to find the DAO object library. The problem occurs because MICROSOFT has developed a new data access object model called ACTIVEX DATA OBJECTS (ADO) to supersede DAO (you will see a lot more of the ADO object model in Lesson 28). By default, ACCESS 2000 uses ADO instead of DAO.

Using the DAO model in an Access 2000 application is not difficult, however. You need only make two simple changes:

1. Tell Access where the DAO library is stored on your computer so its objects can be used by your VBA code.
2. Eliminate any ambiguity in your code between objects in the ADO object model and objects in the DAO object model.

To make the DAO object library accessible within the current ACCESS application, do the following:

37 Press **Ctrl-G** to bring up the dedicated VBA editor.

38 Select **Tools** → **References** from the VBA editor's main menu.

39 Scroll down until you find something that looks like "MICROSOFT DAO 3.*x* Object Library".

40 Check the box to include the DAO library.

The "scope" of the change to the references is limited to the current ACCESS application (that is, the current "mdb" file). You have not made any significant, irreversible change to your ACCESS environment.

The next step is to deal with the ambiguity issue. Ambiguity exists because the ADO and DAO object models both contain objects with the same name (e.g., Recordset and Field). Thus, if you enable DAO and run the code from Section 21.3.1.4, ACCESS 2000 will not know, for example, whether every reference to a Recordset is meant to be an ADO recordset or a DAO recordset.

To solve the problem, you have two options:

1. Remove the reference to the ADO object model so that your current application uses the DAO object model only. The ADO reference is set by default in ACCESS 2000; however, there is nothing to stop you from

opening the reference window (as above) and unchecking one of the defaults.

2. Leave the ADO reference intact, but add the prefix **DAO.<*object name*>** to all the declaration statements in your VBA code that are ambiguous.

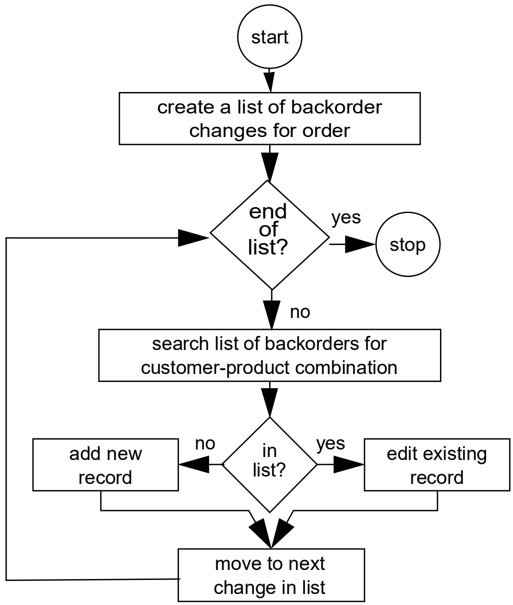
Although it could be argued that the first option is more efficient if you *know* you are not going to use ADO objects anywhere in your application, you should probably play it safe and simply modify a few `Dim` statements in your subroutine:

```
NL  Public Sub UpdateBackOrders(lngCustID As Long, lngOrderID As Long)
NL    'routine to process changes to backorders associated with an order
NL    Dim dbCurr As DAO.Database
NL    Dim rsChanges As DAO.Recordset, rsBackOrders As DAO.Recordset
NL    Dim qdf As DAO.QueryDef
NL    …
```

21.4.4 Understanding the UpdateBackOrders subroutine

A flow chart for the `UpdateBackOrders` subroutine is shown in Figure 21.8. To understand the VBA code in Section 21.3.1.4, you must first understand what the program is meant to accomplish.

FIGURE 21.8: A flow chart for the ***UpdateBackOrders*** subroutine.



In the sections below, certain aspects of the code from the subroutine are explained. The purpose here is not to make you a programmer; rather, it is to give you a better understanding of the basic workings for VBA and the DAO object model. For more information on these two topics, consult one of the many reference books available.

21.4.4.1 Declarations section

In VBA, it is good practice to declare variables before using them. When the `Option Explicit` statement is included at the top of the module, variables *must* be declared. The following lines of code declare four reference variables. Each reference variable has an *object* data type and will therefore be set to point to (or "reference") an object.

```
NL  Dim dbCurr As Database
NL  Dim rsChanges As Recordset,
    rsBackOrders As Recordset
NL  Dim qdf As QueryDef
```

21.4.4.2 Setting references to objects

When assigning a value to a non-object variable, you can use a simple assignment statement such as

```
NL  x = 5
```

Following the execution of this assignment statement, the location in memory named **x** contains the integer value 5. VBA relies on a

```
NL  Dim x As Integer
```

statement at the top of the program to know how much memory to allocate to hold the value of **x**.

When the variable is a reference to an object, the location corresponding to the variable name does not contain the object. Instead, the variable actually contains the *address of the object*. In lower-level languages such as C and C++, you have to manipulate memory addresses explicitly (this is one reason that these languages are considered difficult to learn).

In VISUAL BASIC, you do not need to know anything about memory addresses. However, you do need to use the **Set** keyword to make it clear that you are setting a reference to an existing object, rather than assigning a value to a variable. The following statements from **UpdateBackOrders** assign references to objects:

```
NL  Set dbCurr =
    DBEngine.Workspaces(0).Databases(0)
NL  Set rsBackOrders =
    dbCurr.OpenRecordset("BackOrders",
    dbOpenDynaset)
NL  Set qdf =
    dbCurr.QueryDefs!pqryBackOrderChang
    es
```

In the first statement, the variable **dbCurr** is set to point to the currently open database.

The **CurrentDb()** function can be used instead of the **DBEngine…** notation. For example, the following statement can be used to save some typing: **Set dbCurr =**

CurrentDb The second **Set** statement assigns a recordset object to a variable called **rsBackOrders**.

Recordset objects are the most useful objects in the DAO model. You can think of a recordset as being an invisible version of a datasheet. When you make a change to the data in a recordset and commit the change, the underlying database is changed.

Note the recordset object that **rsBackOrders** references is created on the right-hand side of the statement by the **OpenRecordset** method of the **dbCurr** database object. The **OpenRecordset** method accepts a number of parameters. In this case, the first parameter tells the method to create a recordset based on the **BackOrders** table. The second parameter (**dbOpenDynaset**) is a built-in constant that tells the method to create a specific type of recordset.

Different types of recordsets can be used for different purposes. For example, a read-only recordset is preferable when no changes are going to be made to the data because read-only recordsets are faster and require less memory than dynamic recordsets.

The final **Set** statement creates a reference to a QueryDef object called **pqryBackOrderChanges**. This querydef is simply a reference to the parameter query you created in Section 21.3.1.1. The advantage of using a querydef is that it permits a recordset object to be created based on the results of a saved query.

21.4.4.3 Creating a recordset from a parameterized select query

Consider the following statements:

```
NL  With qdf
NL    .Parameters!pOrderID = lngOrderID
NL    Set rsChanges =
      .OpenRecordset(dbOpenSnapshot)
NL  End With
NL  Set qdf = Nothing
```

The **With … End With** keywords are simply a convenience feature to save some typing. **With qdf** means that if an object reference is omitted in subsequent lines, the object is assumed to be **qdf**.

The **qdf** variable points to the **pqryBackOrderChanges** query. In the second line of code, the parameter named **pOrderID** is set to equal the value of the variable **lngOrderID**. In other words, the value of the parameter is set by VBA code. In the third line of code, the **OpenRecordset** method of the querydef object is used to create a new recordset containing the results from the parameter query. The parameter passed to the **OpenRecordset** method indicates that a "snapshot" (read-only) recordset is all that is required since no modifications will be made to the list of backorder changes.

The final line of code is a special type of **Set** statement. Whenever an object is no longer referenced by any variables, it is "destroyed". In other words, the memory occupied by the object is released back to the computer for use elsewhere. Generally, it is good practice to destroy any objects you create when you are sure you have no more need for them.

In VISUAL BASIC, objects are supposed to be destroyed automatically when variables go out of scope. For example, when the **UpdateBackOrders** subroutine ends, all objects created within the subroutine should be destroyed. This "automatic garbage collection" feature is absent from languages like C and C++. Such languages emphasize performance and require you to explicitly allocate and deallocate all memory used by all objects. If you forget to reclaim the memory used by an object, you create what is known as a memory leak: the program continues to use memory until either the system runs out of memory and crashes or the program is halted.

21.4.4.4 Looping through the list of changes

The following lines of code define the loop described in the flow chart. When a recordset is created, its current record pointer is located at the first record. The pointer can be moved down through the records using the **MoveNext** method of the recordset. If the **MoveNext** method is executed while the record pointer is at the last record in the recordset, the **EOF** (end-of-file)[1] property of the recordset is set to True.

The loop starts at the first record and executes until the end-of-file marker is encountered.

```
NL  Do Until rsChanges.EOF
NL    …
NL    rsChanges.MoveNext
NL  Loop
```

21.4.4.5 Searching for a matching record

The **CustID** for the customer in question is passed to the subroutine as an argument (**lngCustID**). The **ProductID** for each product requiring a backorder change is found in the

1 "End-of-file" and "beginning-of-file" (BOF) are vestigial terms from the days of reading and writing to disk files in BASIC. "End-of-recordset" would be a better name for this property.

rsChanges recordset. Using these two pieces of information, it is possible to search through the rsBackOrders recordset to determine if the customer-product combination already exists.

A convenient way to search through a recordset is to use the FindFirst method. The argument passed to the FindFirst method is a string that specifies the search conditions. In other words, it is similar to the WHERE clause strings you created for the DLookUp() function earlier in this lesson.

```
NL  rsBackOrders.FindFirst "CustID = "
    & lngCustID & " AND ProductID = '"
    & rsChanges!ProductID & "'"
```

If a record matching the search criteria is found in the recordset, the record pointer is set to point to the record. In addition, the recordset's NoMatch property is set to False. Conversely, if no matching record is found, the record pointer is left at EOF and the NoMatch property is set to True.

The value of rsBackOrders.NoMatch can therefore be used to determine whether to add a new backorder record or update an existing one:

```
NL  If rsBackOrders.NoMatch Then
NL    …
NL  Else
NL    …
NL  End if
```

21.4.4.6 Editing an existing record

If NoMatch is false (i.e., a backorder for the product and customer in question already exists), the following code is executed:

```
NL  With rsBackOrders
NL    .Edit
NL    !QtyOnBO = !QtyOnBO +
      rsChanges!BOChange
NL    .Update
NL  End With
```

The recordset's Edit and Update methods are used to initiate and commit changes to a recordset. Once the Update method is executed, the changes are saved to the underlying table. In this code, the value of the QtyOnBO field is changed by the amount of the rsChanges!BOChange field. Since the record pointer is set by the FindFirst method, the correct backorder record is being updated.

21.4.4.7 Adding a new record

Adding a new record is similar to editing an existing record except that the new record must be created before its fields are assigned values. The AddNew method creates a new blank record and sets the record pointer to point to the record. Since the record is new, the CustID and ProductID fields must be filled in. Again, if you forget to execute the Update method, the changes are not saved to the underlying table.

```
NL  With rsBackOrders
```

```
NL      .AddNew
NL      !CustID = lngCustID
NL      !ProductID = rsChanges!ProductID
NL      !QtyOnBO = rsChanges!BOChange
NL      .Update
NL   End With
```

21.4.4.8 Closing open recordsets

Programming is much like being a good roommate: it is good practice to close what you open:

```
NL   rsChanges.Close
NL   rsBackOrders.Close
```

21.4.5 Why you cannot add backorders to your order details query

In this section, we are going to revisit the question: Why don't we simply include the **BackOrders** table in a join query instead of using a **DLookUp()** function to get the outstanding backorders for each product?

Although it certainly is possible to drop the **BackOrders** table into the **qryOrderDetails** query, it does not mean that the resulting query can be used for data entry. ACCESS can recognize anomalous joins and makes the recordsets based on such joins non-updatable.

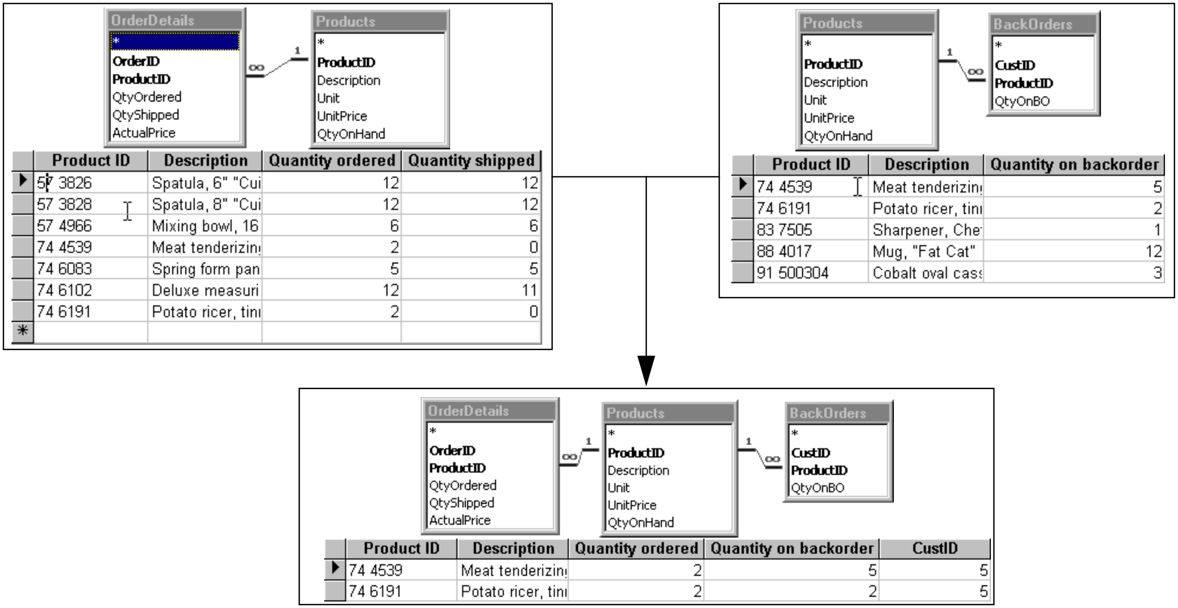
A perfect example of an anomalous join is the relationship that you may have been tempted to create for determining the backordered quantity. In the original **qryOrderDetails** query, a valid one-to-many relationship exists between **Products** and **OrderDetails** (each product can appear in many order details, but each order detail refers to only one product).

If **BackOrders** is added to the query, the result can be conceptualized as two one-to-many relationships with the one side (**Products**) at the center. To understand how these two relationships interact, consider them separately, as shown in Figure 21.9. First, we should assume that a condition has been added to the query so that only the order details for a single order (say **OrderID** = 1) are included.

The results for the **OrderDetails-Products** relationship are shown on the left-side of Figure 21.9. The results for the **BackOrders-Products** relationship are shown on the right side of Figure 21.9.

Recall that a join query takes each record on the *many* side of the relationship and joins it with the corresponding record on the *one* side of the relationship. As such, the only products records that shown in left-hand results set are those for which there is a matching record in the **OrderDetails** table. Similarly, the only products records that shown in right-hand results set are those for which there is a matching record in the **BackOrders** table.

FIGURE 21.9: Decomposition of the ***OrderDetails-Products-BackOrders*** query.



Now consider the result of combining the two relationship, as shown at the bottom of Figure 21.9. The combined results set is a list of products which have been ordered in **`OrderID`** = 1 *and* which have been backordered by at least one customer.

Although this might be an interesting query for management decision making, consider the implications of basing your order subform on such a query: Products that belong to **`OrderID`** = 1, but which do not appear anywhere in the **`BackOrders`** table are not included in the results set. As such, if ACCESS permitted you to add such product using the order form, the order detail would disappear from the subform as soon as it was entered). This is clearly not the type of behavior we seek.

In general, any three-table join with the *one* side in the middle is going to result in a non-updatable recordset. However, any well-formed three-table join with the many side in the middle is fine. An example of this is the **`Orders-OrderDetails-Products`** join you created in Figure 21.5.

21.5 Application to the assignment

A useful feature of the application would be to add any outstanding backorders for a customer when an order for the customer is being created.

41 Add a feature to your order form to add backordered products to a customer's order when the order is created.

Although this is left as an exercise, you may want to consider the following hints:

1. A parameterized append query can be used to copy the **`ProductID`**s on backorder for a particular customer to the **`OrderDetails`** table.
2. As soon as you fill in the **`CustID`** field on the main order form, you know which customer is placing the order. Thus, **`CustID`** can be used to trigger *and* provide the parameter for the append query.
3. Obviously, if the **`QtyOnBO`** field for a particular item is zero, it should not be added to **`OrderDetails`**.
4. If records in a table are changed (e.g., by an append query) after a form based on the table is opened, the **`Requery`** method for the form has to be executed to show the changes.
5. The **`ActualPrice`** and **`QtyShipped`** fields are set by event-driven procedures on the order subform. However, if order details are added by an append query, the events on which the procedures are based are never raised. As such, the append query should take care of setting these fields to appropriate values.

Lesson 22: An introduction to data warehousing

22.1 Introduction: Data access for decision makers

When you used the ACCESS report writer to create an invoice report in Section 16.4, you may have made an important observation:

> *Creating a report from a relational database requires a solid understanding of where the data is stored and how the tables are related.*

In this case, you are the person who implemented the database so you know that each `Order` consists of many `OrderDetails` and that `OrderDetails.ActualPrice` is what the items sold for, not `Products.UnitPrice` and so on. But who else in your organization could realistically be expected to know all these details?

22.1.1 Database specialists versus business specialists

A real-world business application typically contains hundreds or thousands of tables and a mind-boggling web of relationships between tables. The good news is that within each organization, there is an individual who understands how all the data fits together—the database administrator (DBA). The bad news is that the DBA is a database specialist, not a business specialist.

The decision-makers who most need the information locked up inside the database (marketing managers, executives) seldom have any training (or even interest) in the subtleties of third-normal form, concatenated keys, or referential integrity. Similarly, as a class of individuals, DBAs are not known for their marketing instincts or general business acumen.

22.1.2 Dimensional data modeling

In the early 90s, a new class of database application—the data warehouse—emerged to address the problems encountered by managers as they tried to access information locked up inside of transaction processing systems. At the simplest level, a data warehouse is simply a read-only copy of the data in a transaction processing system. However, instead of being optimized for transaction processing, a data warehouse is optimized for reporting and decision support. Specifically, a data warehouse is based on a “dimensional” data model rather than a “normalized” data model.

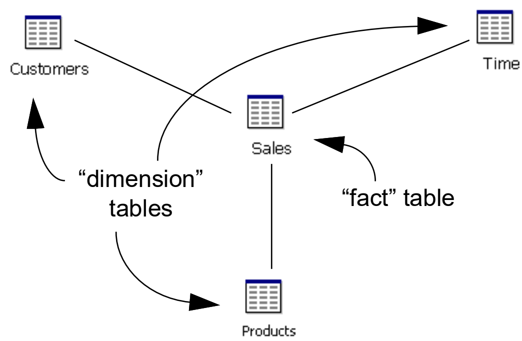
An example of a dimensional data model is shown in Figure 22.1: the center table—**`Sales`**—contains the "fact" of primary importance to decision makers: the dollar amount sold. The other tables—**`Customers`**, **`Time`**, and **`Products`**—are the "dimensions" along which sales vary.

For example, a manager may want to know who her best customers are this quarter and what they are buying. Finding an answer using a normalized database would require some reasonably sophisticated knowledge of both a query language and the table structure of the database. However, as you will see, it is very easy to answer this type of question when the source of the data is organized into facts and dimensions. The ease with which business users can create complex queries is an important benefit of the data warehousing approach to decision support.

22.1.3 Building a data warehouse

In this lesson, you will build a very small-scale data warehouse. Although your warehouse will be implemented in ACCESS and will be a fraction of the size of a real-world warehouse, even a small warehouse is sufficient to illustrate the critical elements of data extraction and dimensional data modeling. In Lesson 23 you will use your data warehouse to explore your data in greater detail and answer complex questions about your business.

FIGURE 22.1: A dimensional data model.



22.2 Learning objectives

- understand the difference between data models for transaction processing and data models for decision support
- denomalize data to create dimension tables
- extract data to create a fact table
- build a star schema
- use grouping to change the granularity of a fact table

22.3 Exercises

Since you may not have enough data in your order entry application to yield interesting query results, we will use the NORTHWIND TRADERS sample database (recall Section 4.3.2) as the source for our data warehouse. The NORTHWIND TRADERS database is also small by real-world standards; however, it contains enough orders (just over 800) to make the querying exercises in this lesson and Lesson 23 worthwhile.

22.3.1 Preliminaries

Rather than alter the NORTHWIND TRADERS sample database, we are going to create links to its tables (recall Section 8.3.3) and **extract** the data into a new database file.

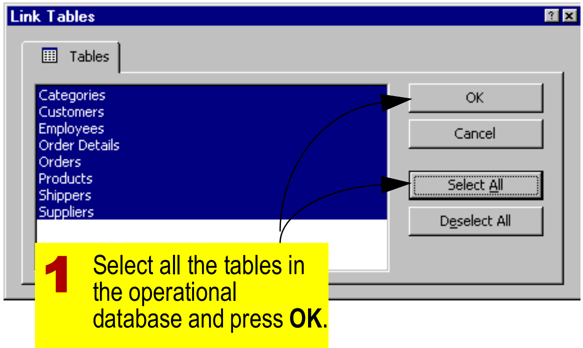
1 Create a new blank database in ACCESS called **`OrderEntryWarehouse`**.

2 From the main menu, select **File** → **Get External Data** → **Link Tables**.

3 Use the search feature of the "Link" dialog box to find the NORTHWIND TRADERS database (recall Section 4.3.2).

4 When asked which tables you would like to create links to, select all the tables, as shown in Figure 22.2.

FIGURE 22.2: Select tables in the operational system to link to.



Your data warehouse database should now contain links to all the tables in a transaction processing system application.

5 Since you are still playing the role of the DBA at this point, bring up the relationships window to get a sense for the structure of the NORTHWIND TRADERS application.

22.3.2 Extraction, cleaning, and transformation

Data extraction is the process of copying the data from the **operational system** (i.e., the NORTHWIND TRADERS order entry system) to the data warehouse. During extraction, two things normally happen to the data:

1. **Cleaning** — Data cleaning (or scrubbing) involves removing incorrect or inconsistent data, missing values, and so on. As you can imagine, this is costly and difficult process that involves a combination of specially written programs and manual intervention.
2. **Transformation** — Data from the transaction processing system must be transformed into a format suitable for reporting, analysis, and other decision support activities. Typically, transformation involves de-normalizing dimension tables and pre-computing fact tables. Both these processes are illustrated in the following sections.

To keep things simple, we are going to assume that the NORTHWIND TRADERS order entry application has been designed to minimize the possibility of errors entering the database. As such, the data cleaning stage is assumed to be unnecessary.

In the next few sections, you will use action queries to copy data from the linked tables you created in Section 22.3.1 to new tables in your `OrderEntryWarehouse` database.

22.3.3 Creating dimension tables

Dimension tables are relatively static: they contain lists of products, customers, and so on. However, since the dimensions determine the types of questions you can ask of your data warehouse, it is important to put some thought into their design. There are two important questions to answer before you dive in:

1. What dimensions are important for making business decision?
2. What is the appropriate level of granularity for each dimension?

These questions are addressed on a case-by-case basis in the following sections.

22.3.3.1 The product dimension

Clearly, you are going to want to look at sales by product. The granularity issue is whether you need to look at individual SKUs (stock keeping units) or whether a coarser-grained approach (e.g., product category) is sufficient.

6 Create a new query called `qryProductDimensionExtract` based on the `Products` table.

7 From the **Query** menu, select **Make Table Query**.

8 When prompted for the name of the new table, enter `dimProducts`.

Since you already have a linked table called `Products`, you cannot use the same name. In addition, the `dim` prefix is an easy way to indicate that the table contains denormalized dimension data.

In the Northwind Traders database, each product is assigned to a category (beverages, condiments, and so on). If we analyze products by individual SKUs (in this case, `ProductID`), then the granularity is quite fine. However, if we lump all products within a particular category together and perform our analyses at the category level, then the granularity is more coarse.

If your dimension it too fine-grained, your data warehouse will be very large may involve excessive processing to respond to user queries. However, if your dimension is too coarse-grained, you will be unable to ask certain kinds of questions.

In this case, we are going to include both product and category information. This will permit the user of the warehouse to drill-down to the appropriate level of granularity.

9 Include the `Categories` table in the query. There should be a one-to-many relationship between `Categories` and `Products`.

10 Project the `ProductName` and `CategoryName` fields into the query. These will be the values the user sees.

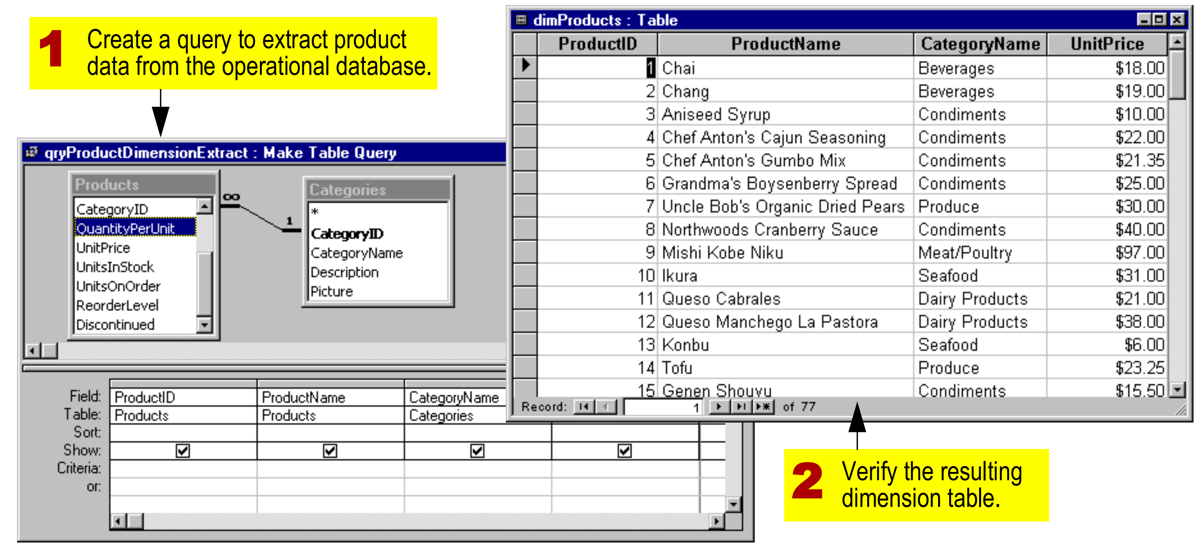
11 Project the `ProductID` field into the query. This value will be used as a key to link the dimension table to the fact table.

12 Finally, project the `UnitPrice` field into the query.

The rationale for including `UnitPrice` in the dimension table is that users may want to limit their analysis to high (or low) valued items. Having the `UnitPrice` field in the data warehouse allows users to apply price-related constraints to their queries. More generally, knowing what fields to include in a data warehouse requires a good understanding of how users make decisions. When it doubt, it is probably best to err on the side of including too much.

13 Select **Query** → **Run** to execute the query. Examine the contents of `dimProducts`, as shown in Figure 22.3.

FIGURE 22.3: The extraction query for the product dimension.



22.3.3.2 Taking a closer look at the products dimension table

There are a couple of things to notice about the `dimProducts` table:

1. **Denormalization** — When designing transaction processing systems, we make every effort to eliminate redundancy in our tables (recall the discussion of normalization in Section 7.1.1). In data warehousing, database design logic is turned on its head: In order to save the computational effort of making a join when running queries against the data warehouse, the dimension table includes information from multiple entities (e.g., products and categories). Since data in the warehouse is

never changed or edited, this denormalized structure does not lead to the anomalies discussed in Section 7.1.

2. **User-friendly values** — Since the extracted data is ultimately going to be used for creating reports, meaningful field values (such as `ProductName` and `CategoryName`) are used instead of key fields (like `CategoryID`). Key fields (such as `ProductID`) are only added when necessary for linking to a fact table.

22.3.3.3 The customer dimension

The customer dimension is similar to the product dimension in that a hierarchical relationship in implicit in the data. In this case, assume that users require data right down to the level of the individual customers, but may also want to aggregate across cities, regions, and countries.

14 Create a new make-table query called `qryCustomerDimensionExtract`.

15 Use `dimCustomers` as the name of the target table.

16 Include the `Customers` table. In the NORTHWIND TRADERS database, the region information is included within the `Customers` table so there is no need to create a join to another table.

17 Project the following fields into the query: `CompanyName`, `City`, `Region`, and `Country`.

18 Project `CustomerID` to enable the table to be linked to the fact table.

19 Execute the query and verify the contents of `dimCustomers`, as shown in Figure 22.4.

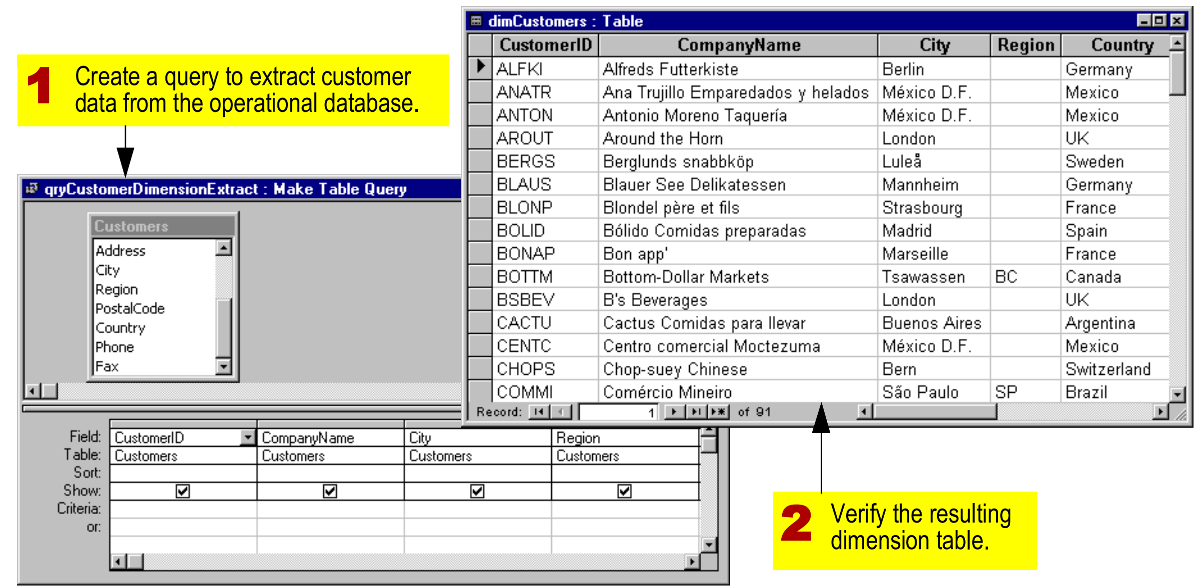
22.3.3.4 The time dimension

The time dimension is different from the products and customers dimension in that time exists independently of any particular data warehouse application. As a consequence, it is possible to create a *generic* time dimension table consisting of a date ID plus days of the week, months, quarters, years and so on. This dimension table could be used for all data warehouse applications.

In this lesson, we are going to take a different approach for two reasons.

1. **Event time vs. calendar time** — Although sales per day (or hour or minute) may be a meaningful piece of information, we are also interested in sales *per order*. One may not normally think of `OrderID` as a measure

FIGURE 22.4: The extraction query for the customer dimension.



of time; however, it is important to remember that each order is an event. Since many orders can occur per day, the granularity desired in this context is finer than the granularity of a generic date-based dimension table.

2. **Date manipulation functions**— Since each order has an order date, it is possible to derive the coarser-grained values of time using specialized data manipulation functions.

20 Create a new make-table query called `qryTimeDimensionExtract`.

21 Use `dimTime` as the name of the target table.

22 Include the `Orders` table and project the `OrderDate` field.

23 Project the `OrderID` field so that a link can be made to the fact table.

22.3.3.5 Transforming the OrderDate field

The `OrderDate` field is defined as a Date/Time data type and contains all the information we require about day, month, year and so on. The trick is to extract this information and display it in a user-friendly format. To do this, we will use calculated fields and the built-in `DatePart()` function.

24 Create a new calculated field in `qryTimeDimensionExtract` called `Year` and define it as follows:

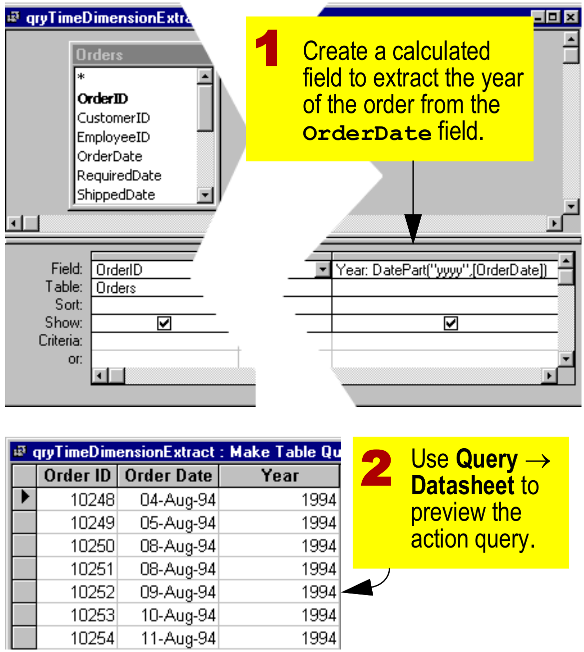
NL `Year: DatePart("yyyy", OrderDate)`

The `DatePart()` function takes two arguments: a special set of characters that determines what part of the date is returned and a valid date. For example, the argument "yyyy" tells the function to return a four-digit year.

Use the on-line help system and search under "datepart" to learn more about the function and its arguments.

25 Select **View → Datasheet View** to preview new table and verify the `DatePart()` function, as shown in Figure 22.5.

FIGURE 22.5: Use the ***DatePart()*** function to show the year of an order.



MICROSOFT updates the NORTHWIND TRADERS database from time to time. As a consequence, the dates return by your queries may not correspond exactly to those shown in Figure 22.5.

The procedure for transforming the month is the same except that `DatePart()` only returns the ordinal number of the month, not the name. Since the whole purpose of this exercise is to provide user-friendly, readable values for each dimension, a means of mapping month numbers to month names is required.

One approach is to create a lookup table of month numbers and months. A somewhat simpler approach (which we will use here) is to use the `Choose()` function within ACCESS.

26 Create a new calculated field called month and define it as follows:

NL `Month: DatePart(“m”, OrderDate)`

27 Preview the results and verify that the result is a number from 1 to 12.

28 Modify the `Month` field so that the `DatePart()` function provides the first argument for the `Choose()` function:

NL `Month: Choose(DatePart(“m”, OrderDate), “Jan”, “Feb”,“Mar”,“Apr”,“May”,“Jun”,“Jul”,“Aug”,“Sep”,“Oct”, “Nov”, “Dec”)`

The `Choose()` function maps an index number (1, 2, …) to the corresponding value in a list of choices. For example, the index number 2 maps to the second choice, and so on. See on-line help for more information about the `Choose()` function.

29 Create a new calculated field called `Quarter` as follows:

NL `Quarter: “Q” & DatePart(“q”, OrderDate)`

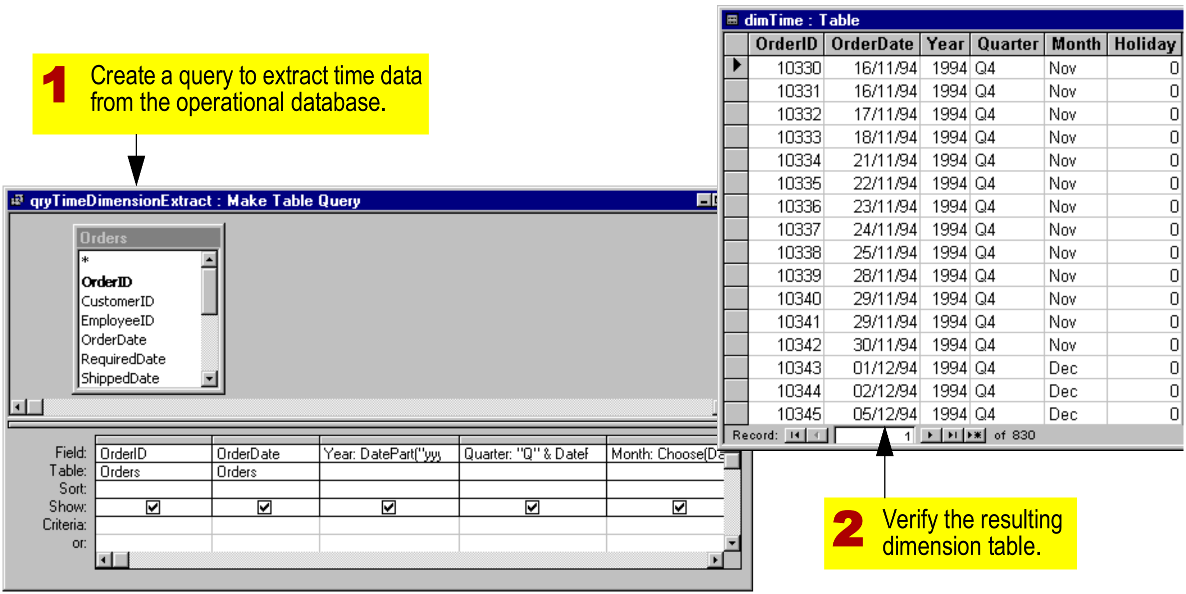
When the first argument for the `DatePart()` function is “q”, the function returns a value 1 to 4 corresponding to the quarter. To make it more readable, the letter “Q” is added to the front of each value:

30 Include a final calculated field called `Holiday`:

NL `Holiday: False`

The `Holiday` field creates a new field in the `dimTime` table to indicate whether a particular date is a holiday. This type of information is often useful in a retail context for interpreting spikes in demand. Of course, someone has to go through the `dimTime` table and change `Holiday` = True where appropriate.

31 Execute the query and verify the results, as shown in Figure 22.6.



OrderID	OrderDate	Year	Quarter	Month	Holiday
10330	16/11/94	1994	Q4	Nov	0
10331	16/11/94	1994	Q4	Nov	0
10332	17/11/94	1994	Q4	Nov	0
10333	18/11/94	1994	Q4	Nov	0
10334	21/11/94	1994	Q4	Nov	0
10335	22/11/94	1994	Q4	Nov	0
10336	23/11/94	1994	Q4	Nov	0
10337	24/11/94	1994	Q4	Nov	0
10338	25/11/94	1994	Q4	Nov	0
10339	28/11/94	1994	Q4	Nov	0
10340	29/11/94	1994	Q4	Nov	0
10341	29/11/94	1994	Q4	Nov	0
10342	30/11/94	1994	Q4	Nov	0
10343	01/12/94	1994	Q4	Dec	0
10344	02/12/94	1994	Q4	Dec	0
10345	05/12/94	1994	Q4	Dec	0

FIGURE 22.6: The extraction query for the time dimension.

22.3.4 Creating a fact table

A fact table contains one or more results of interest for each unique combination of dimensions. For example, if we were interested in the value of sales of products to customers per day, then we would compute this value for each product × customer × day and store it in a table. In the case of NORTHWIND TRADERS, the results would be 77 × 91 × 365 = 2.5 million facts per year.

The number of non-zero facts is be much smaller than 2.5 million since all customers do not order all products every day of the year. Despite this sparseness however, fact tables tend to be very large.

22.3.4.1 Determining foreign keys

Through the selection of keys in the dimension tables, we have already determined the foreign keys that must be included in the fact table: `ProductID`, `CustomerID`, and `OrderID`.

32 Create a new make-table query called `qryFactExtract`.

33 Use `factSales` as the name of the target table.

34 Add the `Order` and `OrderDetails` tables to the query.

Note that the `Order` and `OrderDetails` tables contain all the fields we need to create joins to the dimension tables: `ProductID`, `CustomerID`, and `OrderID`.

35 Project the foreign keys into the query definition.

22.3.4.2 Calculating sales

What remains to be determined is the dollar value of sales for each combination of the dimensional values. To calculate the total sale for each product × customer × order combination, we have to know a couple of things about the data:

1. The total value of an order is the sum of extended prices of the line items in the order.
2. The `OrderDetail.UnitPrice` value can be discounted by the amount in `OrderDetails.Discount`. The extended price calculation must therefore include the discount.

36 Create a calculated field called total sale as follows:

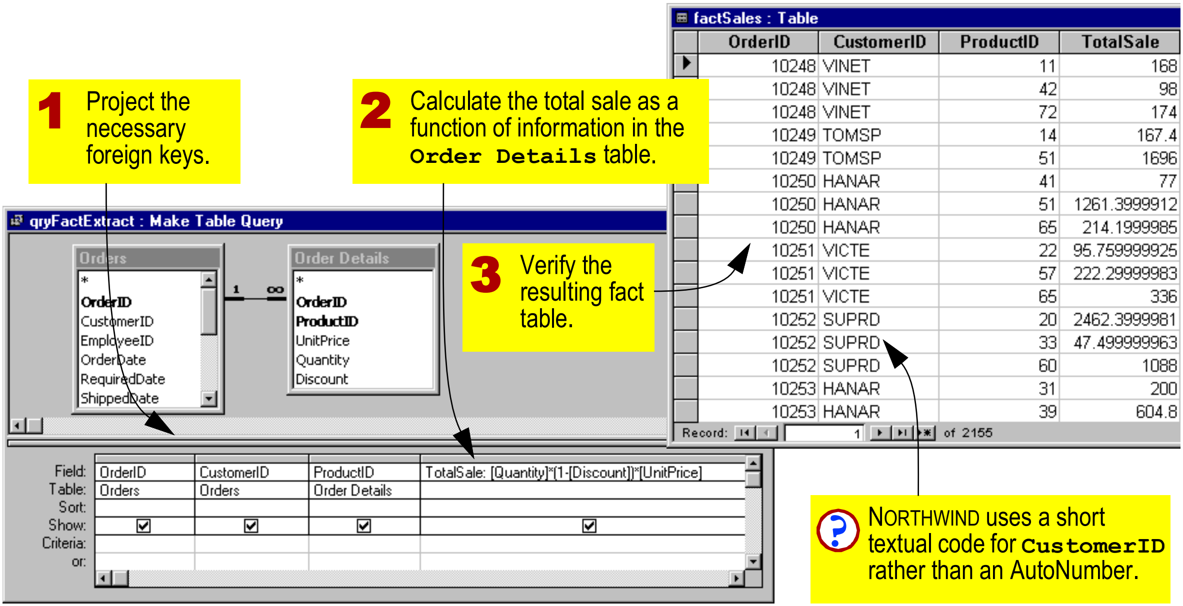
```
NL  TotalSale: Quantity *
    (1-Discount)* UnitPrice
```

37 Preview the results and examine the results as shown in Figure 22.7.

22.3.5 Refresh intervals

Let's summarize what you have done to this point: You have extracted and transformed data from one database and stored it in another database. The new database (`OrderEntryWarehouse.mdb`) is simply a static

FIGURE 22.7: The fact table for order-level analyses of sales.



copy (or "snapshot") of the NORTHWIND TRADERS application.

Clearly, your warehouse data is out-of-date as soon as a new transaction is added to the operational system. But since we plan to use the data warehouse to see the big picture (e.g., sales trends over the last four quarters), an order here or there does not make that much difference. You can refresh the data warehouse daily, weekly, monthly, or according to whatever schedule makes sense.

22.3.6 Creating a star schema

A star schema is a set of relationships between a fact and several dimension tables. Since the fact table is at the center and many dimension tables are around the perimeter (recall Figure 22.1) the configuration resembles a star—hence the name.

38 Create a new select query called `qryStarSchema`.

39 Add the `factSales`, `dimCustomers`, `dimTime`, and `dimProducts` table to the queries.

40 Drag the primary keys onto the corresponding foreign keys in the fact table to create query-level relationships, as shown in Figure 22.8.

Since the data warehouse is read-only, there is no need to create relationships in the relationship window or specify referential integrity.

41 Project the finest-grained field from each dimension table into the query (specifically: `OrderID`, `CompanyName`, `ProductName`).

42 Project the `TotalSale` field.

Since the `TotalSale` field has a numeric data type, you may want to show it in the query formatted as currency.

43 Right-click anywhere on the `TotalSale` field, bring up the properties sheet, and enter "Currency" in the **Format** property.

Changing the format of a query field is merely an aesthetic enhancement—the underlying representation of the number remains the same. You may also change the data type of the `TotalSale` field in the `factSales` table—the result is the same.

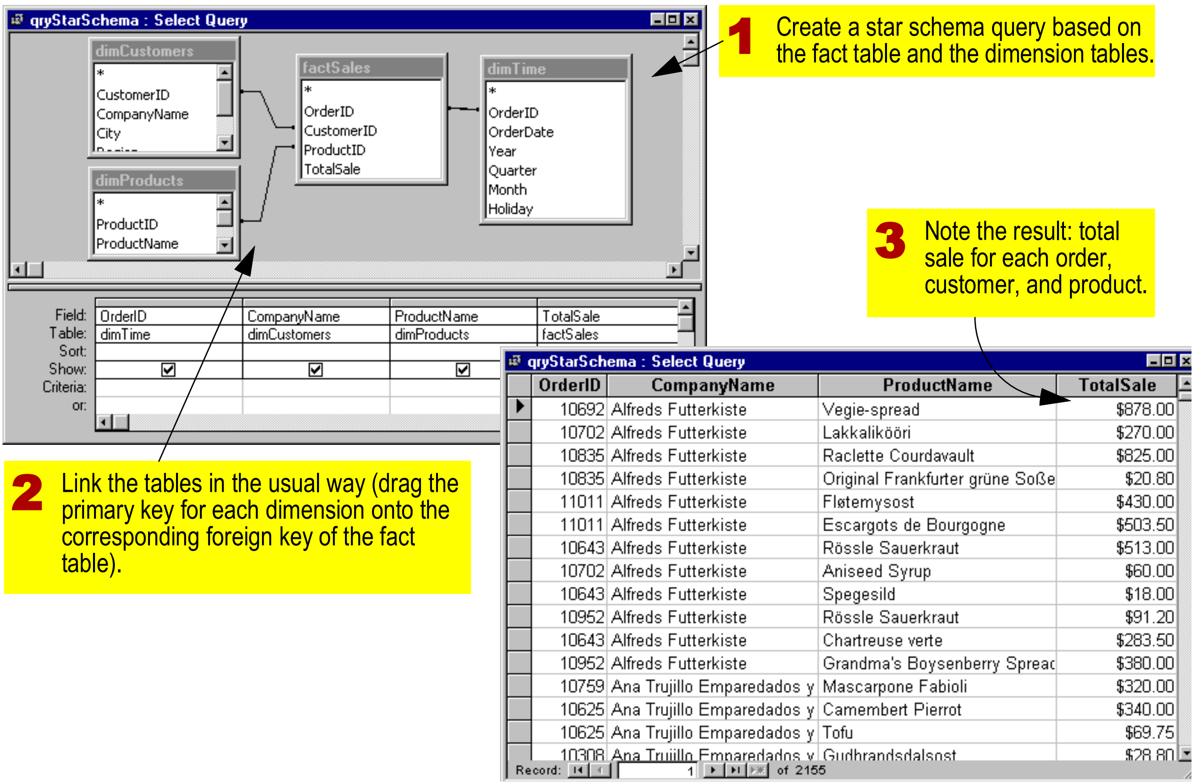
44 View the star schema query in data sheet mode, as shown in Figure 22.8.

45 Ensure you understand the meaning of each row: the `TotalSale` represents the amount per product per order per customer. This is identical to the granularity of the `OrderDetails` table.

22.3.7 Aggregating data using the GroupBy operator

The level of granularity in Figure 22.8 is probably too fine to be useful for many decision making purposes. In this section, you are going to use the "totals" feature in QBE (which is

FIGURE 22.8: Create a star schema to join the fact table with several dimension tables.



identical to the `GroupBy` operator in SQL) to aggregate the data.

22.3.7.1 Setting up grouping and totals

46 Switch back to the design view of `qryStarSchema`.

47 Select **View** → **Totals** from the main menu. Alternatively, press the sigma (Σ) button on the toolbar.

48 Notice that a "Total" row is added to the query definition grid and that term "Group By" appears in the row for every field.

49 Leave the "Group By" entry for all the foreign keys, buy change it to "Sum" for the `TotalSale` field, as shown in Figure 22.9.

50 Preview the results. You will note no change from the previous result since grouping on unique values of `OrderID`, `CompanyName`, and `ProductName` results in individual order details.

22.3.7.2 Different levels of aggregation

There are two ways to change the level of aggregation in a star schema: change the level of granularity for a dimension or drop the dimension from the results set altogether.

51 Before switching back to design view, make a mental note of the number of records in the results set (2155 records are shown in Figure 22.9; however, the number you see may vary depending on the version of the NORTHWIND TRADERS database you are using).

52 Switch to design view, click on the grey bar above the `OrderID` field in the query definition grid, and press **Delete**.

53 Preview the results and make a mental note of the number of records in the results set.

In this modified query, you are grouping on `CustomerID` and `ProductID` and summing extended price. What this means is that the value of `TotalSale` reflects the sum of sales for each unique combination of product and customer regardless of when (i.e., in which order) the products were ordered.

54 Return to design mode and delete the `ProductID` field from the query definition grid.

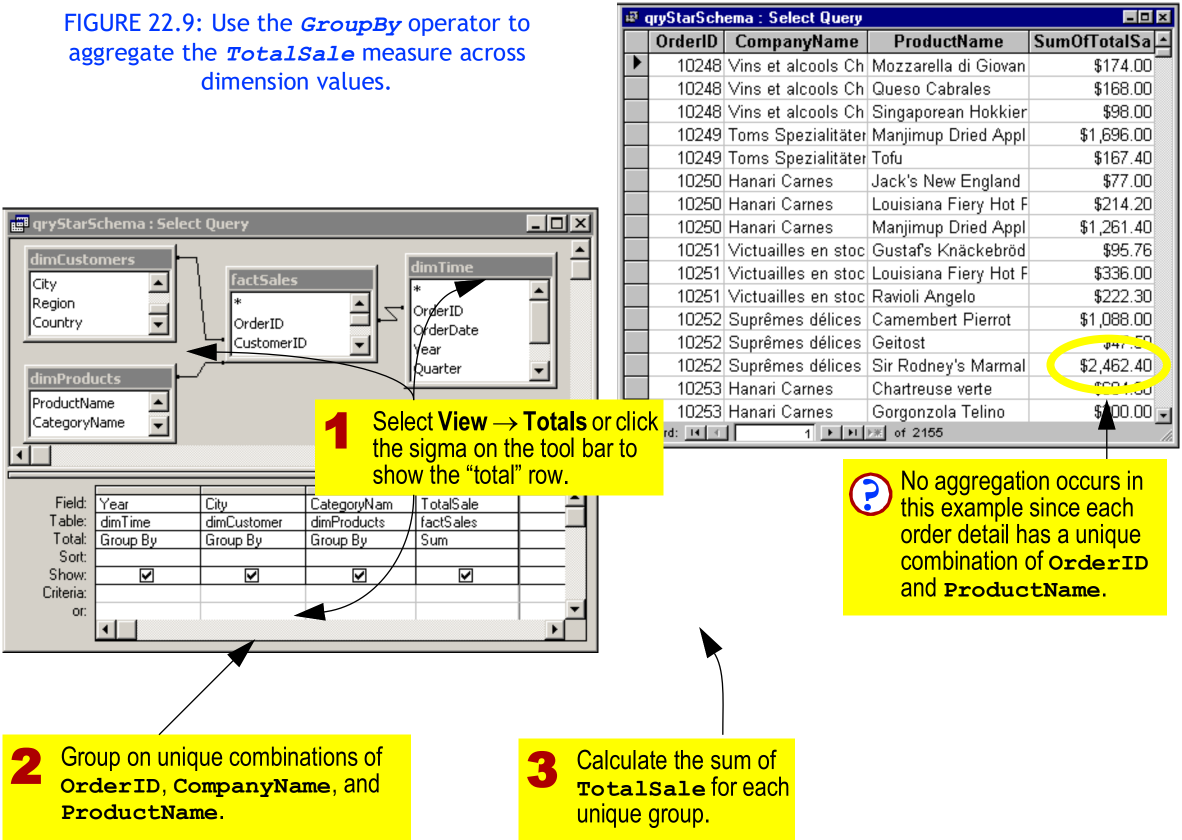
55 Preview the results and make a mental note of the number of records in the results set.

In this case, you are grouping on `CustomerID` only. The `TotalSale` field represents the total value of all products in all orders to the customer in question.

56 Return to query design mode and delete the `CustomerID` field.

57 Preview the results.

FIGURE 22.9: Use the *GroupBy* operator to aggregate the *TotalSale* measure across dimension values.

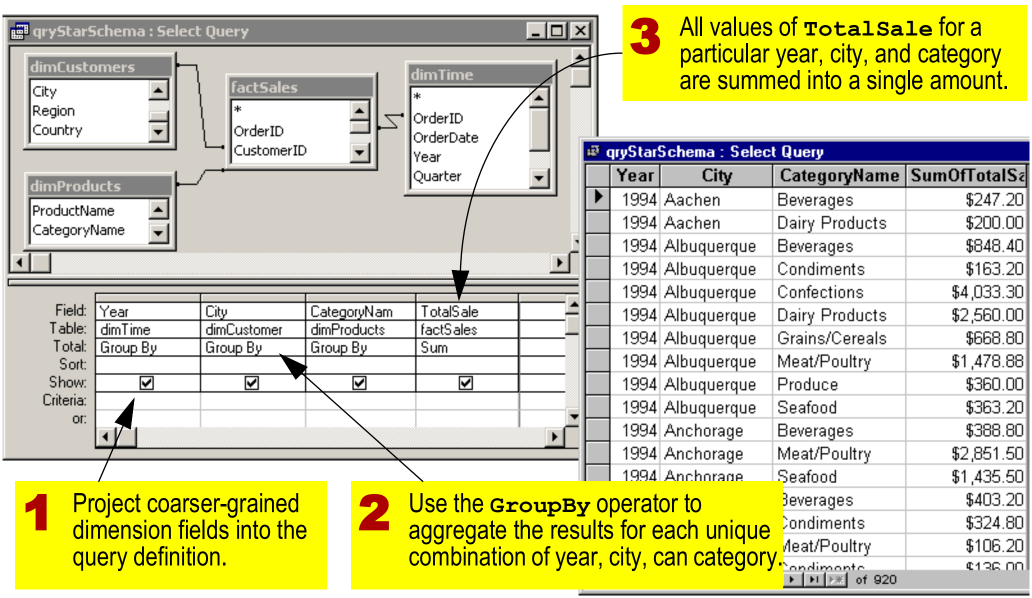


Now, the total sales for all customers, products, and orders is collapsed into a single value. To get subtotals for particular values of one or more fields, reverse the process by adding them to the query and using the `GroupBy` operator.

58 Project `Year`, `City`, and `CategoryName` into the query definition grid.

59 Verify the results as shown in Figure 22.10.

FIGURE 22.10: Total sales by year, city, and category.



22.3.8 Using aggregation and a star schema to answer a business question

The top part of the query in Figure 22.10 is a dimensional data model. To test the hypothesis that it is easier for decision makers to create their own queries using dimensional data models, we can start by considering a business question:

- *What were the total sales of each product in each city in Canada for 1994? Break the results down by quarter and sort the results in descending order of importance.*

60 Create a new query called `qryQuestion`.

61 Repeat the steps in Section 22.3.6 to create a star schema query.

62 Project `City`, `Quarter`, `ProductName`, and `TotalSale` into the query.

63 Ensure the “Totals” feature is on and that you are grouping by unique combinations of `City`, `Quarter`, and `ProductName`.

64 Sum the `TotalSale` field.

65 Set the query to sort on `TotalSale` in descending order.

66 To constrain the results to Canada in 1994, project the `Country` and `Year` fields into the query. For both fields, ensure the “Show” box is unchecked and that the “Group By” entry is replaced by “Where”.

67 View the results as shown in Figure 22.11.

Based on the results of the query, you may re-evaluate the effectiveness of you sales programs in Canadian cities other than Montreal.

Hopefully you agree that a reasonably query-literate individual could construct this type of query and interpret its results. In Lesson 23 you will perform more sophisticated queries and analysis using your new data warehouse.

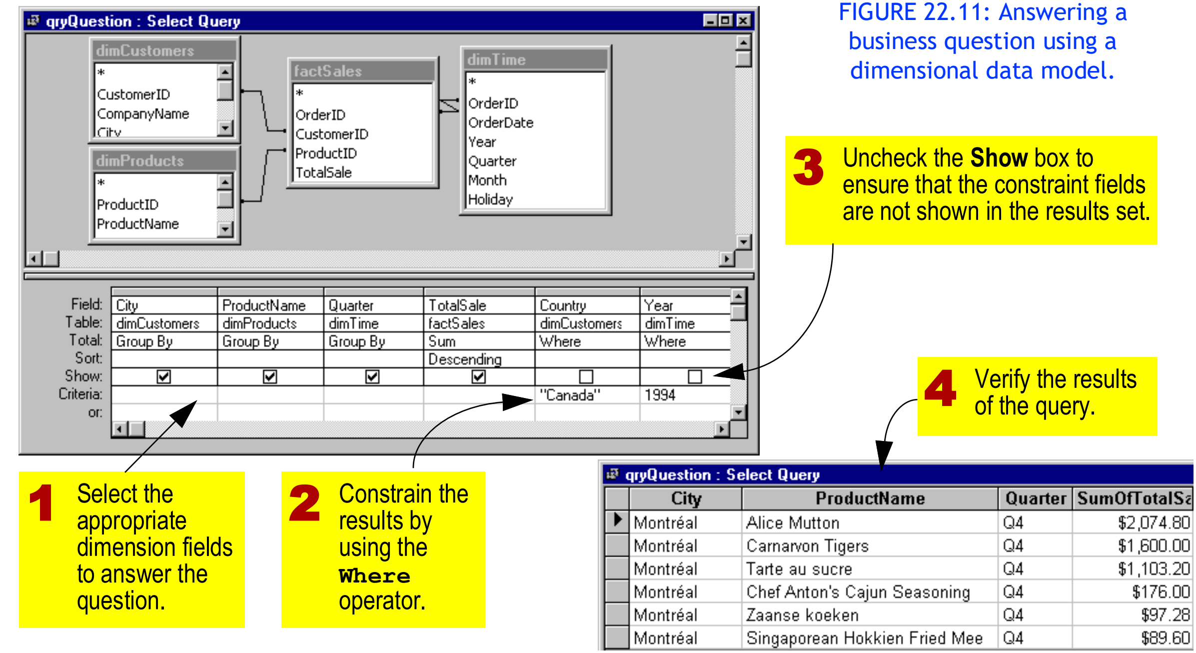
22.4 Discussion

22.4.1 Rationale for data warehousing

Data warehousing is based on three basic observations:

1. Normalized data models are difficult for business users to understand and navigate. There are better methods of storing and representing data for decision support applications.
2. The computational load placed on an operational system by a decision support application may be considerable. In other words, it is conceivable that a middle manager tucked away in a cubical somewhere could bring an organization’s

FIGURE 22.11: Answering a business question using a dimensional data model.



primary operational system to its knees with a well-intentioned but poorly designed query. It is better to isolate mission-critical transaction processing systems from the end-user computing revolution.

3. Many aspects of an organization's operations have an implicit time element that is ignored by the transaction processing system. An example of this in the order entry system is the inventory level (`QtyOnHand`) of each product.

Up until the early 1990s, vendors of databases and transaction processing applications were insisting that their transaction processing systems could do both. This is slowly changing as users and vendors adopt a more pragmatic stance.

22.4.2 The first law of data warehousing

There is one simple design rule that dominates the design and implementation of data warehouses: *disk space is cheap; time is expensive*.

Time, as it is used here, does not mean computational processing time. It means the time of the decision maker who is waiting for a query to return a result. One of the implications is that hardware vendors sell a lot of expensive gear. Very large arrays of hard drives and parallel processing machines are all the rage in data warehousing.

22.4.3 Multiple fact tables

Assume that your firm processes several thousand orders per day and that many of the decision makers in the organization are concerned with monthly measures of performance broken down by region. Although it is certainly possible to get this information by projecting a coarse-grained measure of time (e.g., month) into the query and using the `GroupBy` operator to calculate monthly sales totals, this approach requires a considerable amount of processing.

Given the first law of data warehousing, a better approach involves a straight trade-off between disk space and query performance: create a second fact table with pre-computed monthly totals. In practice, it is not uncommon to see multiple fact tables containing the same "fact" but with different levels of aggregation precomputed. This is one reason that firms often have multiple terabyte data warehouses.

22.5 Application to the assignment

68 Ensure you have implemented all the extraction queries discussed in this lesson.

69 Set the primary key for each dimension table and the fact table.

ACCESS automatically creates indexes for primary keys so you do not need to worry about indexing your tables manually. Although indexes increase the size of your database, the retrieval of records in an indexed table is orders of magnitude faster than in an un-indexed table.

Lesson 23: Using multidimensional data

23.1 Introduction: Reporting, OLAP, and data mining

In this lesson, we are going to briefly explore the ways in which multidimensional data can be queried and manipulated to provide answers to business questions. Of course, an in depth discussion of reporting, on-line analytical processing (OLAP) and data mining are well beyond the scope of the lesson. Instead, the objective here is simply to introduce terminology and some simple yet powerful decision support tools.

23.1.1 An example

To help sell its INTELLIGENT MINER data mining software, IBM uses the example of SAFEWAY PLC—the third largest chain of retail food stores in the United Kingdom. The SAFEWAY chain consists of more than 410 stores across the UK, 70,000 employees, and 25,000 product lines.[1] Point-of-sale scanners within the stores capture the details of about eight million transactions per week. This corresponds to a weekly addition to the data warehouse of roughly 500 MB.

The problem faced by organizations like SAFEWAY is that collecting data is one thing; knowing what to do with the data is another. For example, in an analysis of their scanner data, SAFEWAY management discovered that one particular type of cheese was not widely bought—indeed, it was ranked 209th within its product group. However, a second analysis using data mining software uncovered an interesting relationship: people who bought that particular brand of cheese also bought high-margin items (such as premium wines) on the same visit.

The lesson: a superficial level of data analysis led to the recommendation to de-list an unpopular product. However, a deeper analysis indicated that the company's most profitable customers were buying the product as a complement to high-margin products. Thus, it is possible that de-listing the cheese could have costly repercussions.

23.1.2 Tools for data analysis

The wine and cheese example begs the question: does one need hundreds of thousands of dollars worth of data mining software to perform "deep analysis" of data? The answer is: probably not (although it couldn't hurt). What

1. Source: IBM's web site and *DB2 magazine On-Line*, Vol. 2, No.1, Spring 1997.

one does need, however, is plenty of high-quality data organized into an appropriate form for decision analysis (recall Lesson 22), some basic querying skills, and a solid understanding of how the business works.

23.2 Learning objectives

- exploit the dimensionality of data warehouse data to create crosstab reports
- use constraints and different GroupBy fields to drill down to finer granularity
- create more complex queries to answer specific business questions
- build a pivot table in EXCEL based on a star schema query
- gain experience with basic OLAP concepts such as pivoting and drilling down

23.3 Exercises

In this lesson, you will use the NORTHWIND TRADERS data warehouse you created in Lesson 22 to create more complex queries and perform some *manual* data mining.

23.3.1 Exploiting dimensionality

1. Modify the star schema query you created in Section 22.3.6 to include *all* the fields from each dimension table.

2. Save the resulting query as `qryStarSchema` and close it.

We will use `qryStarSchema` as the basis for other queries so that we do not have to keep re-creating the star schema relationships.

3. Create a new query. When the "Show Table" dialog appears, click the **Queries** tab and select `qryStarSchema`.

4. Save the query as `qrySalesAnalysis`.

5. Use the aggregation techniques you learned in Section 22.3.8 to answer the following question:

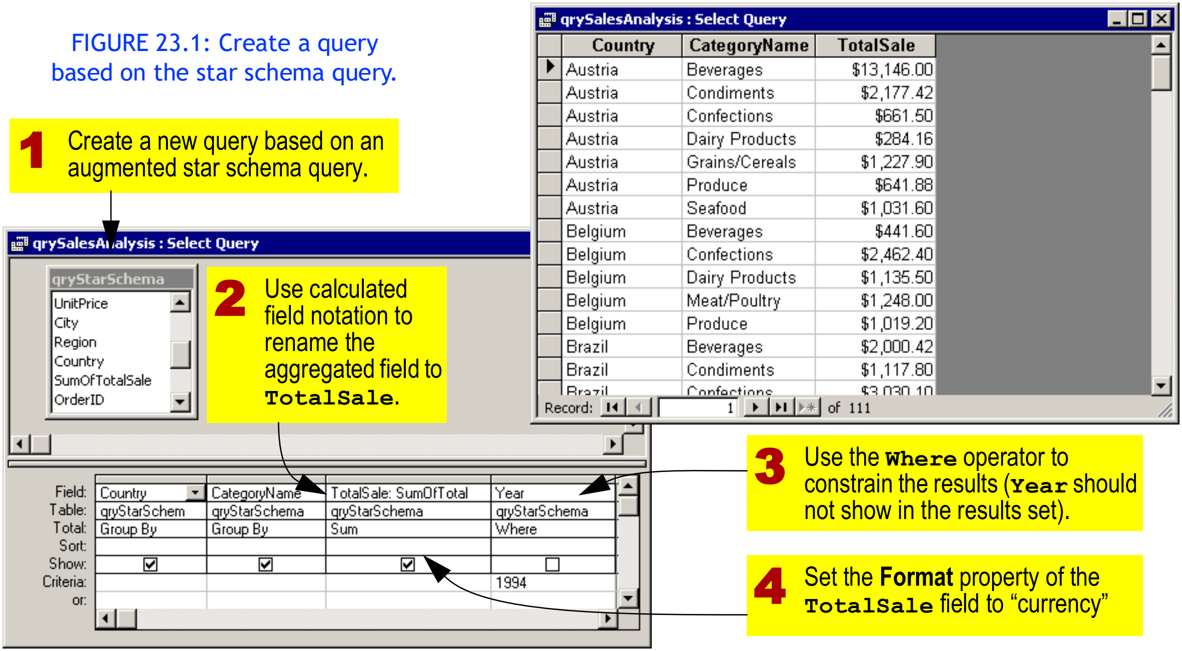
   Which categories of products were selling in which countries in 1994.

If your copy of the NORTHWIND TRADERS database does not have many records from 1994, select a year for which there is more data.

6 Set the **Format** property of the summed field to “currency”.

7 Inspect the results, as shown in Figure 23.1.

FIGURE 23.1: Create a query based on the star schema query.



If you do not like the default field name **`SumOfTotalSale`**, you may use a calculated field to rename the field within the query, for example: **`TotalSale: SumOfTotalSale`** (see Figure 23.1).

23.3.1.1 Creating a crosstab

As Figure 23.1 illustrates, the standard layout of a query results set—with fields as columns and records as rows—is not ideal for viewing multidimensional data. A better way to present two-dimensional relationships is using a crosstab query.

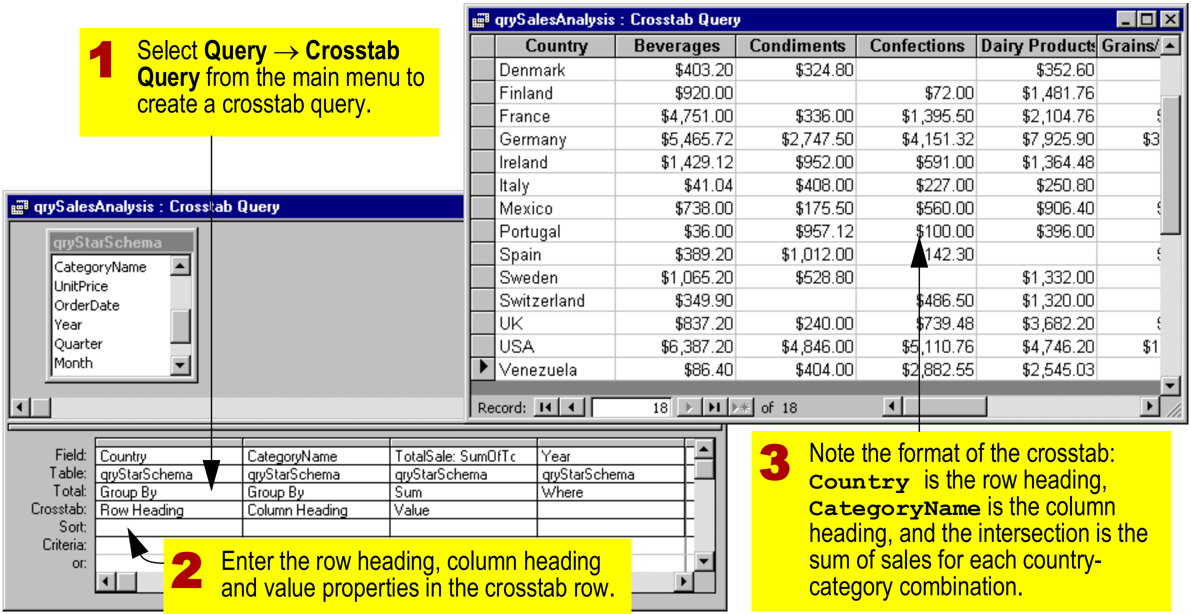
8 Return to design view and select **Query → Crosstab Query** from the main menu. A new "crosstab" row should appear in the query definition grid.

9 In the "crosstab" row, select "Row Heading" under `Country` and "Column Heading" under `CategoryName`.

10 Select "Value" under `TotalSale`.

11 View the results, as shown in Figure 23.2.

FIGURE 23.2: Use a crosstab query to organize two-dimensional data.



Instead of having field names as column headings, the crosstab query has *values* of the `CategoryName` field as column headings. The cells in the crosstab query show the total sales for each combination of country and product category.

Crosstabs are limited to two-dimensional relationships. Pivot tables (introduced in Section 23.3.3) introduce tricks for displaying additional information within the constraints of the two-dimensional format.

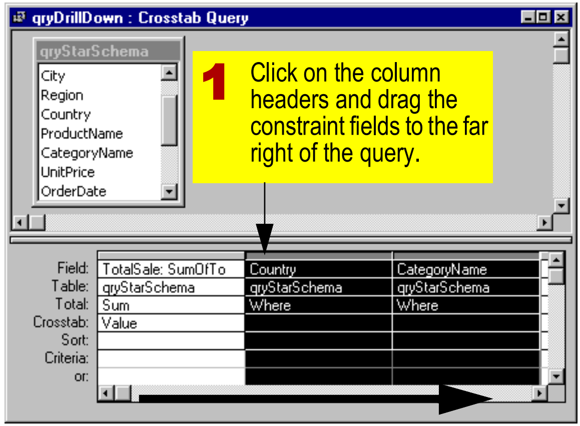
23.3.1.2 Drilling up and down

Taking a close look at the sales analysis query, you may notice that in Venezuela, sales of dairy products is high relative to the sales of products from other categories.[1] By adding some constraints and changing the fields used for grouping, it is possible to explore this issue in greater depth. This process is called drilling down to a finer level of granularity.

12 Use the **File** → **Save As/Export** menu command to save a copy of the query as `qryDrillDown`.

13 Use the grey bar above the `Country` and `CategoryName` fields in the query definition grid to drag the two fields to the far right of the query, as shown in Figure 23.3.

FIGURE 23.3: Move your constraint fields to the far right of the query.



The order of the fields has no effect on the operation of the query. Moving the fields simply helps you keep track of which fields appear in the crosstab and which are merely used as constraints.

[1] The data in the NORTHWIND TRADERS database is presumably fictitious.

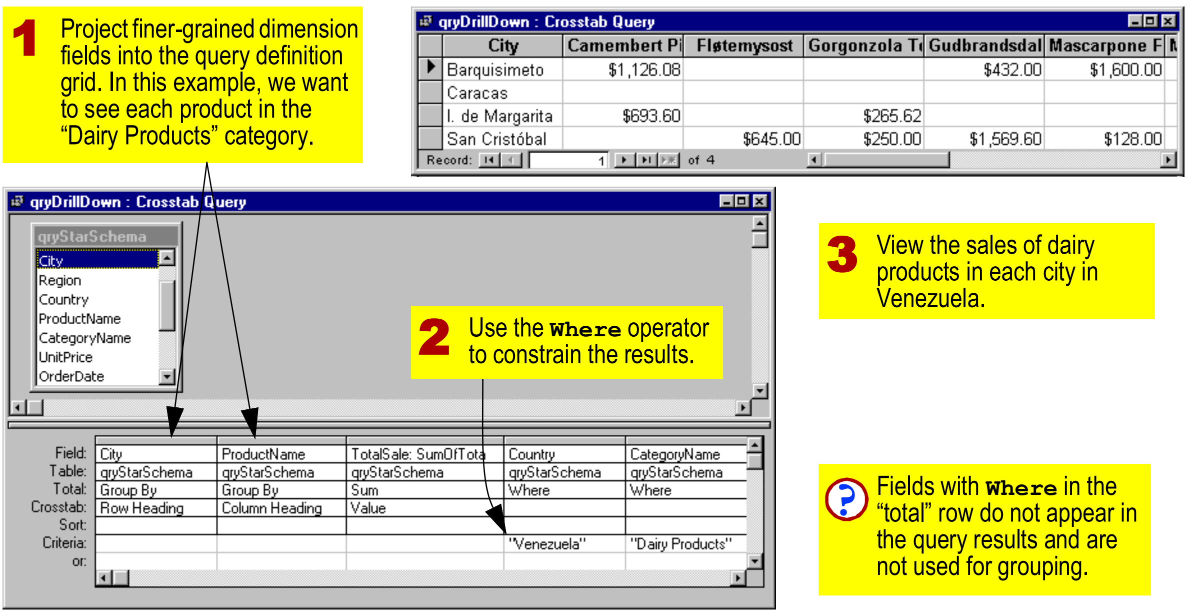
14 Use the `Where` operator in the totals row to specify constraints on the results set. The purpose of the constraints in this case is to limit the results to the sales of dairy products in Venezuela.

15 Project the `City` field into the query. Set the “group by” and “row heading” options.

16 Project the `ProductName` field into the query. Set the “group by” and “column heading” options.

17 Run the query as shown in Figure 23.4.

FIGURE 23.4: Analyze the sales of dairy product in Venezuela.



It is possible to use the detailed information to determine which city-product combinations are driving the sales of dairy products in Venezuela.

23.3.2 Manual data mining

In addition to manual drill down, it is possible to use query tools to manually explore more complex relationships in your data. An obvious relationship that one would expect to find in the context of organizations such as SAFEWAY PLC or NORTHWIND TRADERS is complementarities between products. Products are economic complements if they are worth more to the consumer together than individually. One means of estimating the strength of complementarity using sales data is to perform a basket analysis.

23.3.2.1 Counting the number of matches

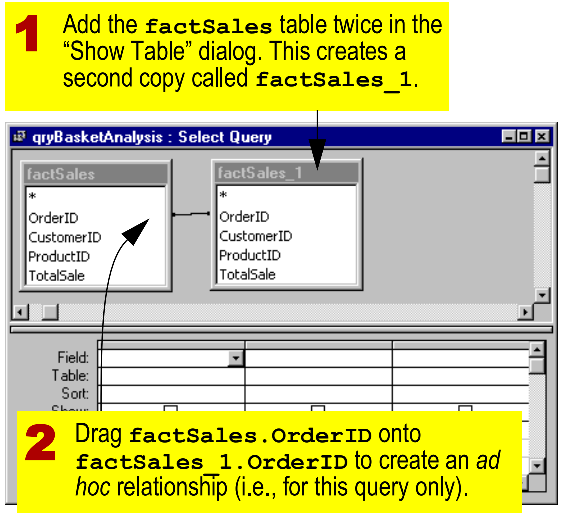
In this section, you will create a new query to count the number of times that a pair of products appears in the same order. This query is a bit trickier since we are joining a table with itself.

18 Create a new blank query called `qryBasketAnalysis`.

19 Project the `factSales` table into the query *twice*.

20 Create an *ad hoc* relationship between `factSales.OrderID` and `factSales_1.OrderID`, as shown in Figure 23.5.

FIGURE 23.5: Create a query based on the fact table joined to itself.



Think for a moment about the meaning of the join in Figure 23.5: The `factSales` table contains data at the granularity of individual order details. Thus every order detail in the `factSales` table is matched with every order detail in the **factSales_1** table that has the same value for **OrderID**.

21 Add the **dimProducts** table to the query twice and ensure the relationships between the table are set (see Figure 23.6).

22 Enable the totals feature and project the **ProductName** field from both Products tables. These are the **GroupBy** fields.

23 Project **factSales_1.ProductID** and set the **Count** operator in the "total" row.

Any field from either table may be used for counting.

24 Project **factSales_1.ProductID** into the query a second time and set its "total" row entry to **Where**. Enter a criterion to ensure that the query does not count the number of time that a product appears with itself.

25 Set the query to sort by the counted field in descending order. In this way, the products with the highest number of matches will sort to the top.

26 Examine the results, as shown in Figure 23.6.

There are two things to notice about this query:

1. Each pair of items appears twice since the query counts the number of times Product_1 appears with Product_2 and the number of times Product_2 appears with Product_1.
2. The measure used—number of orders containing both products—is not particularly useful since the volume sold varies with each product.

In the next section, you will create a new query that reports the *relative frequency* with which each product appears with every other product.

23.3.2.2 Getting the total number of orders for each product

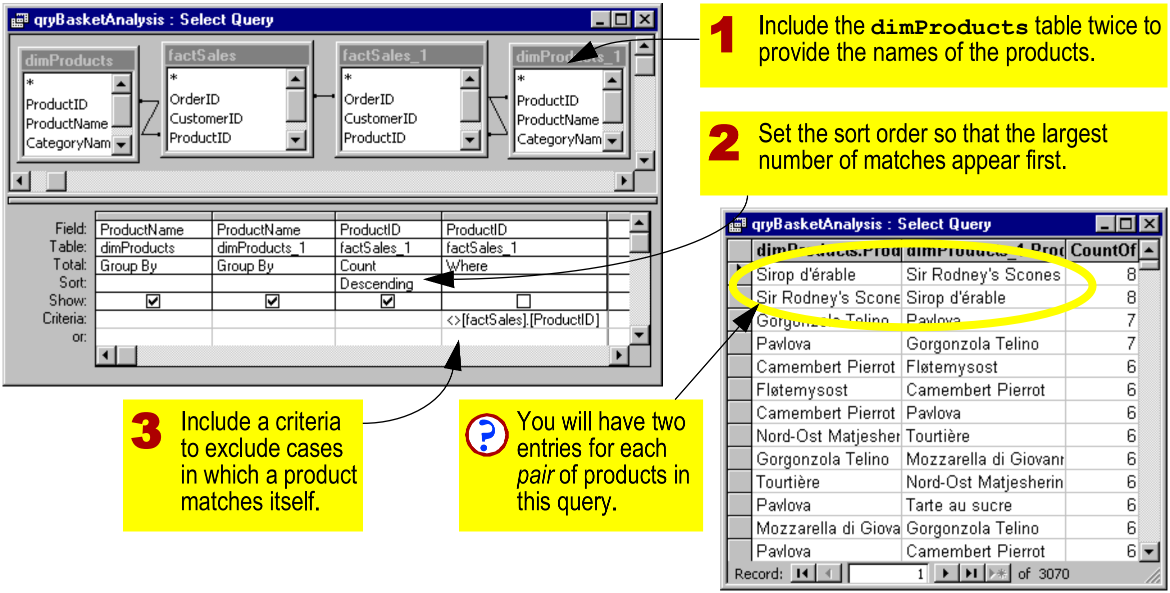
To calculate relative frequency, you must first determine the number of orders in which each product appears.

27 Create a new query called **qryOrdersPerProduct** based on the **factSales** table.

28 Use the **GroupBy** and **Count** operators to count the number times each product appears in an order.

29 Verify the resulting query, as shown in Figure 23.7.

FIGURE 23.6: View the results of the basket analysis query.



30 Save and close `qryOrdersPerProduct`.

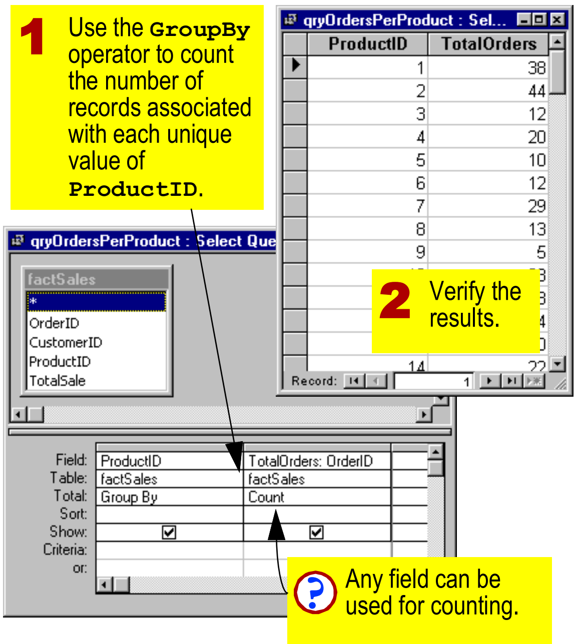
23.3.2.3 Calculating relative frequency

31 Open `qryBasketAnalysis` in edit view and add `factSales.ProductID` to the far left of the query definition grid. Save it and close the query.

32 Create a new query based on `qryBasketAnalysis` and `qryOrdersPerProduct` (see Figure 23.8).

When creating complex queries, it is sometimes useful to exploit the fact that you can base queries on other queries.

FIGURE 23.7: Count the number of times each product appears in an order.



This permits to you decompose the problem into manageable chunks.

33 Create an *ad hoc* relationship between `qryBasketAnalysis.ProductID` and `qryOrdersPerProduct.ProductID`.

34 Project both `ProductName` fields into the query.

To keep the semantics of the query clean, ensure that `dimProducts.ProductName` rather than `dimProducts_1.ProductName` is at the far left of the query definition grid.

35 Create a calculated field called `Frequency` which divides the number of matches by the number of orders for the product.

36 Right-click the calculated field and set its **Format** property to “percent”.

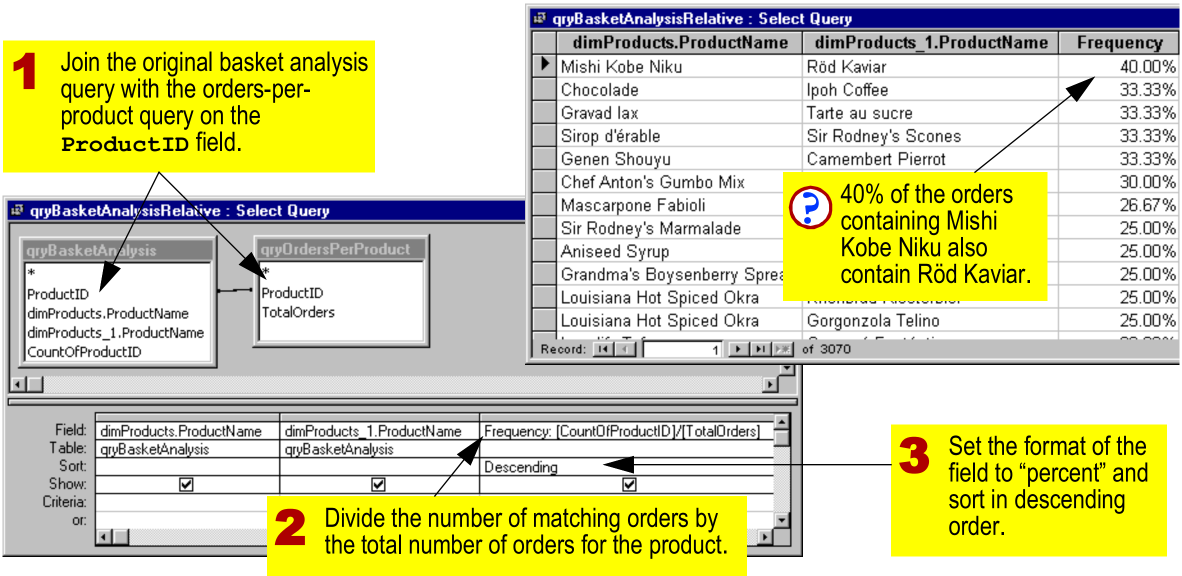
37 Sort on `Frequency` in descending order.

38 Verify the results, as shown in Figure 23.8.

`Frequency` refers to the percentage of orders containing Product_1 (in the first column) that also contain Product_2 (in the second column). For example, 40% of the orders that contain “Mishi Kobe Niku” also contain “Röd Kaviar”.

As in automatic data mining, interpretation requires a solid underlying knowledge of the business and the ability

FIGURE 23.8: Calculate the percentage of orders containing the product in first column that also contain the product in the second column.



to construct a story around the putative relationship.

Spurious relationships can and do occur in data. Good judgement must be used to distinguish real gems of new knowledge from noise.

To make the query in Figure 23.8 even more useful, you could order the results by the total sales. This would allow you to perform the type analyses performed by SAFEWAY described Section 23.1.1. This is left as an exercise.

23.3.3 Exploring the data using pivot tables

The data shown in Figure 23.8 might be better shown as matrix using the crosstab functionality of ACCESS. However, a more flexible way to view any type of multidimensional data is to use the pivot table feature of MICROSOFT EXCEL.

23.3.3.1 Preliminaries

The most elegant way to use the pivot table feature is to have EXCEL access the data in the data warehouse directly. In this section, you will set up an ODBC link to the augmented `qryStarSchema` query you created in Section 23.3.1.

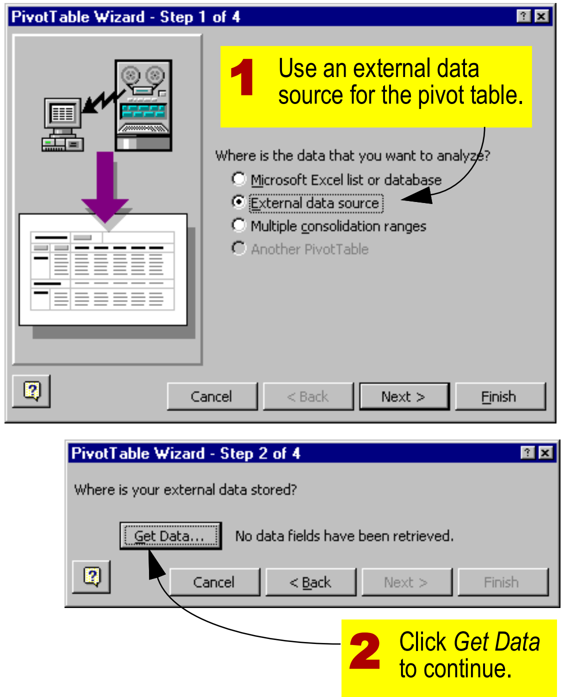
39 Create a new EXCEL workbook and save it as `OrderEntryPivot.xls`.

40 Select **Data** → **Pivot Table Report** from the main menu.

41 When prompted, select "external data source" and "get data" as shown in Figure 23.9.

If you did not install MICROSOFT QUERY, you will have to re-run the setup program from your OFFICE CD-ROM and add it. See Section 9.3.3 for additional information.

FIGURE 23.9: Create a new pivot table within EXCEL using external data.



23.3.3.2 Creating an ODBC connection

42 From the "choose data source" dialog, select "new data source".

43 In the "create new data source" dialog, give your data source connection an easy-to-read name, such as "Order Entry Data Warehouse".

44 Since your data warehouse is a MICROSOFT ACCESS database file, choose the appropriate driver, as shown in Figure 23.10.

45 Press the **Connect** button to continue.

In the next dialog, you are asked to enter specific information about the ACCESS file that you are using as a data source. If you were using (say) a client/server ORACLE database instead, you would get a dialog tailored to making a client/server connection. The sequence of dialogs is shown in Figure 23.11

46 In the "ODBC MICROSOFT ACCESS setup" dialog, press the **Select** button.

47 Navigate to your `OrderEntryWarehouse.mdb` file and press Press **OK**.

48 In the "Create New Data Source" dialog, select `qryStarSchema` as the default "table".

49 Select **OK** twice.

50 From the "query wizard—choose columns" dialog, click on `qryStarSchema` and press the **>** button to include all the fields in the pivot table, as shown in Figure 23.12.

51 Since there is no need to worry about sorting or filtering the data at this stage, press **Next** until you encounter the **Finish** button.

23.3.3.3 Creating the pivot table

You are now back to the EXCEL pivot table wizard. Although you may not realize it, what you just did is set up a "file DSN" ODBC connection called "Order Entry Data Warehouse".

To view or modify your new DSN, you can open the WINDOWS control panel, double click the data sources icon, and select the "file DSN" tab. We discussed the use of file DSNs for ODBC in Section 8.3.3.

52 Press **Next** to move on to Step 3 of the pivot table wizard.

53 Drag the `SumOfTotal` "chicklet" into the "data" area in the center of pivot table.

54 Drag `Quarter` into the "column" area and `CategoryName` and `ProductName` into the "row" area.

55 Finally, drag `Country` into the "page" area. The result should resemble Figure 23.13.

56 In Step 4, indicate where you want the new pivot table to be located and press **Finish**.

FIGURE 23.11: Specify the location of the source ACCESS database.

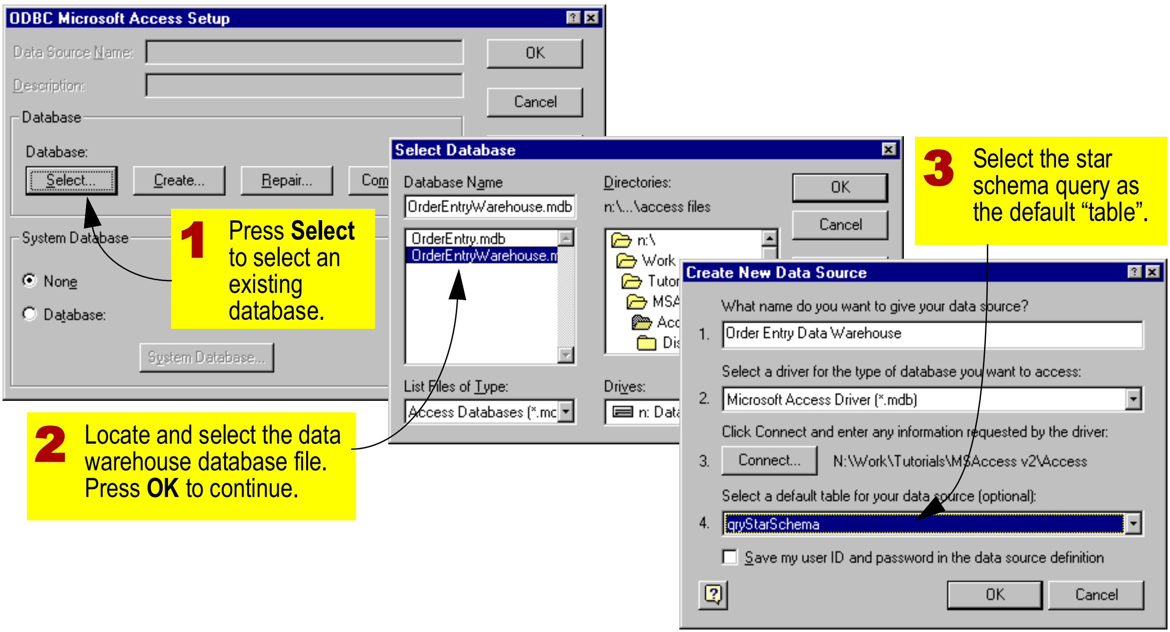
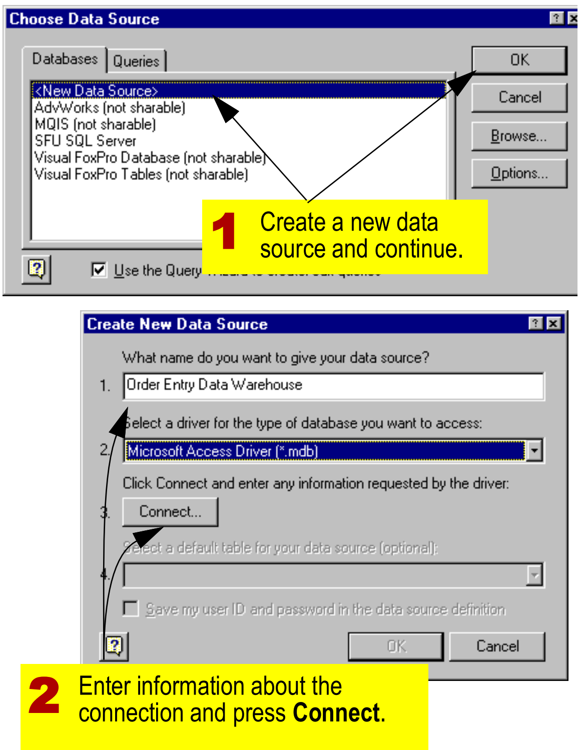


FIGURE 23.10: Create a new data source.



23.3.3.4 Altering the format

Before playing with the pivot table, we should format the data cells to show currency:

57 Right-click anywhere on the pivot table and select "wizard". This returns you to Step 3 of the wizard.

58 Double-click the `Sum of SumOfTotal` chicklet in the "data" area.

59 Press the Number button on the right side (as shown in Figure 23.14) and select "currency".

60 Press **Finish** to return to the pivot table.

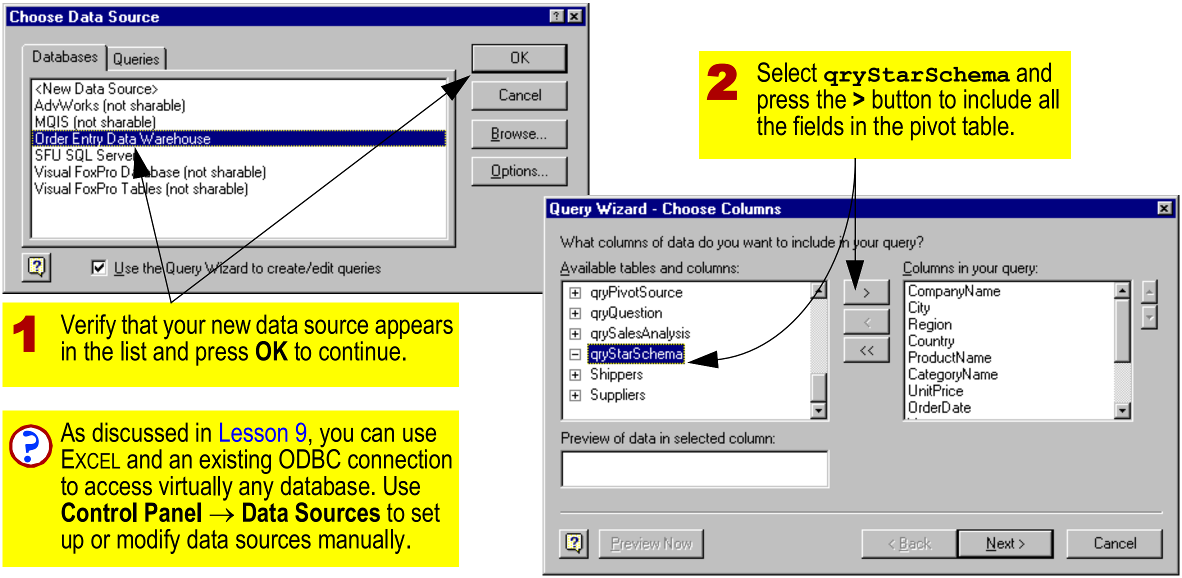
23.3.3.5 Using your pivot table

The best way to learn about pivot tables is to play with one. Clearly, what you have at this stage is similar to a crosstab query. However, unlike a crosstab, you can easily explore your multidimensional data by pivoting around the fact in the middle. For example:

61 Drag the `CategoryName` chicklet off the pivot table and drop it. The field should disappear from the pivot table.

Remember, you can always right-click on the pivot table to get back to Step 3 of

FIGURE 23.12: Select the fields from ***qryStarSchema*** to include in the pivot table.



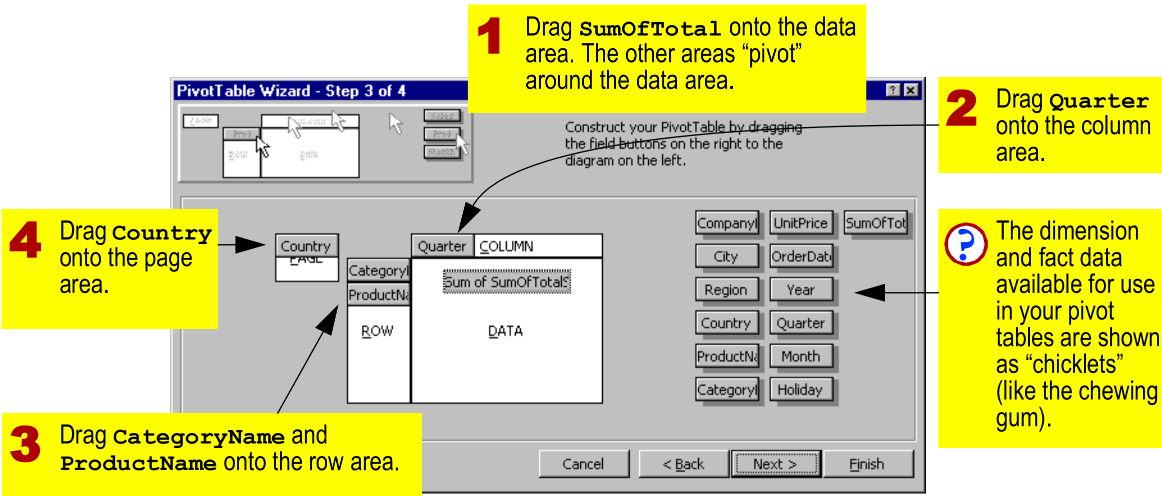
the wizard. In this way, you can easily change the dimension fields that appear in the row, column, and page areas.

62 Drag the `Country` chicklet down beside `Quarter` but then drag `Quarter` to the page area (at the top-left of the pivot table).

63 Click on the `Quarter` drop-down list and select “Q4”. This limits the data in the table to orders placed in the fourth quarter.

64 Double-click on any fact in the table to drill down on specific order detail records from the underlying `qryStarSchema` data source.

FIGURE 23.13: Drag fields to the appropriate location on the pivot table grid.



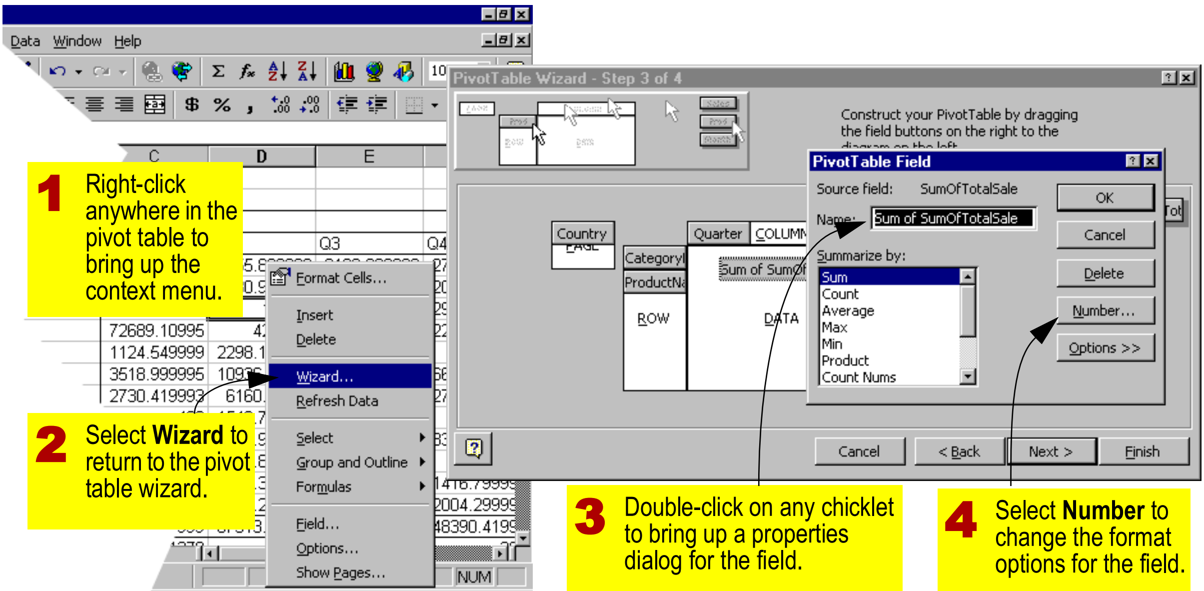
This drill down feature is a bit clumsy since you create a new worksheet every time you double-click a number. You can right-click on the sheet's name tab to delete it once you have examined the detailed data.

23.4 Discussion: Commercial OLAP tools

There are a number of commercial OLAP tools on the market (e.g., ARBOR SOFTWARE's ESSBASE, COGNOS POWERPLAY, and ORACLE's EXPRESS SERVER) that are designed to facilitate interactive analysis with large amounts of multidimensional data.

Such products have clear advantages over EXCEL in terms of scalability and ease of use. For

FIGURE 23.14: format the pivot table data as currency.



example, in EXPRESS SERVER, you simply double-click on any value or bar chart to drill down to a finer level of granularity. The software dynamically re-queries the data warehouse, thereby eliminating the manual process you performed in Section 23.3.1.2. Of course, EXCEL has the virtue of costing much less than $3,000 per user.

23.5 Application to the assignment

65 Modify your `qryOrdersPerProduct` so that it also calculates the total sales volume of each product.

HINT: You can add another column to the query and use the `Sum` operator to get this value.

66 Determine if their is any evidence of a complementary relationship between NORTHWIND'S best selling products and other products it sells.

Lesson 24: An introduction to HTML

24.1 Introduction

Hypertext markup language (HTML) consists of a set of "tags" that tell a web browser how a web page should look and operate. For example the pair of tags `<STRONG>Introduction</STRONG>` tells a browser to display the word "Introduction" in a "strong" format (typically bold).

There are couple of important things to notice about this example:

1. The tags themselves are text. HTML is a text-only standard in which some of the text in a document is content and the rest of the text is used to tell the browser how to display the content. The angled braces "<" and ">" are use to differentiate HTML tags from the rest of the document.
2. It is up to the browser to interpret the tags and render the text accordingly. Although this approach minimizes the amount of information that must be transferred over the network, it leads to some inconsistencies. For example, a tag such as **<STRONG>** may mean one thing in NETSCAPE NAVIGATOR and something slightly different in MICROSOFT'S INTERNET EXPLORER. The result is that you, as the HTML author, have *incomplete* control over how your page looks when viewed in different browsers.

HTML and HTML extensions (such as Dynamic HTML) are evolving in two areas: new functionality and greater control over how the document looks in the browser. Although HTML is nominally a standard, new tags and features (which may not be supported by all browsers) are continually being introduced. When in doubt, check the official standard at www.w3.org.

Although general purpose document authoring tools (like MICROSOFT WORD) and special WYSIWYG[1] HTML editors (like MICROSOFT FRONTPAGE) can write HTML for you, and although these tools are improving, there are still occasions when you must open the hood and deal with the HTML directly. In this lesson, we will use a simple text editor and write all our HTML from scratch.

1 What You See Is What You Get

24.2 Learning objectives

- understand the structure of an HTML document
- use HTML tags to format web-based content
- create hyperlinks between two documents
- use HTML tables to display tabular data and format pages

24.3 Exercises

In this lesson, you will learn about a few basic tags and create some very simple HTML pages.

If you are interested in becoming a *real* web page designer, there are many HTML tutorials and resources on the Internet. We are merely skimming the surface here.

24.3.1 Tag basics

Most tags are found in pairs. For example, the tag `<EM>` means that all text which follows will be emphasized (italicized in most browsers). When the closing tag, `</EM>`, is encountered, the emphasis is turned off. The closing tag in the pair always starts with a slash.

It is possible to nest tags. For example, to get text that is strong *and* emphasized (bold and italic in most browsers), your would nest the tags in the following manner:

```
NL <STRONG><EM>This text shows as bold
   italics is most browsers.
   </EM></STRONG>
```

The amount of whitespace does not matter in HTML—there is either “no space” or “one or more spaces.” The latter is simply interpreted by browsers as a single space. As a result, the HTML below is equivalent to the previous example:

```
NL <STRONG>
NL   <EM> This text shows
NL         asbold italics in most
NL
NL     browsers.</EM>
NL <STRONG>
```

HTML tags are case-insensitive. Thus, the tags `<STRONG>` and `<strong>` are identical. Naturally, the capitalization of your content does matter.

24.3.2 HTML documents

An HTML document is simply an ASCII text file with a htm or html extension.[1] Within the file, there should be three pairs of tags which define the following sections:

1. The overall HTML document — everything between the **<HTML>** and **</HTML>** tags.
2. The header section within the HTML document — everything between the **<HEAD>** and **</HEAD>** tags. The header section contains information about the document and is not visible in the browser.
3. The body section within the HTML document — everything between the **<BODY>** and **</BODY>** tags. The body section contains the visible content of the document.

Thus, the structure of an empty HTML document is:

```
NL  <HTML>
NL    <HEAD>
NL      ...
NL    </HEAD>
NL    <BODY>
NL      ...
NL    </BODY>
NL  </HTML>
```

1 MICROSOFT DOS and WINDOWS 3.*x* can only handle three-letter extensions. As such, MICROSOFT continues to push the htm naming convention. However, since the Internet grew up on UNIX (which has always supported longer filenames), the four-letter html extension is more common.

24.3.2.1 Preliminaries

Web applications typically run on specialized web servers with high-speed fixed connections to the Internet. Web development, in contrast, typically takes place on the developer's lowly PC. In this scenario, you are the developer. In order to get your content and programs on to the web server, you have to transfer (or "publish") your HTML files over the network to your ISP's machine[1]. The transfer is normally accomplished using a utility program based on the FTP (file transfer protocol) standard.

The constant transfers can become a bit tedious as you continually create, upload, test, and revise your content. For this reason, a good FTP client is a necessity. Alternatively, you may want to enable a web server on your desktop machine. In this way, you can create and test your application on a single machine and transfer the files to the "production" web server when the files are complete. Enabling a local web server is discussed in additional detail in Section 24.4.2.

The important thing to keep in mind is that you will always have two copies of the files you

1 ISP stands for Internet service provider. ISPs typically offer two broad class of services: access for dial-up users and hosting of content on their servers.

create: the **local copy** that you work on during development and the **published copy** on the production web server that the whole world can access.

Although maintaining two identical copies of every file in your web application can be tedious, at least you can use your local copy as a backup if something happens to your ISP's server (and vice-versa).

1 Set up a directory on your machine for the local copy of your web application.

It is up to you how you organize your directory. However, it is customary to create a "images" subdirectory and a "private" subdirectory. The images subdirectory contains the files for graphic elements (pictures, icons, buttons, etc.). The private directory can contain resources (e.g., a database file) that outside users are prevented from accessing directly.

24.3.2.2 Creating a document

2 Use NOTEPAD or some other text editor to create a document with the **<HTML>**, **<HEAD>**, and **<BODY>** tags discussed in Section 24.3.1

You cannot use a word processor like MICROSOFT WORD to edit HTML directly because word processing programs save documents in proprietary formats rather than plain ASCII text. Although you can use the **Save As** command to explicitly save the file as plain text, most people find it is easier to simply use NOTEPAD or some other text editor.

3 Save the file as **MyFirst.html**, as shown in Figure 24.1.

4 In the document's header, add a title:

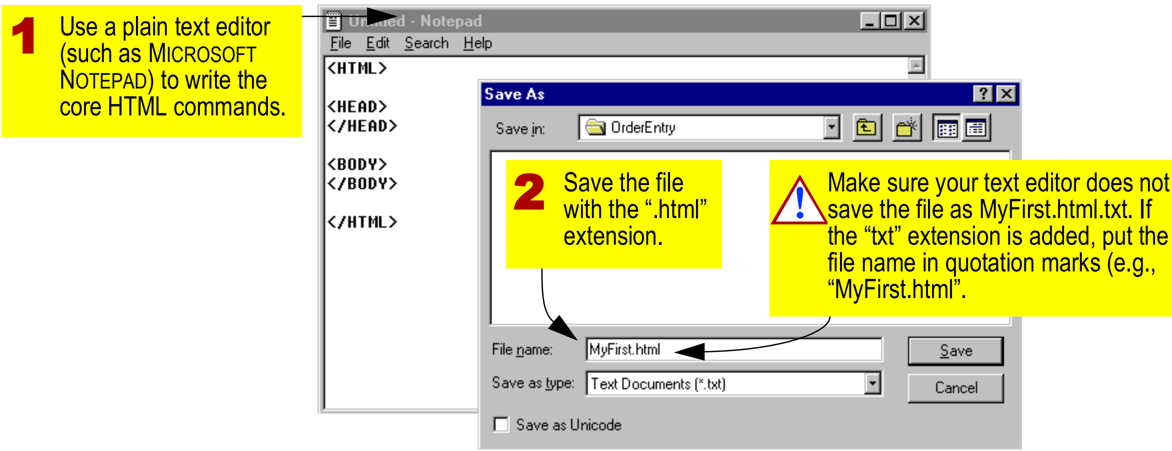
```
NL <HEAD>
NL <TITLE>Kitchen Supply Co.
   Extranet</TITLE>
NL </HEAD>
```

Although the header section is optional and does not show in the browser window, Internet search engines often use information in the header when indexing and display search results. As such, you should always provide a meaningful title for each of your pages.

5 In the document's body, add a heading and welcome message to users, for example:

```
NL <HTML>
NL <HEAD>
```

FIGURE 24.1: Save the skeleton HTML document created in NOTEPAD.




```
NL  <TITLE>Kitchen Supply Co.
    Extranet</TITLE>
NL  </HEAD>
NL  <BODY>
NL  <H1>Welcome to the Kitchen Supply
    Co. Extranet</H1>
NL  Please login to gain secure access
    to the system.
NL  </BODY>
NL  </HTML>
```

6 Save the changes. Your file should resemble Figure 24.2.

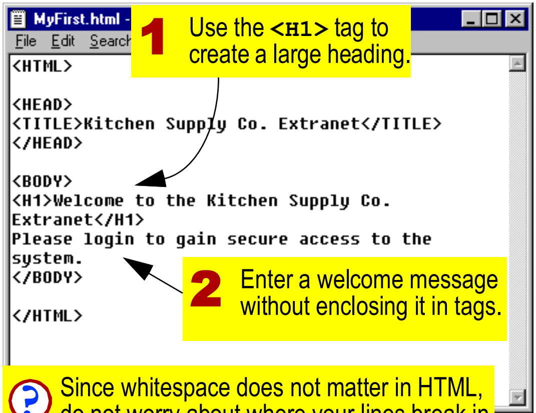
24.3.2.3 Viewing the document

HTML pages are typically transferred to users via a web server. However, it is possible to bypass the web server and open local files directly with your browser.

7 Open your browser and select **File → Open** from its main menu.

⚠ Since there are many different makes and models of browsers in circulation, it is impossible to give specific menu and

FIGURE 24.2: Add a welcome message to the HTML document.

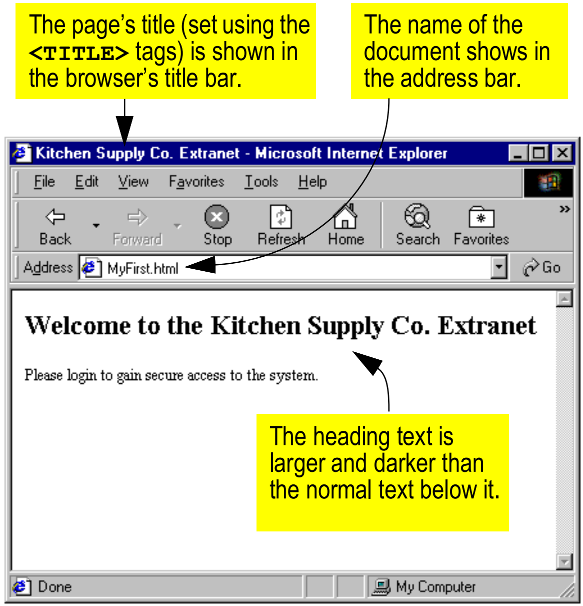


Since whitespace does not matter in HTML, do not worry about where your lines break in the editor. A new line is treated like a single space by the browser.

keystroke commands for browser-based activities. In this lesson, MICROSOFT INTERNET EXPLORER version 5.0 is assumed. If you use a different browser, the command sequences and format of the output shown here may differ slightly from what you see on your computer. However, the underlying principles are the same regardless of the browser used.

8 Navigate to the `MyFirst.html` file and open it. Your page should appear in the browser, as shown in Figure 24.3.

FIGURE 24.3: Preview the HTML document in a web browser.



There are many ways to open a local HTML file in your browser, including double-clicking the file's icon in WINDOWS

EXPLORER or dragging the file's icon onto the browser.

When you compare Figure 24.2 and Figure 24.3, you should get the basic gist of HTML: plain text is rendered into formatted text via the use of special tags.

24.3.3 Adding hyperlinks

Of course, the real power of HTML is not its formatting, but rather its ability to create hyperlinks to other documents and resources on the World Wide Web (WWW). In this section, you are going to create a new page (a list of products) and create a hyperlink to **`MyFirst.html`**.

9 Without closing the current instance of NOTEPAD, open a second instance in a different window (i.e., select **Start → Programs → Accessories → Notepad** from the WINDOWS taskbar).

10 Cut and paste the HTML from **`MyFirst.html`** into the new document and save it as **`ProductList.html`**.

11 Edit the heading to reflect the fact that this page is a list of products for KITCHEN SUPPLY CO.

12 Delete the welcome message and enter the following without tags, as shown in Figure 24.4.

```
NL  <BODY>
NL  <H1>Kitchen Supply Company: List of
    Products</H1>
NL  Go to login page
NL  </BODY>
```

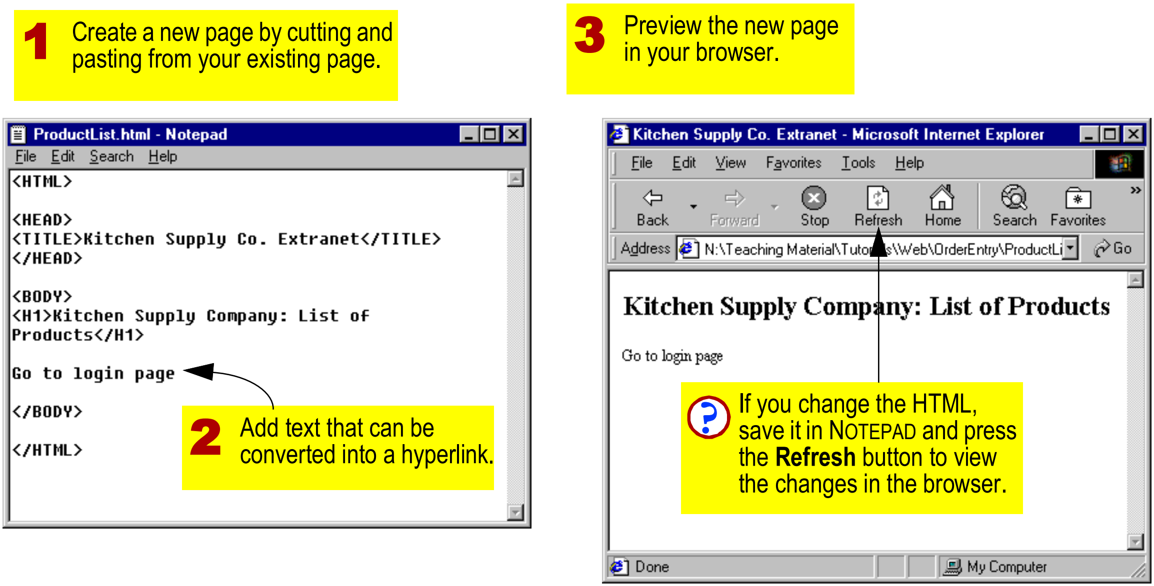
To create a hyperlink, you need to know the Uniform Resource Locator (URL) of the target document. The URL consists of three items of information:

- the protocol to use to access the information (e.g., HTTP, FTP, Telnet, and so on);
- the Internet Protocol (IP) address of the machine on which the document or resource resides (e.g., **`mis.bus.sfu.ca`**); and,
- the name of the document or resource (e.g., **`MyFirst.html`**)

When the target of the hyperlink is located on the same server as the document containing the hyperlink, the URL can be replaced with the relative path and filename of the target.

IP addresses are numeric. For example, to access the DELL COMPUTER site, you type **`http://143.166.82.178`**. Fortunately, you can also use the easier-to-remember

FIGURE 24.4: Create a new page for showing a list of products.



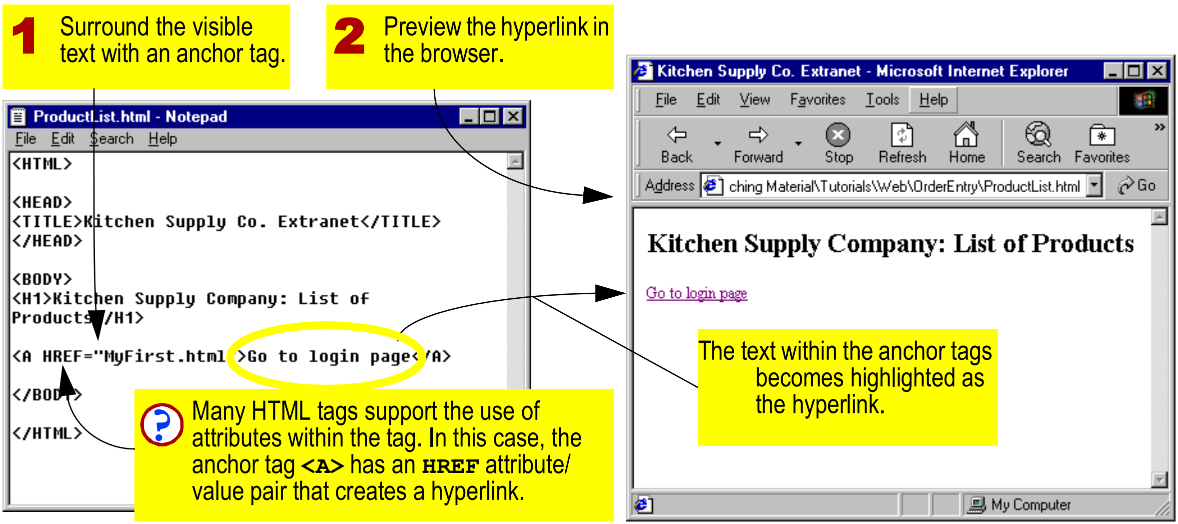
textual address (**`www.dell.com`**) thanks to a world-wide distributed database called the **Domain Name System** (DNS). A DNS lookup translates a textual IP address into its numeric equivalent before sending the request out on the network. The administrator of the **`dell.com`** domain is responsible for assigning textual names to numeric IP addresses and storing this information in a local DNS database. For example, DELL has one IP address registered as **`www.dell.com`** and another registered as **`support.dell.com`**.

13 Use the anchor tag and the **HREF** attribute to transform the plain text in Figure 24.4 into a hyperlink:

```
NL   <BODY>
NL   <H1>Kitchen Supply Company: List of
     Products</H1>
NL   <A HREF="MyFirst.html">Go to login
     page</A>
NL   </BODY>
```

14 Save the document and press the browser's refresh button to view and test the hyperlink, as shown in Figure 24.5.

FIGURE 24.5: Create a hyperlink to another HTML file in the same directory.



If the target document is in a subdirectory relative to the current document, then the name of the subdirectory has to be included:

```
NL   <A HREF="pages/MyFirst.html">
```

In UNIX, directories are separated by slashes (“/”) rather than back-slashes (“\”) as in MS-DOS and WINDOWS. The convention in HTML is to use UNIX-style directory names for URLs, regardless of the platform you are using.

If the target document is in the current document’s parent directory, the double-dot notation has to be used to ascend one level in the directory tree:

```
NL  <A HREF="../MyFirst.hml">
```

The notation “..” is shorthand for the parent of the current directory.

If the target document is on a different machine, then the full URL (including the machine name and directory) has to be used:

```
NL  <A HREF="http://mis.bus.sfu.ca/
    pages/MyFirst.html">
```

24.3.4 The paragraph tag

Since whitespace is ignored in HTML, the only way to create space between paragraphs is to use the paragraph tag, **<P>**. In the original HTML standard, one could use the **<P>** tag without a closing tag. However, the use of opening and closing tag is consistent with other HTML tags and is therefore preferred.

15 Nest a **<P>** tag inside the hyperlink tag:

```
NL  <BODY>
NL  <H1>Kitchen Supply Company: List of
    Products</H1>
NL  <A HREF="MyFirst.html"><P>Go to
    login page</P></A>
NL  </BODY>
```

24.3.5 Using HTML tables

Tables are used within HTML to

- format tabular data, and
- provide page designers with additional control over the layout of the page.

At this early stage, you should not concern yourself with the finer points of page layout. Instead, our focus is on using tables to display lists of data.

16 Add a **<TABLE>**... **</TABLE>** pair underneath the hyperlink.

24.3.5.1 Rows and headings

A table consists of one or more rows and each row consists of one or more cells. Rows are designated using the **<TR>** tag and cells are designated using the **<TD>** tag. Special heading cells (typically bold and centered) are designated using **<TH>** tags.

17 Add a row of headings to the table:

```
NL <BODY>
NL ...
NL <TABLE>
NL   <TR>
NL     <TH>Product ID</TH>
NL     <TH>Description</TH>
NL     <TH>Unit</TH>
NL     <TH>Price</TH>
NL   </TR>
NL </TABLE>
NL </BODY>
```

18 Add a second "detail" row that contains product information:

```
NL <BODY>
NL ...
NL <TABLE>
NL   <TR>
NL   …
NL   </TR>
NL   <TR>
NL     <TD>51 5012</TD>
NL     <TD>Water jug, s.s. w/ice guard,
       2 litre</TD>
NL     <TD>EA</TD>
NL     <TD>$23.50</TD>
NL   </TR>
NL </TABLE>
NL </BODY>
```

19 Save the document and preview it in your browser, as shown in Figure 24.6.

24.3.5.2 Adding tag attributes

It is possible to use tag attributes to control the format of the table and elements within the table:

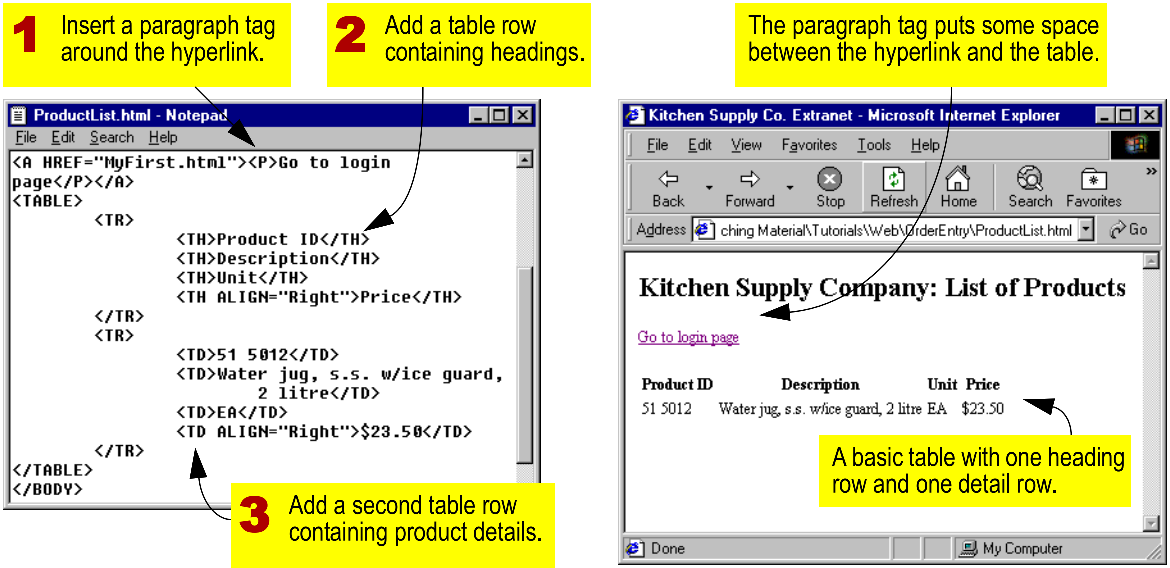
20 Change the format of the table so that it has solid borders and inserts some padding (space) between cells:

```
NL <BODY>
NL ...
NL <TABLE BORDER="1" CELLPADDING="5">
NL   …
NL </TABLE>
NL </BODY>
```

21 Add `ALIGN="Right"` to the `<TH>` tag used for the "price" heading. Add the same attribute to the `<TD>` tag used for the price body rows.

If the attribute value is a single word or number (i.e., if the value contains no spaces), then quotation marks are not required. Thus, it is possible to use `ALIGN=Right` in your `TABLE` tag without having bad things happen. However—as you will see in Lesson 25—it is possible to have attributes with spaces (e.g.,

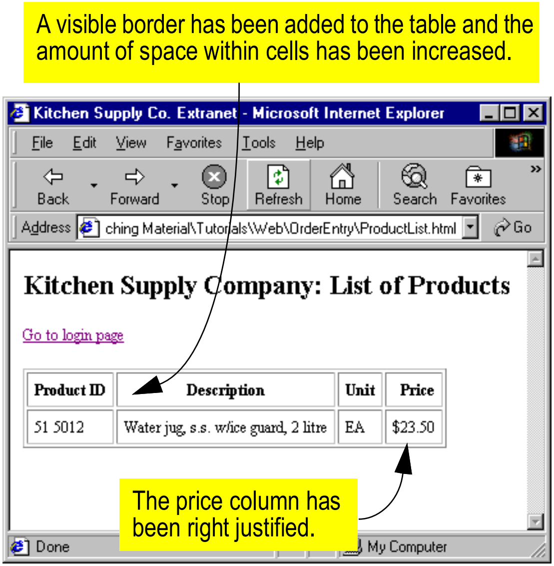
FIGURE 24.6: A basic table to show product information.



VALUE=51 5012). Without quotation marks, the browser will assume the attribute **VALUE** is equal to “51” and will try to interpret “5012” as a different attribute. To avoid such problems, it is considered good practice to put *all* attribute values in quotation marks.

22 Save the changes and view the document in the browser, as shown in Figure 24.7.

FIGURE 24.7: The table with some formatting attributes specified.



24.4 Discussion

24.4.1 Authoring options

Clearly, writing HTML by hand is tedious and error-prone. Fortunately, there are many different tools that can do most of the slug work for you.

24.4.1.1 Dedicated HTML editors

There are many commercial and shareware WYSIWYG HTML editors that are about as easy to use as modern word processors. The editors allow you to use the mouse to format text, create tables, and so on. In addition, some allow you to “round trip”—that is switch back and forth between WYSIWYG and raw HTML modes. Examples of commercial products in this group include Microsoft FrontPage, Adobe PageMill, Macromedia Dreamweaver, and Allaire HomeSite.

24.4.1.2 Word processors

In addition to dedicated HTML editors, word processors themselves are getting better at saving documents in HTML format. The advantage of using a word processor is that you can continue to work within a familiar environment and exploit features that tend to be weak in dedicated HTML editors (for example, spell checking and table editing).

However, it is important to realize that word processors typically support much richer formatting and page layout options than HTML is currently capable of expressing. As such, the limitations of HTML should be kept in mind when authoring documents. Even common enhancements, such as embedded graphics, footnotes, and special formatting can overwhelm the HTML translator and result in an HTML document that bears little resemblance to its source. In addition, most word processors are not very good at round tripping. Adding non-standard HTML tags or scripting code in HTML mode can cause problems when you switch back to word processing mode.

You have to be careful to select an authoring tool that suits your purposes. Many of the tools designed for beginners or casual users take it upon themselves to "fix" your code. If you are a developer and are using scripts and advanced tags, you may find that the tool obliterates your work in its attempt to be helpful.

24.4.1.3 Application development suites

A third way to create web content is to use an "integrated application development suite". This is an ill-defined, emerging class of tools that includes MICROSOFT VISUAL INTERDEV and NETOBJECTS FUSION (MICROSOFT FRONTPAGE and ALLAIRE HOMESITE/COLDFUSION might also be included). Application development suites are much more than HTML editors; they include tools for site management, creation of dynamic content and scripting, database integration, and support for team-based design and implementation. Although these tools are very powerful, they are also complex. Using them effectively requires an organization-wide commitment to the development methodology advocated by the suite.

24.4.2 Setting up a local web server

A web server is simply a program that runs on a computer and "listens" for requests from other programs. In subsequent lessons, access to a MICROSOFT web server is required in order to execute server-side scripts. Fortunately, a full-featured web server may be sitting on your desktop right now.

All versions of WINDOWS since WINDOWS 95 include a scaled-down version of MICROSOFT's flagship web server—INTERNET INFORMATION SERVER (IIS)—on the installation disk. Depending on your version of WINDOWS, the bundled copy of IIS may be referred to using an alias such as "peer web services" or "personal web server".

The terms "personal" and "peer" are intended to drive home the point that the web server included with WINDOWS is meant for development work or as a very small server for a network of colleagues and friends. For industrial-scale web applications, MICROSOFT sells its full version of IIS as part of the BACKOFFICE suite.

24.4.2.1 Advantages of using a local web server

If you are using these tutorials as part of a course and your instructor has set up a web server for you, there is no strict requirement for you to set up a local web server. However, you might want to consider the advantages below before skipping the remainder of this section. For those who do not have an instructor taking care of you, read on.

There are a number of important advantages to having a local web server up and running when you are developing a web application:

1. You do not need a persistent, high-speed connection to the Internet to test your pages.
2. You can edit your files directly on the web server, thereby eliminating the upload step from the create → upload → test → revise cycle.
3. If, during testing, your web application does something so bad that you need to restart the server (not unheard of, unfortunately), you do not impact the other users of the server.

24.4.2.2 Is IIS running already?

Some programs, like MICROSOFT FRONTPAGE, configure the local web server behind the scenes when they install. To see whether IIS is already running on your machine, type the following into the URL window of your browser:
`http://localhost`

The IP address of the local machine is always 127.0.0.1. A small file called `hosts` in your Windows folder maps this numerical address to “localhost”.

If a web page, a blank page, or any page that does not contain errors pops up, IIS is already installed and configured on your computer. If you get an error or get shunted off to a search engine looking for “localhost”, then IIS is not setup on your computer.

24.4.2.3 If IIS is not running

If the results of the localhost test are negative, then you have to have to install IIS. The good news is that it is a very simple procedure. The bad news is that the procedure itself depends greatly on which version of WINDOWS you are running. As such, the first place to start is a search for terms like “web browser” and “peer web services” in the WINDOWS help system

(**Start** → **Help**). At a very general level, the install procedure involves the following:

1. Ensure you have the TCP/IP networking protocol installed (which you do if you use your computer to access the Internet).
2. Install the web server from your WINDOWS installation media. The sequence for installing IIS on a WINDOWS 2000 machine is shown in Figure 24.8 and Figure 24.9.

FIGURE 24.8: Install and administer the IIS web server on your desktop machine (Part 1).

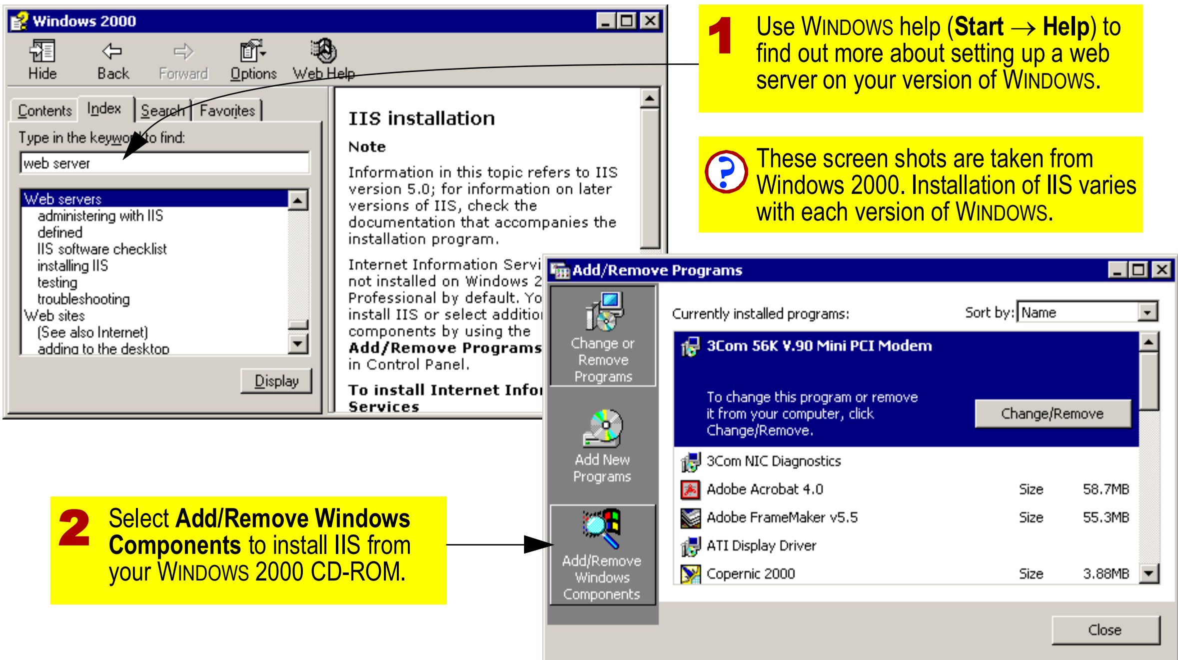
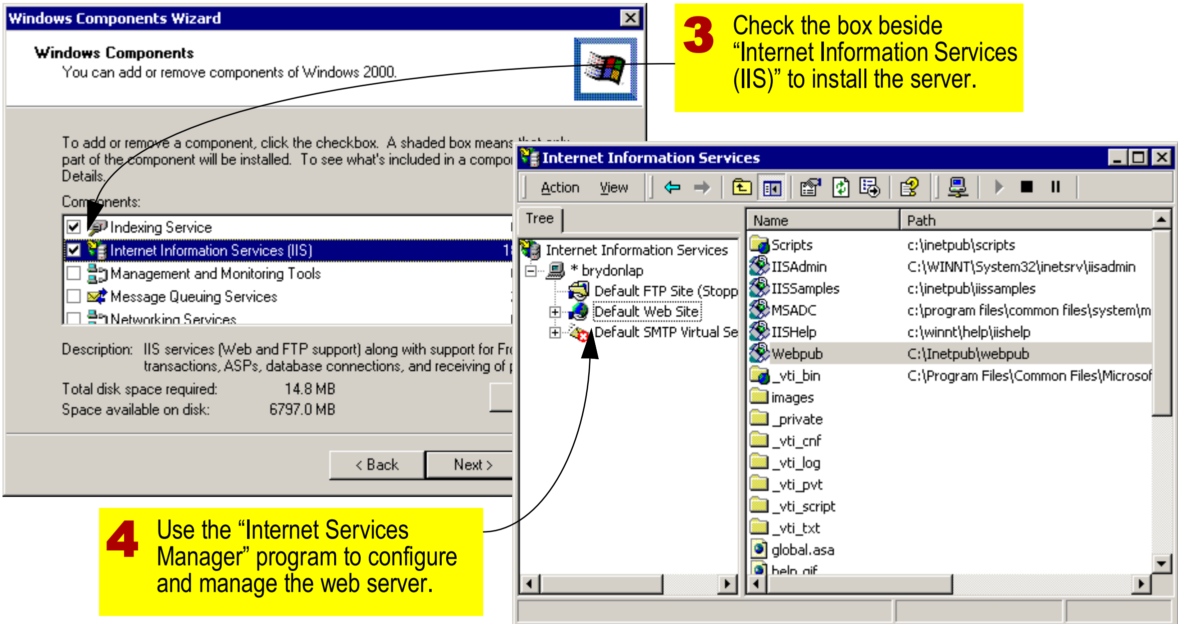


FIGURE 24.9: Install and administer the IIS web server on your desktop machine (Part 2).

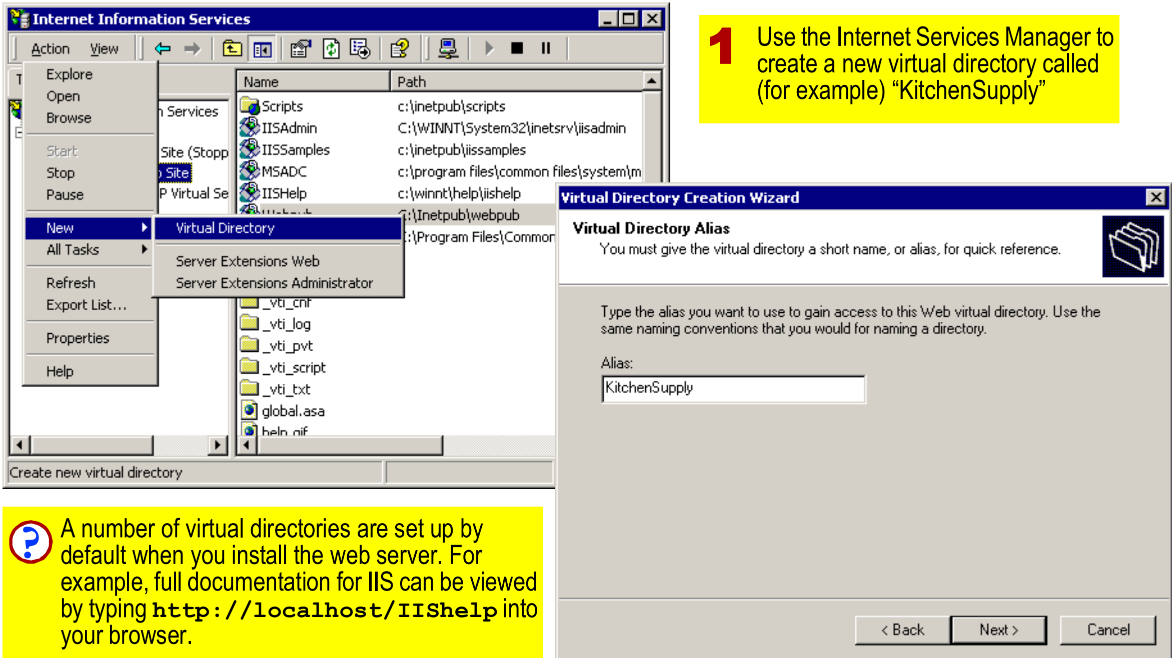


If you are running an older version of WINDOWS (pre 98), you may want to search on the MICROSOFT site for and upgraded version of IIS.

3. Find the web server's administration tool. It is normally named "Internet Services Manager" or something similar. The tool will allow you to set up virtual directories, set permissions on the directories, start the web service, and so on. Figure 24.10 and Figure 24.11 show the procedure for creating a virtual directory called `http://localhost/KitchenSupply` in a physical

directory called `C:\Documents and Settings\My Documents\KitchenSupply`.

FIGURE 24.10: Create a virtual directory on the web server (Part 1).

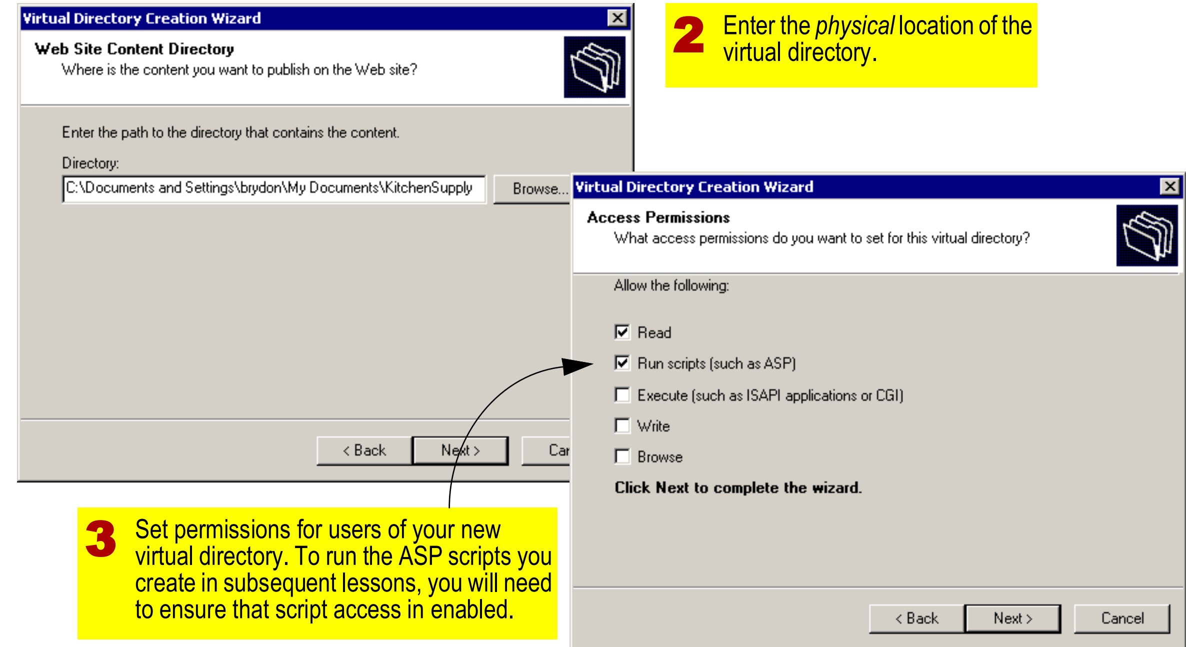


A number of virtual directories are set up by default when you install the web server. For example, full documentation for IIS can be viewed by typing **`http://localhost/IIShelp`** into your browser.

You can use any directory names you like; these are simply examples.

4. Put your content in a physical web server directory and create a virtual directory to access the files.

FIGURE 24.11: Create a virtual directory on the web server (Part 2).



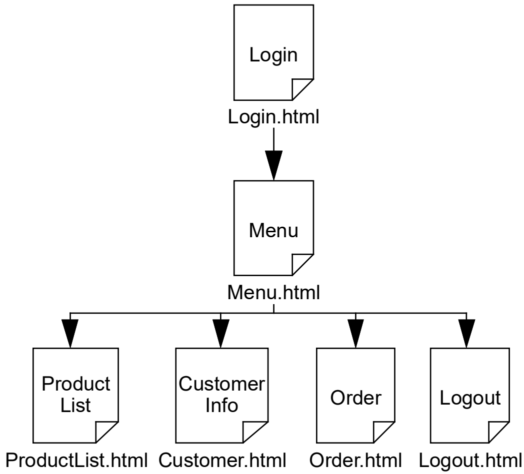
24.5 Application to the project

24.5.1 Application structure

For the simple web-based order entry system you will create in subsequent lessons, you will need to start with six pages with HTML content. The structure of the application and HTML pages is shown in Figure 24.12.

Since only authorized users are permitted to place orders, the point of entry is a login page. The login page requires the user to enter a

FIGURE 24.12: The structure of the web-based order entry system.



username and password. If the username/password pair are valid, then the user is transferred to a main menu. From the menu, the user can

- view a list of products,
- update her customer information (e.g., contact person, shipping address, and so on),
- place or view an order, and
- logout.

At this early stage, you should concentrate on creating the basic documents. Dynamic content, hyperlinks, and form elements will come later.

23 Create the HTML documents shown in Figure 24.12. Each document should have a header and body section.

24 Add meaningful titles (using the `<TITLE>` tag) to the header sections of your documents.

25 Use the heading tags (e.g., `<H1>`) to add visible headings to all your pages.

It is possible to spend an enormous amount of time formatting your pages. At this point, you are *not* encouraged to invest much effort in making your pages look good by adding additional tags, backgrounds, images, and so on.

24.5.2 Local web server

26 Review Section 24.4.2 to determine whether there is a web server installed and configured on your machine.

27 If no web server is installed and it is practical to install one, consult the WINDOWS documentation for instructions on how to install a scaled down version of IIS.

If a web server has not been set up for you, you will need to install a local server before you can continue with Lesson 25.

Lesson 25: HTML forms

25.1 Introduction

To execute a meaningful business transaction on the Internet, the client side (that is, the user's browser) has to be able to send information to the web server. This lesson provides a brief overview of different ways in which browsers can communicate with web servers.

25.1.1 Web 101

The Internet is just a **packet-switched** network that is capable of carrying all types of traffic including mail, chat, and proprietary data. The hypertextual, multimedia infrastructure we know as the World-Wide Web (WWW) is just one type of Internet traffic.

What gives the WWW its power is two high-level standards developed by Tim Berners-Lee in the early 1990s while working at CERN: HTML and HTTP. The HTML standard, which you saw in Lesson 24, defines how documents are displayed within web browsers. The **hypertext transfer protocol** (HTTP) defines how the web server and the web browser communicate over the Internet in the first place. You can think of these protocols as being layered: HTML is "carried by" HTTP.[1]

25.1.2 HTTP requests and responses

Whenever you click on a hyperlink on a web page, your browser creates an HTTP **request** and sends it out on the network. An HTTP request is simply text message that is routed to a particular web server. For example, consider the following HTTP request:

NL
```
GET mis.bus.sfu.ca/Default.htm
HTTP/1.0
```

This `GET` messages asks a particular web server (`mis.bus.sfu.ca`) to send back a particular document (`Default.htm`). It also specifies the version of HTTP to avoid possible confusion as the protocol evolves.

Upon receipt of the request, the web server obliges by sending back an HTTP **response** containing the HTML contents of the requested page (in this example, `Default.htm`).

[1] Other lower-level protocols—such as the Transmission Control Protocol (TCP) and Internet Protocol (IP)—"carry" HTTP. However, to understand the basics, we can assume all the nitty-gritty network stuff is given and focus on the so-called "application layers".

25.1.3 Sending additional data

Of course, a simple **GET** request is insufficient if you want to tell the web server who you are or that you wish to order product number “51 5012”. Fortunately, the HTTP standard defines two basic mechanisms for passing additional data to the web server in the request: **query strings** and **form fields**. In this lesson, you will learn about both and get some experience creating simple HTML forms.

25.2 Learning objectives

- understand what an HTTP request is and how it is used to retrieve content from web servers
- pass information to the server using GET requests and query strings
- pass information to the server using POST requests and form fields
- learn how to create simple forms in HTML
- create more advanced form elements like radio buttons, check boxes and drop-down lists

25.3 Exercises

In the following exercises, you will send a number of HTTP requests to a special “Echo Request” utility. Echo Request is a small program on a web server that extracts and displays (“echos”) the information sent to it by clients.

The Echo Request document used in this lesson is located on a web server at Simon Fraser University (`e-commerce.bus.sfu.ca`). If you have your own web server software installed (see Section 24.4.2), you can copy the `EchoResquest.asp` file from the project package to a virtual directory on your local web server. You then use the URL of the local copy (`localhost/` *`<virtual directory>`*`/EchoRequest.asp`) in the place of `e-commerce.bus.sfu.ca`.

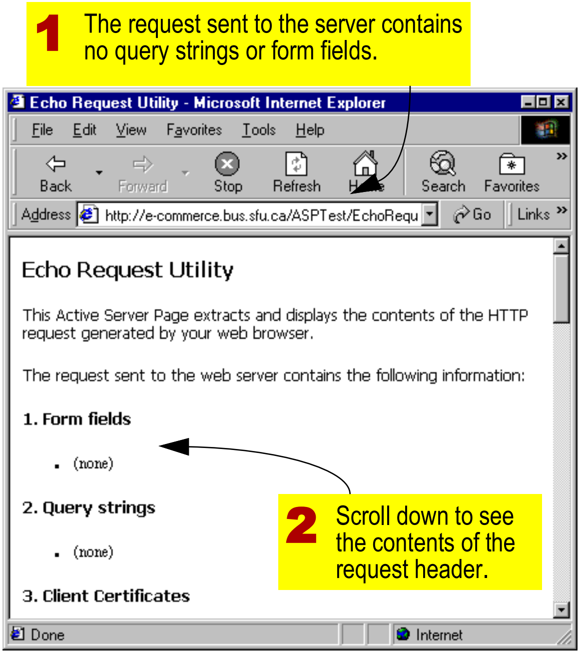
1 Send an “empty” HTTP **GET** command to the Echo Request utility:

NL `http://e-commerce.bus.sfu.ca/ASPTest/EchoRequest.asp`

2 Examine the information returned by the Echo Request utility, as shown in Figure 25.1.

In addition to returning information about basic collections (form fields and query strings), the Echo Request utility returns information about the request’s advanced collections, such as **Client Certificate** (for

FIGURE 25.1: Result of sending a simple *GET* command to the Echo Request utility.



robust authentication), **Cookies**, and **Server Variables**. You can ignore this information for now. In this lesson, we are only interested in the Form and Query String collections.

25.3.1 Passing data using query strings

A query string is a quick and easy means of passing information to the web server. The information is passed to the server by appending one or more query/value pairs to the URL in the **GET** request.

3 Enter the following URL and query string combination into your browser. The result is shown in Figure 25.2.

NL
```
http://e-commerce.bus.sfu.ca/
ASPTest/EchoRequest.asp?
UserName=sammy&Password=sam
```

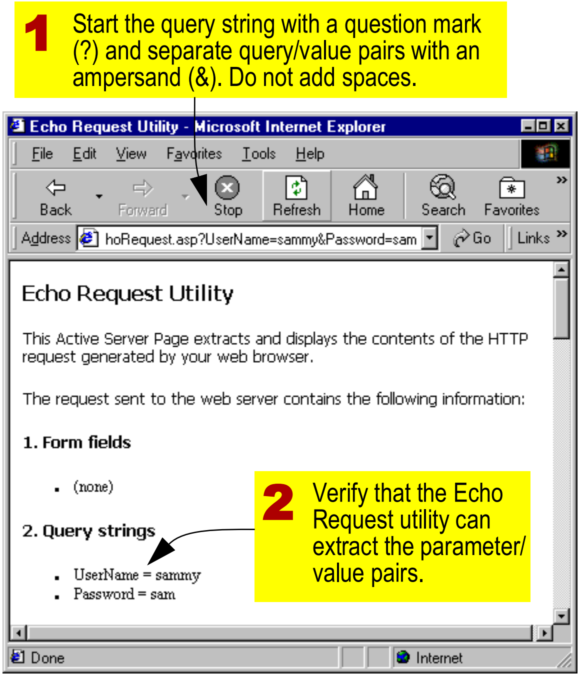
A query string consists of one or more query/value pairs. In the example above, you send two such pairs to the server:

- UserName = “sammy”
- Password = “sam”

Note that the start of the query string is signified by a question mark (`?`). When multiple parameter/value pairs are passed, an ampersand (`&`) is used to separate the pairs.

The name “query string” is unfortunate because passing data in this manner has little in common with the database concept of “queries”. A better name might be “parameter string”. However—as is often the case in computing—we are stuck with the legacy name.

FIGURE 25.2: Send a query string with two query/value pairs to the server.



Given the less-than-intuitive syntax of query strings, it is clear that they cannot be used for collecting information from users. For example, you would not expect users to append their user name and password to a URL in order to log on to a secure system. In addition, the query string shows in the browser's address field. Because of this, query strings should not be used for information that users (or people standing behind users) do not need to see.

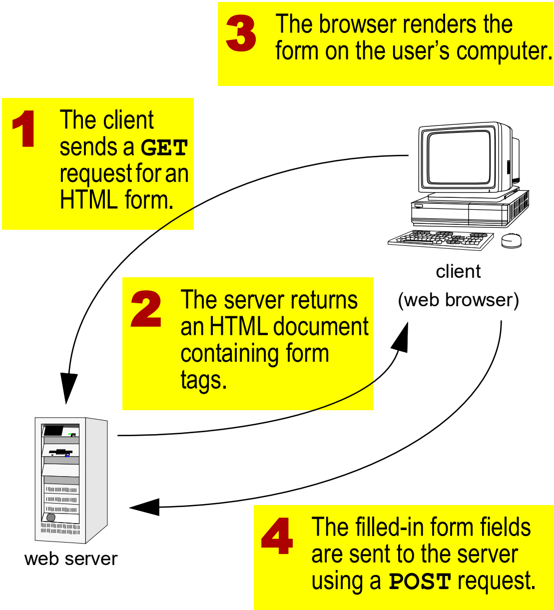
As it turns out, query strings are most often used by developers to pass status information to the web server in order to get around the "statelessness" of the HTTP protocol. You will see how this works in Lesson 26.

25.3.2 Passing data using forms

Fortunately, the HTTP and HTML standards provide mechanisms for displaying forms on the client browser and transferring the information entered by the user back to the web server. The key elements of the HTTP/HTML forms infrastructure are shown in Figure 25.3.

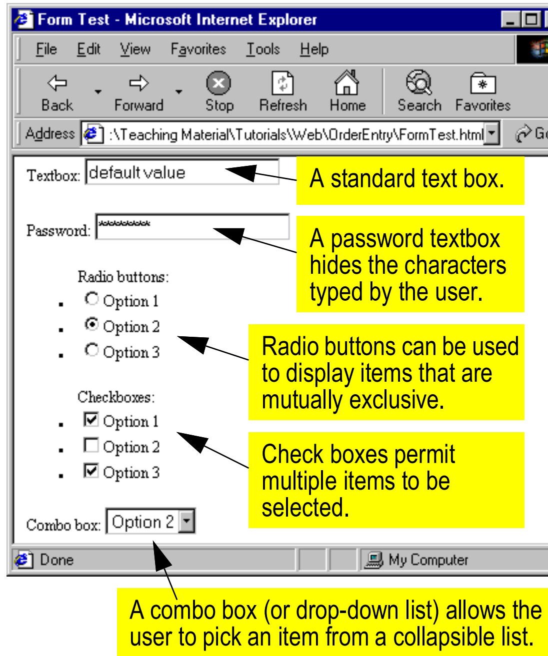
At this point, nothing has been said about what the web server *does* with the information that it receives from the user. Generally, this information is processed by a program running on the server. However, getting information from the web server to a server-side processing routine and back to the web server is not trivial. Indeed, it may involve many acronyms (e.g., CGI, ASP, CFML, PHP, JSP, ISAPI, ADO, COM, etc.). We will deal with some processing issues in later lessons.

FIGURE 25.3: Key elements of the HTML/ HTTP form infrastructure.



An HTML form is like any other HTML document that resides on the web server. However, when the browser receives the form, it recognizes special HTML tags and renders the form elements on the user's screen. Figure 25.4 shows some of the form elements that can be defined using HTML.

FIGURE 25.4: HTML form elements rendered by a web browser.



To define a form, you must provide the following:

- the type of HTTP request that is to be created by the browser and sent to the web server (the **METHOD** attribute),
- the destination of the HTTP request (the **ACTION** attribute), and
- a **SUBMIT** button to send the HTTP request.

In HTML, these elements can be adding using the following tags:

```
NL  <FORM METHOD="POST"
    ACTION=target URL>
NL    form elements ...
NL    <INPUT TYPE="submit" VALUE=label>
NL  </FORM>
```

There are two possible values for the **METHOD** attribute: **GET** and **POST**. All you need to know at this point is that **POST** is almost always used with forms.

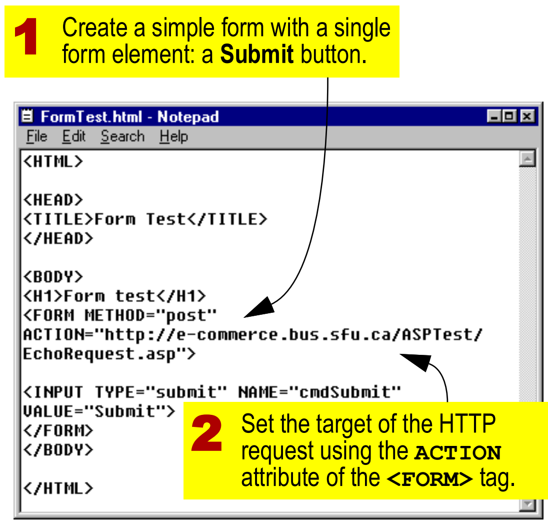
4 Create a new HTML document with header and body sections. Save it as **FormTest.html**.

5 In the body section, enter the following:

```
NL  <BODY>
NL  <H1>Form test</H1>
NL  <FORM METHOD="post" ACTION="http://
    e-commerce.bus.sfu.ca/ASPTest/
    EchoRequest.asp">
NL  <INPUT TYPE=submit NAME="cmdSubmit"
    VALUE="Submit">
NL  </FORM>
NL  </BODY>
```

6 Save the document. The HTML code is shown in Figure 25.5.

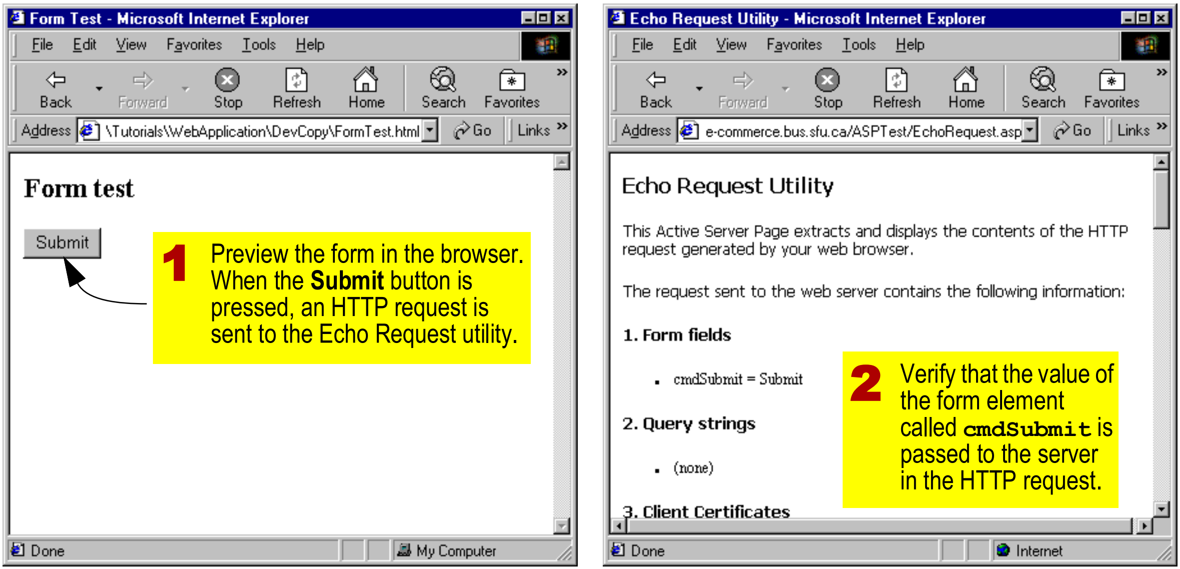
FIGURE 25.5: Create a basic HTML form consisting of a **Submit** button.



Remember: whitespace is ignored in HTML and thus the wrapping of your lines within the editor is irrelevant.

7 Test the form using a web browser. When you press the **Submit** button, the Echo Response utility will show you the contents of the HTTP request, as shown in Figure 25.6.

FIGURE 25.6: Create a basic HTML form and test it by sending the HTTP response to the Echo Response utility.



25.3.3 Basic form elements

A form that simply sends the value of the **Submit** button is not particularly useful. Elements that are typically included on forms include the textbox and its cousin, the password textbox.

8 Add the following to your HTML document (after the **<FORM>** tag but before the **Submit** button):

```
NL <BODY>
NL <H1>Form test</H1>
NL <FORM METHOD="post" ACTION="http://
   e-commerce.bus.sfu.ca/ASPTest/
   EchoRequest.asp">
NL <P>Textbox: <INPUT TYPE="text"
   NAME="txtItem" VALUE="default
   value"></P>
NL <INPUT TYPE=submit NAME="cmdSubmit"
   VALUE="Submit">
NL </FORM>
NL </BODY>
```

Make sure you remember the quotation marks around the attribute values. Otherwise, browsers will misinterpret "default value".

9 Under the textbox you added above, add a password textbox:

```
NL <P>Password: <INPUT TYPE="password"
   NAME="txtPassword"></P>
```

The form elements above are nested inside of paragraph tags to space things out a bit; however, you may choose to format your page however you like. In general, it is very difficult to get form elements to line up nicely. As such, web designers routinely place form elements inside of invisible table cells or use other formatting tricks. At this stage, you should not worry about how your forms look.

10 Save the file and refresh the form in your browser.

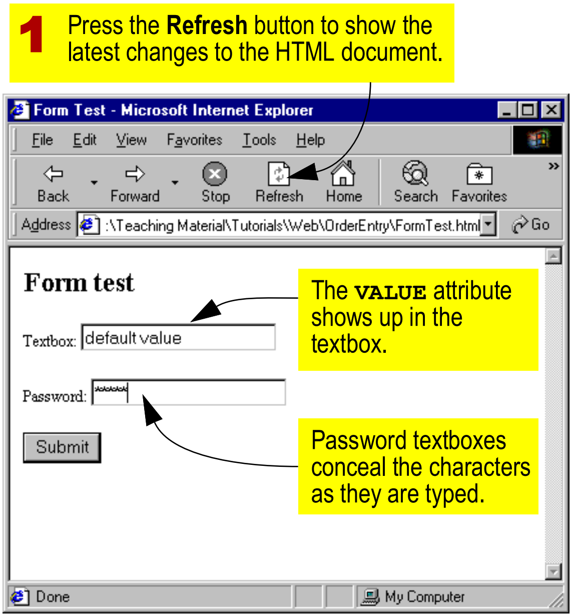
11 Press the **Submit** button and verify that the values you typed were sent to the web server. This is shown in Figure 25.7.

Although the value in the password textbox is concealed by the web browser, this feature is merely intended to hide the value from other people who may be able to see the web browser. There is no protection or encryption once the **Submit** button is pressed—the password travels across the network as plain text. See Section 25.4 for more information on security and encryption.

25.3.4 Other form elements

All the form elements created so far use the **<INPUT>** tag with different values for the **TYPE**

FIGURE 25.7: Add a textbox and password textbox to the form.



attribute. The possible types of form elements are summarized below.

TABLE 25.1: Form elements in HTML.

TABLE 25.1: Form elements in HTML.

TYPE value	**Usage**
text	simple textboxes for names, etc.
password	concealed text for confidential information such as passwords
submit	a button required to trigger the creation of an HTTP request and send it to a web server
reset	also rendered as a button; resets all form elements to the values specified in their **VALUE** attributes (it any)
radio	creates a group of radio buttons of which *only one* can be selected (mutually exclusive options)
checkbox	creates one or more checkboxes of which *zero or more* can be checked (non-mutually exclusive options)
button	creates a button the form that is not assigned any behavior by default
hidden	the element's value is transferred to the server, but no field is shown on the form; can be used instead of query strings to transfer status information

The other attributes of the **<INPUT>** tag, **NAME** and **VALUE**, can take on any values specified by the creator of the form.

As in all programming endeavors, it is good policy to adopt a meaningful and consistent naming policy for HTML elements.

12 Add a group of radio buttons to your form:

```
NL  …
NL  <BODY>
NL  …
NL  <P>Password: <INPUT TYPE="password"
    NAME="txtPassword"></P>
NL  <UL>Radio buttons:
NL  <LI><INPUT TYPE="radio"
    NAME="rdoGroup" VALUE="opt1">Option
    1</LI>
NL  <LI><INPUT TYPE="radio"
    NAME="rdoGroup" VALUE="opt2">Option
    2</LI>
NL  <LI><INPUT TYPE="radio"
    NAME="rdoGroup" VALUE="opt3">Option
    3</LI>
NL  </UL>
NL  <INPUT TYPE=submit NAME="cmdSubmit"
    VALUE="Submit">
NL  </FORM>
NL  </BODY>
NL  …
```

<UL> is a standard HTML tag used to define an "unordered list" (an "ordered list" uses the **<OL>** tag and has numbers instead of bullets). Each "list item" in the list is defined using the **<LI>**…**</LI>** tags. Of course, it is not necessary to nest radio buttons within a list—you may format them any way you wish.

13 Save the document and test the radio buttons by selecting one of the options.

14 Send the HTTP request to the server and observe the result.

Note that all the radio buttons are assigned the same name yet only one value is assigned to the **rdoGroup** field. A radio button will always return a single value since the browser prevents the user from selecting more than one option. The value of **rdoGroup** is simply the **VALUE** attribute of the radio button that is selected.

Checkboxes are similar to radio buttons, except that the former permit multiple values to be assigned to a single form field. Checkboxes are often used for on-line customer surveys. If respondents are asked, "which product features are important?' and more that one answer is possible, multiple checkboxes can capture a multivalued response, e.g., **chkFeatures** = {price, quality, service, reputation}

15 Add a group of checkboxes to your form:

```
NL  …
NL  <BODY>
NL  …
```

```
NL  <LI><INPUT TYPE="radio"
    NAME="rdoGroup" VALUE="opt3">Option
    3</LI>
NL  </UL>
NL  <UL>Checkboxes:
NL  <LI><INPUT TYPE="checkbox"
    NAME="chkGroup" VALUE="opt1">Option
    1</LI>
NL  <LI><INPUT TYPE="checkbox"
    NAME="chkGroup" VALUE="opt2">Option
    2</LI>
NL  <LI><INPUT TYPE="checkbox"
    NAME="chkGroup" VALUE="opt3">Option
    3</LI>
NL  </UL>
NL  <INPUT TYPE=submit NAME="cmdSubmit"
    VALUE="Submit">
NL  </FORM>
NL  </BODY>
NL  …
```

16 Save the document and test the checkboxes by selecting more than one of the options. The results are shown in Figure 25.8.

25.3.5 Combo boxes

It is possible to create combo boxes similar to those created in ACCESS in Lesson 15.

"Combo box" is a MICROSOFT term but it is used here for consistency with the ACCESS material. In the Internet world, such form elements are typically called "drop-down lists" or "list boxes".

The combo box itself is defined by the **<SELECT>** tag. The list that shows when the combo box is activated is defined by one or more **<OPTION>**… **</OPTION>** tags:

- the **VALUE** attribute within the option tag determines the value that is returned if the option is selected by the user;
- the text within the opening and closing tags determines what shows in the combo box.

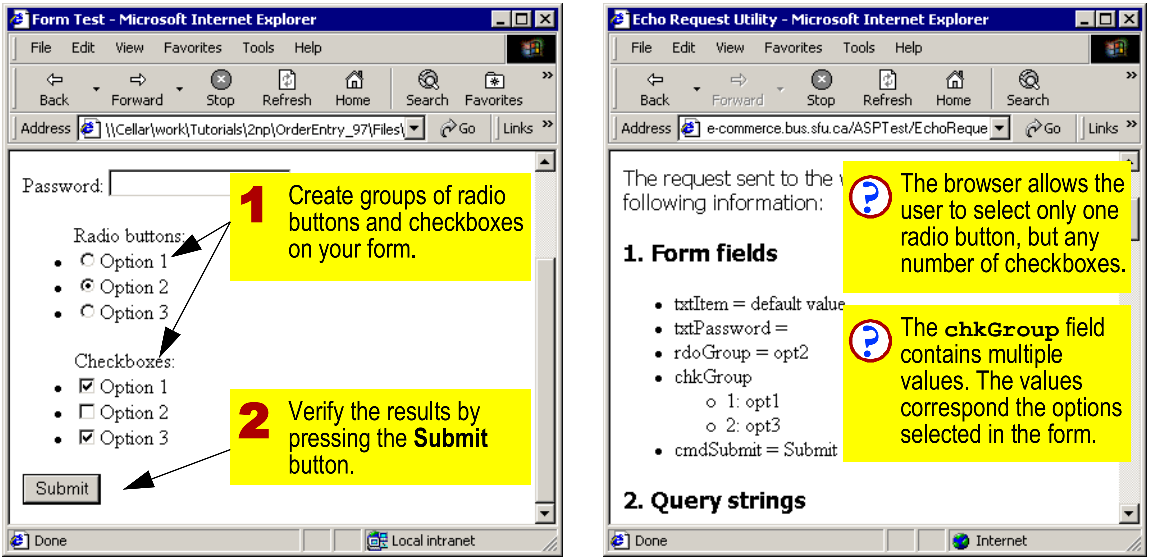
For example, the following tag adds the value "Option 1" to the list of items in a combo box called "cboItems":

```
NL  <SELECT NAME="cboItems">
NL  <OPTION VALUE="opt1">Option 1</
    OPTION>
NL  … other list items…
NL  </SELECT>
```

If the user selects the value "Option 1" from the combo box, then the value of **cboItems** is set to "opt1". In other words, HTML provides a means of hiding the key in the **<OPTION>** tag's **VALUE** attribute while showing the user a more meaningful value in the visible list (recall using ACCESS to do the same thing in Section 15.3.2.4).

17 Add a combo box to your form:

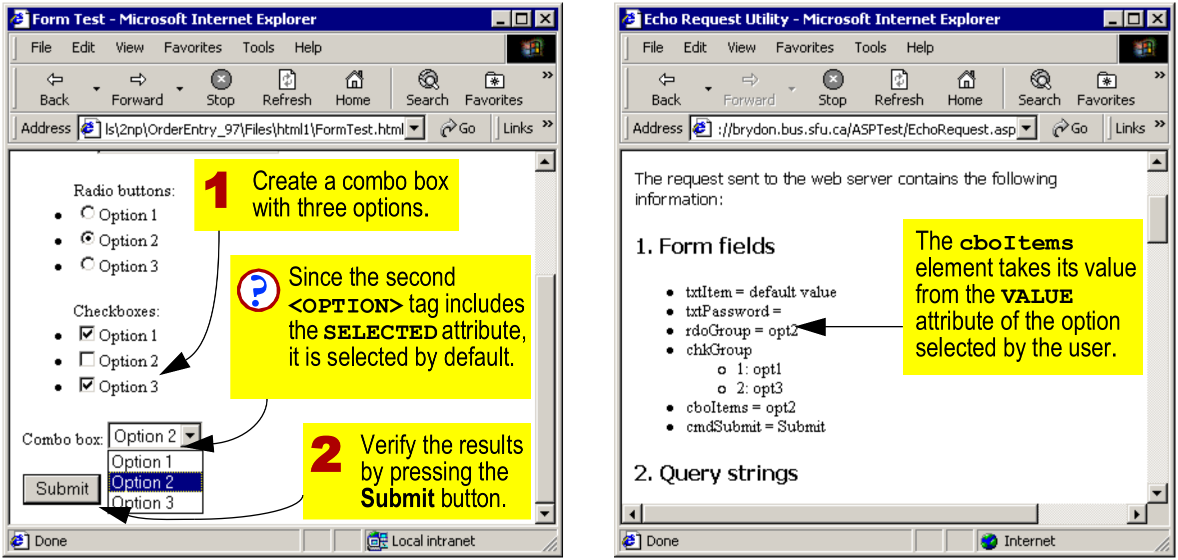
FIGURE 25.8: Create radio button and checkbox groups on the form.




```
NL  <BODY>
NL  …
NL  <LI><INPUT TYPE="checkbox"
    NAME="chkGroup" VALUE="opt3">Option
    3</LI>
NL  </UL>
NL  <P>Combo box:
NL  <SELECT NAME="cboItems">
NL  <OPTION VALUE="opt1">Option 1</
    OPTION>
NL  <OPTION VALUE="opt2"
    SELECTED>Option 2</OPTION>
NL  <OPTION VALUE="opt3">Option 3</
    OPTION>
NL  </SELECT></P>
NL  <INPUT TYPE="submit"
    NAME="cmdSubmit" VALUE="Submit">
NL  </FORM>
NL  </BODY>
```

18 Save the document and test the combo box. The result is shown in Figure 25.9.

FIGURE 25.9: Create a combo box (dropdown list) on the form.



25.3.6 Menu forms

Customers will use your application's main menu to select the pages they wish to view. For example, they may wish to update their customer profile and then logoff. Alternatively, they may need to view an existing order and then create a new order. A simple menu for navigating from page to page (e.g., from `Customer.html` to `Logoff.html`) is easy to create using hyperlinks (recall Section 24.3.3). However, if the content of the target page is variable (e.g., one may jump to a new order or a particular order created in the past), then a form-based menu provides greater flexibility.

25.3.6.1 Multiple submit buttons

19 Open `Menu.html` for editing and add form definition tags to the body section. At this point, set the `ACTION` attribute to the Echo Request utility.

20 Add a menu item and a separate **Submit** button for each of the choices available to the user. An invisible two-column table can be used to simplify the positioning of the menu items and buttons.

```
NL  <HTML>
NL  <HEAD>… </HEAD>
NL  <BODY>
NL  …
NL  <FORM METHOD="POST" ACTION="http://
    e-commerce.bus.sfu.ca/
    ASPTest/EchoRequest.asp">
NL  <TABLE>
NL  <TR>
NL  <TD>Update Customer Profile</TD>
NL  <TD><INPUT TYPE="Submit"
    NAME="cmdCustomer" VALUE="Go"></TD>
NL  </TR>
NL  <TR>
NL  <TD>View Product List</TD>
NL  <TD><INPUT TYPE="Submit"
    NAME="cmdProduct" VALUE="Go"></TD>
NL  </TR>
NL  <TR>
NL  <TD>Add or View Orders</TD>
NL  <TD><INPUT TYPE="Submit"
    NAME="cmdOrder" VALUE="Go"></TD>
NL  </TR>
NL  …
NL  </TABLE>
NL  </FORM>
NL  </BODY>
NL  <HTML>
```

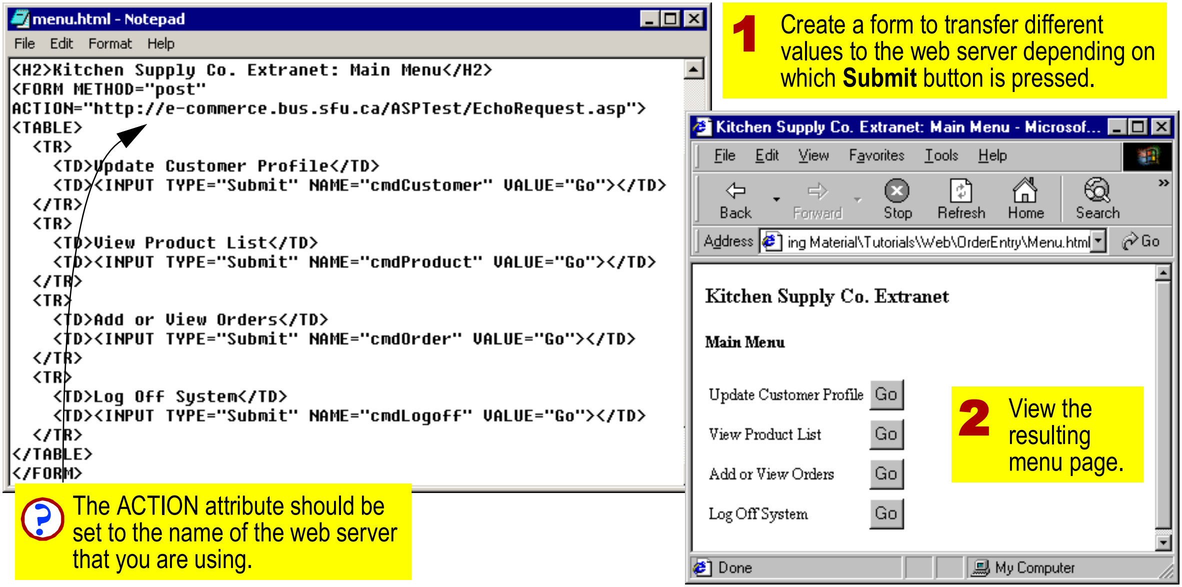
Use a different descriptive name for each button (e.g., `cmdCustomer`, `cmdProduct`, `cmdOrder`, `cmdLogoff`).

21 Save the file and view it in the browser, as shown in Figure 25.10.

22 Test the menu by clicking on different **Go** buttons. As the Echo Request utility shows in Figure 25.11, the name/value pair for the button that is pressed—and *only* the button that is pressed—is transferred to the server.

In a subsequent lesson, you will write a server-side script to determine which button was pressed and transfer the user to the appropriate page. For now, creating the menu items and buttons is sufficient.

FIGURE 25.10: Create a form-based menu for the application.

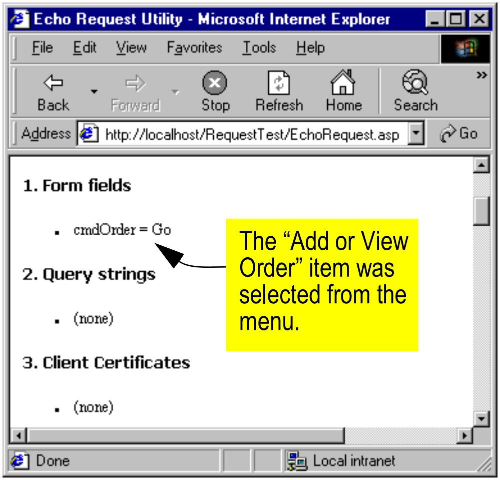


25.3.6.2 Menu items and combo boxes

The menu in Figure 25.10 is incomplete because the user has no means of indicating whether she wants to create a new order or view an existing order (i.e., an order that has already been processed or one that has been started by not yet finalized). If the user wants to view an existing order, then she must have some way of telling the system *which* order (remember, a customer may have placed many different orders in the past).

To correct this shortcoming, you can add a combo box that lists all the existing orders (using some compact naming convention, such as order date) plus an option for creating an entirely new order.

FIGURE 25.11: The name/value pair of the selected item is transferred to the server.



23 In the "Add or View Orders" table cell, add the following code to create a combo box:

```
NL  <FORM METHOD="POST" ACTION="http://
    e-commerce.bus.sfu.ca/ASPTest/
    EchoRequest.asp">
NL  <TABLE>
NL  …
NL  <TR>
NL  <TD>Add or View Orders
NL  <SELECT NAME="cboOrderID">
NL  <OPTION VALUE=0>(new order)</
    OPTION>
NL  <OPTION VALUE=1>15 May</OPTION>
NL  <OPTION VALUE=2>21 May</OPTION>
NL  </SELECT>
NL  </TD>
NL  <TD><INPUT TYPE="Submit"
    NAME="cmdOrder" VALUE="Go"></TD>
NL  </TR>
NL  …
NL  </TABLE>
NL  </FORM>
```

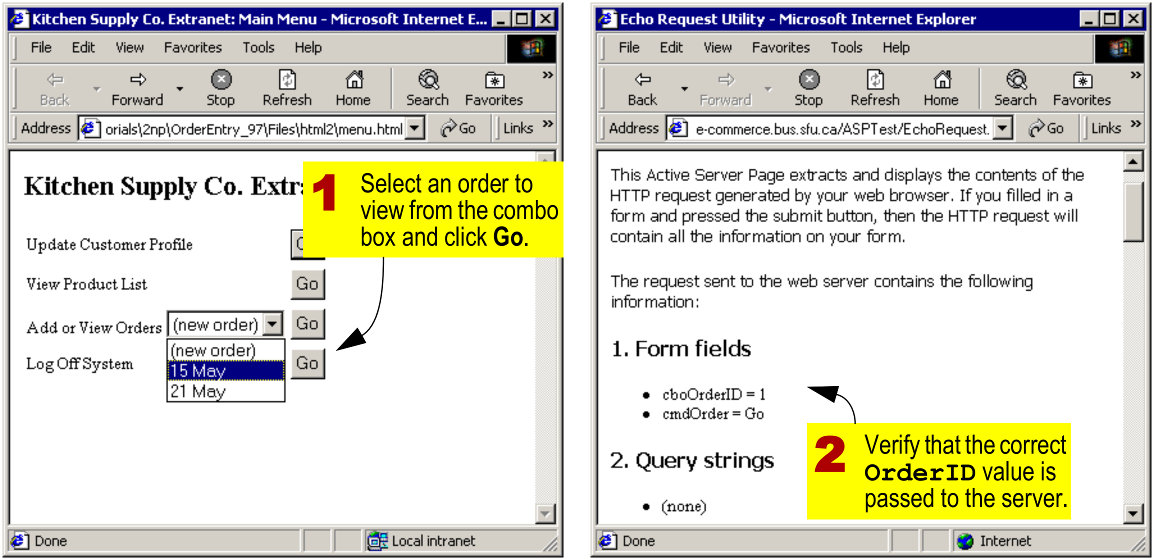
I have added the combo box to the same table cell (that is, within the same **<TD>…</TD>** tags) as the prompt "Add or View Orders". For this to work, you must remember to delete the closing **</TD>** tag at the end of the prompt and add a new one at the end of the **</SELECT>** tag. Of course, you can format your menu page anyway you like.

This combo box above assumes that two orders (with `OrderID`s of 1 and 2) have already been entered into the system. Since it is not possible to have an `OrderID` of zero, the 0 option value is used to indicate a new order.

24 Test the menu by selecting a value from the combo box and clicking the appropriate

Go button. One possible outcome is shown in Figure 25.12.

FIGURE 25.12: Add a combo box to the menu to specify which order to view or update.



By using a form to create the main menu, you are able to pass the server more information than if a simply hyperlink were used. In Figure 25.12 for instance, the server is sent two pieces of information:

1. The user wishes to view an order (`cmdOrder`=Go).
2. The order that the user wishes to view has `OrderID`=2.

25.4 Discussion

25.4.1 Security concerns

In Figure 25.7, you sent a password over the public Internet to a web server. In travelling to the web server, the IP data packets carrying the HTTP request could have been routed through many different sub-networks and computers. Since HTTP is a plain-text protocol, all field/value pairs are easily read by so-called "packet sniffers". A packet sniffer is a program that can extract data from the stream of IP packets travelling along a network.

Although reading all the network traffic passing through a router or server would involve a fair bit of sniffing, it is fairly straightforward to create a program that watches for field names such as "`txtCreditCardNo`" or the characteristic sequences of digits that prefix credit card numbers.

25.4.2 Encryption

To get around the problem, most confidential information is now encrypted by the browser, sent over the network, and decrypted by the target web server. A packet sniffer looking at the contents of the HTTP request would only see a jumble of characters ("cipher text") in the place of well-defined field/value pairs.

There are different standards for encryption and authentication, but the most common on the Internet at this time is the secure sockets layer (SSL). Although it is possible—in principle—to decrypt streams of encrypted data, it makes little economic sense in practice to even try.

25.5 Application to the project

25 Add a form to your login page and set its **`ACTION`** attribute to point to the Echo Request utility (this will be changed later).

26 Add a textbox called **`txtUserName`**.

! In Section 25.3.3, you created a textbox with the **`VALUE`** attribute preset to "default value". The user could replace the initial value by typing something into the form field. It is important to note, however, that providing an initial value to the **`VALUE`** attribute is optional. Indeed, in the context of a textbox for a user name, use of a default value makes little sense (unless you can somehow guess who the user is before she visits your site).

27 Add a password textbox called **`txtPassword`**.

28 Add a submit button called `cmdSubmit`. Its `VALUE` attribute should be set to “Submit”.

Lesson 26: Server-side scripting

26.1 Introduction: creating content dynamically

As you discovered in Section 24.3.5, maintenance of a product list using HTML is a labor-intensive, mind-numbing experience—every change to price or inventory information involves searching through the HTML table cells, making the changes manually, and publishing the updated file to the web server.

Although it is possible to use the **Save as HTML** within ACCESS to automatically generate an HTML table containing product information, the resulting document is static. If a price is changed or a new product is added, the "save as" procedure must be repeated.

A better approach is to have the server create the product list dynamically, the moment it is requested. In this way, the most recent database information is used.

26.1.1 Scripting basics

Although there are a number of different ways to create documents dynamically on the server using scripts (small programs), the process typically involves the following steps:

1. The user clicks on a link to a document. The user's browser generates an HTTP request and sends it to the web server in the normal way (recall Lesson 25).
2. The server receives the request and retrieves the document file requested by the browser. If the file has a non-HTML extension, it is pre-processed before being transferred to the browser.
3. During preprocessing, the server checks for special non-HTML tags and programming code (e.g., VBSCRIPT, COLDFUSION, JAVA).
4. If special tags are found, the server processes them. For example, if the tags are VBSCRIPT, the VISUAL BASIC code they contain is executed by the ACTIVE SERVER PAGES (ASP) processor. If the tags contain COLDFUSION MARKUP LANGUAGE (CFML) queries, the SQL statements are executed by the COLDFUSION processor.
5. In most cases, the code within the special tags returns a result, such as a calculation or a database lookup. The server replaces the special tags with the result before the document is sent to the client.

Since all the work is done on the server, this process is called “server-side scripting”. After all the server-side processing is complete, the document sent to the browser is plain HTML. Users have no way of knowing that the document showing in their browser contains content that was generated dynamically in the instant it took the server to respond to the request.[1]

26.1.2 A simple example

Consider the following VBSCRIPT embedded within standard HTML tags:

```
NL  <HTML>
NL  <HEAD>…</HEAD>
NL  <BODY>
NL  <P>The sum of 3 and 5 is:
    <%= 3+5 %></P>
NL  </BODY>
NL  </HTML>
```

The the code within the special script tags **<%… %>** is executed by the ASP processor and replaced by the result. Thus, the HTML that the browser receives is simply:

```
NL  <HTML>
NL  <HEAD>…</HEAD>
NL  <BODY>
NL  <P>The sum of 3 and 5 is: 8</P>
NL  </BODY>
NL  </HTML>
```

The ability to mix executable code within the structure and format of an HTML document facilitates a straightforward division of labor: The look and feel of the page can be created in plain HTML by artists and web design specialist. Then, programmers and database specialists can insert scripting tags within the HTML to enable dynamic generation of content.

26.1.3 The preprocessor

In this lesson, you will use the ACTIVE SERVER PAGES (ASP) functionality that is bundled with MICROSOFT’s INTERNET INFORMATION SERVER (IIS) to build some simple server-side scripts.[2]

To complete these exercise, you must be able to place ASP files into an IIS web server directory and access the directory with a web browser. An ASP-capable “personal web server” is included with MICROSOFT WINDOWS and is useful for

---

[1] “Instant” is a relative term. When the amount of processing done by the server is considerable (e.g., searching for a cheap flight on EXPEDIA.COM), the delay in returning a page to the user can also be considerable.

[2] Do not confuse ACTIVE SERVER PAGES with “Application Service Provider”—another popular Internet term that uses the same acronym.

building and testing web sites when you do not have a persistent Internet connection to a server.

Although ASP supports the use of a number of scripting languages, we are going to focus on VISUAL BASIC SCRIPTING EDITION (known as VBSCRIPT). VBSCRIPT is a simplified subset of VISUAL BASIC similar to the VBA language used in Lesson 18.

26.2 Learning objectives

- create dynamic web pages using MICROSOFT'S ACTIVE SERVER PAGES technology
- understand the essentials of server-side scripting
- use VBSCRIPT within ASP pages
- understand the Response object and how is it used to send information to browsers
- understand the Request object and how is it used to receive information from browsers
- understand how server-side scripting differs from CGI programming

26.3 Exercises

An ASP file is a plain-text file, virtually identical to an HTML file. There are two important differences, however:

1. ASP files end with an .asp extension.
2. ASP files can contain scripting tags in addition to plain text and HTML tags.

26.3.1 A simple example

In this section, you will create a simple ASP file and review some basic programming constructs such as looping and branching.

1 Use a text editor to create a new file. Add the basic HTML elements to the document (header, body, title)

To save time, you might want to create a file called `Skeleton.html` that contains the core HTML tags. You can cut and paste from this file whenever you need to create a new document.

2 In the body, enter the following ASP and HTML code:

```
NL  <HTML>
NL  <HEAD>…</HEAD>
NL  <BODY>
NL  <P>The sum of 3 and 5 is:
    <%= 3+5 %></P>
```

```
NL  </BODY>
NL  </HTML>
```

3 Save the file as **ASPTest.asp**.

Remember to use the asp extension instead of html for files containing scripting language commands. If the asp extension is not used, the web server will not preprocess the ASP tags and VBSCRIPT code will simply be transferred to the browser as plain text.

Recall that in Lesson 24, you could preview HTML-only documents by opening the files without going through a web server: you simply used **File** → **Open** in your browser or double-clicked on the file. In contrast, ASP files *must* be accessed through the server in order for the VBSCRIPT code to be executed. Hence the name “server-side scripting”.

4 Copy your local copy of **ASPTest.asp** to the web server you are using for the remainder of the project.

From this point forward, the “publish” step will be implied. Ever time you make a change to your local copy of a file, you must publish the change to the web server. Of course, if you are running a local web server, you can omit the publishing step entirely.

5 Type the URL of the **ASPTest.asp** document into the address bar of your browser and verify that the script executed property. This is shown in Figure 26.1.

The URL should be of the form: **http://*<server name>/<directory>*/ASPTest.asp**.

26.3.2 Using the Response object

Programming in the ASP environment requires an understanding of ASP’s built-in objects. In this section, you will take a brief look at the **Response** object.

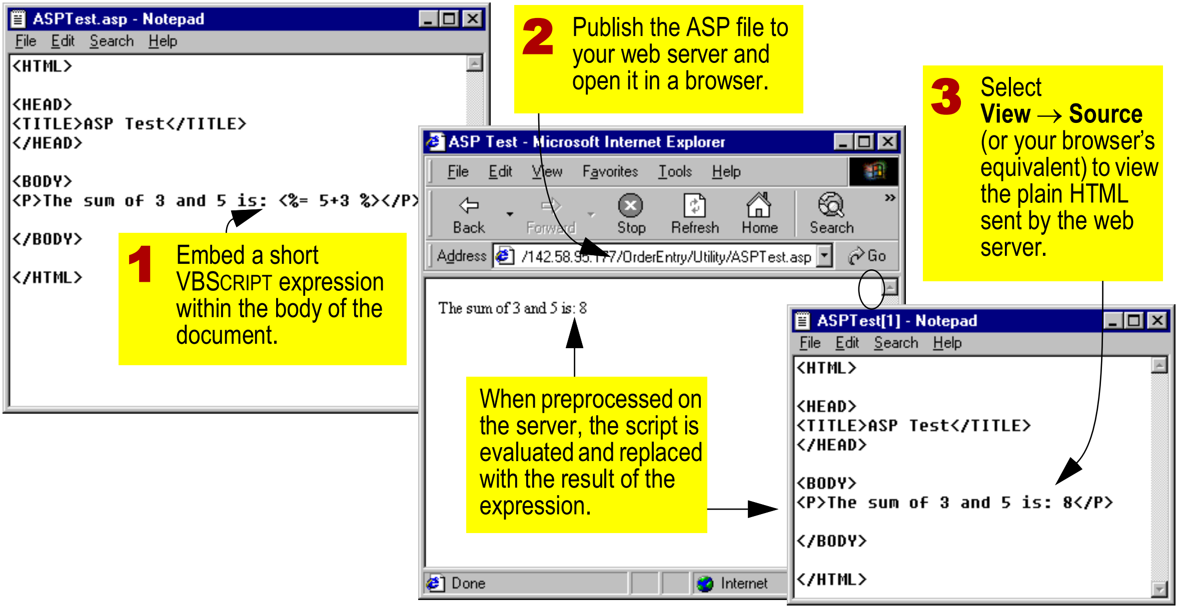
26.3.2.1 The Write method

6 Edit the script in **ASPTest.asp** to read:

```
NL  <HTML>
NL  <HEAD>…</HEAD>
NL  <BODY>
NL  <P>The sum of 3 and 5 is:
    <% Response.Write(3+5)%></P>
NL  </BODY>
NL  </HTML>
```

Ensure that you *do not* include the equals sign after the opening **<%** tag.

FIGURE 26.1: Use NOTEPAD to create an ASP document containing a very short VBSCRIPT expression.



The `Response` object provides a means of controlling the web server's HTTP response to the client. For example, the `Write()` method writes a string of text directly to the browser.

The equals sign you used in Section 26.3.1 is shorthand for the `Write()` method. In other words, `<%= "Hello" %>` and `<% Response.Write("Hello") %>` are equivalent.

You use the `Response.Write()` method whenever you need to dynamically evaluate an expression and output the result to the browser. For example, you can combine VBSCRIPT looping constructs with HTML to generate a variable-length list of items.

7 Add the following to the ASP document:

```
NL  <BODY>
NL  <P>The sum of 3 and 5 is:
    <% Response.Write(3+5)%></P>
NL  <UL>List:
NL  <% For i = 1 To 5 %>
NL    <LI>This is item:
      <% Response.Write(i) %></LI>
NL  <% Next %>
NL  </UL>
NL  </BODY>
```

Notice how it is possible to intermix scripting code and HTML. For example, a `For… Next` statement is normally treated as a single long statement. However, it is possible to suspend the statement after the `For` part (using a closing tag `%>`), enter some HTML, and pick up the statement again (using an opening tag `<%`) for the `Then` part. In this way, the HTML within the loop is written to the browser every time the loop executes.

8 If necessary, transfer the modified file to the web server and verify the results, as shown in Figure 26.2.

26.3.2.2 The Redirect method

Another useful `Response` object method is `Redirect`. The `Redirect` method is used to immediately send the user's browser to a different URL.

9 Enter the following VBSCRIPT code at the top of your document, *before* the `<HTML>` tag:

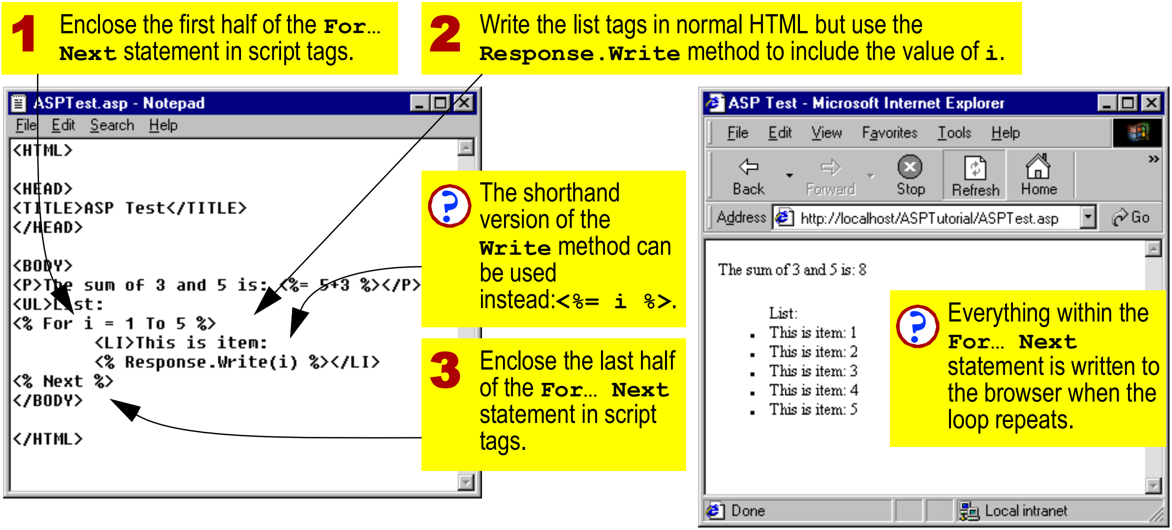
```
NL  <% Response.Redirect
    "MyFirst.html" %>
NL  <HTML>
NL  …
NL  </HTML>
```

10 Save the change and view `ASPTest.asp` in the browser. It should immediately transfer you to the URL passed as a parameter to the `Redirect` method.

If you do not have a page called `MyFirst.html` (created in Section 24.3.2.2), the redirect will fail and will return an HTTP 404 (page not found) error.

The `Redirect` method fails if it is executed after anything has been written to the HTTP response. As such, it must be located at the top of the document before any HTML tags, including `<HTML>`.

FIGURE 26.2: Use a VBSCRIPT looping construct and the ***Response.Write*** method to create a variable-length list.



A simple redirect such as this is of little use. However, when combined with the conditional branching construct in VBSCRIPT (the `If… Then` statement), the redirect feature can be used to control access to pages and the flow of a web-based application.

11 Modify the redirect statement so that it is conditional on the value of a query string, `LI` (short for “log in”):

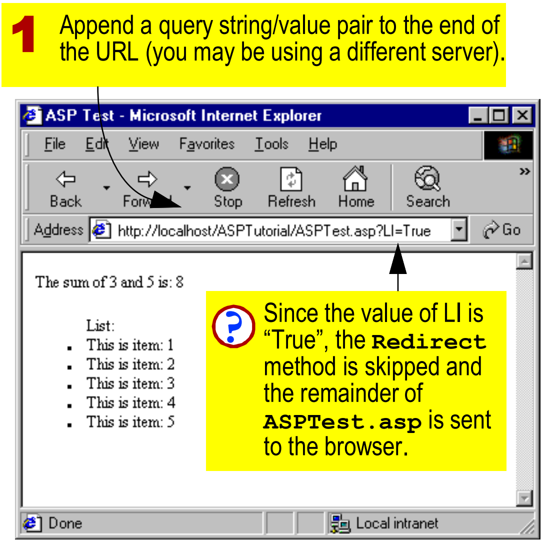
```
NL <% If Request.QueryString.Item("LI") <> "True" Then
NL Response.Redirect "MyFirst.html"
NL End If %>
NL <HTML>
NL …
NL </HTML>
```

12 Add the following query to the end of the URL in your web browser's address field:

NL `http://... /ASPTest.asp?LI=True`

13 Press the **Enter** key to view the results, as shown in Figure 26.3.

FIGURE 26.3: Add a "logged in" query string to the end of the URL.



14 Change the query string to read:

NL `... ASPTest.asp?LI=False`

15 You should be transferred to the URL specified for the `Redirect` method.

In this way, users are prevented from viewing the contents of `ASPTest.asp` unless the correct query string and value are passed in the URL.

26.3.3 Using the Request object

In Lesson 25 you learned about using query strings and forms to generate HTTP requests and send them to a web server. Using a second built-in ASP object—the `Request` object—you can extract the information contained in HTTP requests and use it in server-side processing. For example, in the previous section, you used the `Request` object to extract the value of the `LI` query string sent by the client. The value was then used by the server to make a simple decision whether to redirect the user.

26.3.3.1 Request object collections

The `Request` object takes all the information in the HTTP request and organizes it into tidy collections. For our purposes, the two most important collections are

- `QueryString` — contains the query/value pairs appended to the URL, and
- `Form` — contains the field/value pairs sent when the form's **Submit** button is pressed.

Like all collections, the contents of the **QueryString** and **Form** collections can be accessed using the **Item** method combined with the name of the key that uniquely identifies the item.

For example, assume a form has the textbox shown below:

```
NL  <HTML><BODY>
NL  <FORM METHOD="POST"
    ACTION="demo.asp">
NL  <INPUT TYPE="text"
    NAME="txtUserName">
NL  <INPUT TYPE=submit NAME="cmdSubmit"
    VALUE="Submit">
NL  </FORM>
NL  </BODY></HTML>
```

When the **Submit** button is pressed, the **Request** object containing the **QueryString** and **Form** collection is sent to the page specified in the **ACTION** attribute (in this case, a file called **demo.asp**).

Within demo.asp, it is possible to extract and use the values contained in the Request object. For example, in the code below, the user name entered into the form is saved to a local variable (**strUserName**) and then converted to uppercase:

```
NL  <HTML>
NL  <HEAD><TITLE>Demo.asp</TITLE><HEAD>
NL  <BODY>
NL  strUserName =
    Request.Form.Item("txtUserName")
NL  <P><%= UCase(strUserName) %></P>
NL  </BODY></HTML>
```

Of course, there is a shortcut: Since **Item** is the default method for all collections in ASP, you can omit it and use the following syntax instead:

```
NL  strUserName =
    Request.Form("txtUserName")
```

26.3.3.2 Processing forms

In this section, you will create an ASP page that contains a mock authorization routine for the login page you created in Section 24.5. The routine is "mock" because true authorization requires database access, which we have not yet covered.

16 Create a new ASP file called **Authorize.asp**.

Since the **Authorize.asp** page will contain ASP code only (no HTML) *do not* add the **<HTML>**, **<HEAD>**, and **<BODY>** tags.

17 Enter the following VBSCRIPT code to implement the mock login decision process.

The resulting ASP file is shown in Figure 26.4.

```
NL  <%
NL  strUserName=Request.Form.
    Item("txtUserName")
NL  strPassword=Request.Form.
    Item("txtPassword")
NL  If strUserName=strPassword Then
NL    Response.Redirect "Menu.html"
NL  Else
NL    Response.Redirect "Login.html"
NL  End if
NL  %>
```

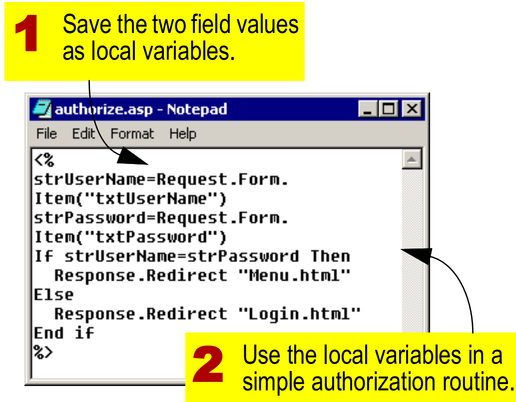
In the code above, the values from the form are assigned to local variables (`strUserName` and `strPassword`). The only reason the local variables are used is to simplify the `If… Then` statement; they could just as easily have been omitted and `Request.Form.Item(…)` used instead.

In VBSCRIPT, variables do not have to be declared before use. Contrast this with the use of the `Dim` statement in VBA (see Section 18.3.4.1).

The mock login procedure is as follows:

- If the user's user name and password are identical, then the user is transferred to the application's main menu.

FIGURE 26.4: Create a mock authorization routine using VBSCRIPT.



VBSCRIPT does not have separate data types (string, integer, etc.) and does not require variable declaration. As such, you must be careful not to misspell the names of variables in your code.

- If the user name and password are not identical, the user is transferred back to the login page.

Of course, we will change this later, but it is sufficient for our current purposes.

If you did not create a `Menu.html` page in Section 24.5, you will get an HTTP 404 (page not found) when the authorization succeeds.

18 Edit the `Login.html` file you created in Section 25.5. Replace the existing value of the `ACTION` attribute with the relative URL of the new ASP page:

```
NL  <HTML>
NL  …
NL  <FORM … ACTION="Authorize.asp">
NL  …
NL  </HTML>
```

19 Test your login-authorization pages by typing in different combinations of user names and passwords.

26.3.4 Dealing with statelessness

There are two fundamental shortcomings with the current authorization scheme (apart from its simplicity, of course):

1. If the authorization fails, the user is simply returned to the login page with no explanation
2. The authorization mechanism can be bypassed by simply typing the URL of the menu page directly into the browser.

Remember that HTTP is a stateless protocol. What that means is that the web server simply serves up pages without any memory of which user has viewed which pages. In the authorization routine that you have created, the user is redirected back to the login page on authorization failure; however, the web server has no memory of how the user got to the login page. As a consequence, it is up to the application designer to build memory into the application. A simple way to accomplish this is to use query strings.

20 Change the file name extension of `Login.html` to `Login.asp`.

Since you will add some VBSCRIPT code to the login page, it must have an .asp extension. Otherwise, the code will be ignored by the web server.

21 Open `Login.asp` in your editor.

22 Add the following conditional heading before the form. You may choose to add heading tags or other formatting.

```
NL  <HTML>
NL  …
NL  <BODY>
NL  <H3><% If Request.QueryString.
    Item("LI") = "Fail" Then
```

```
NL   Response.Write("Login incorrect:
     please try again")
NL Else
NL   Response.Write("Please enter your
     user name and password")
NL End If %></H3>
NL <FORM … ACTION="Authorize.asp">
NL …
NL </FORM></BODY></HTML>
```

Another way to accomplish this without using the **Response.Write** method within the code is to keep the HTML code outside of the scripting tags. For example, the code below yields the same result:

```
NL <HTML>
NL …
NL <BODY>
NL <% If
   Request.QueryString.Item("LI") =
   "Fail" Then %>
NL   <H3>Login incorrect: please try
     again</H3>
NL <% Else %>
NL   <H3>Please enter your user name
     and password</H3>
NL <% End if %>
NL <FORM … ACTION="Authorize.asp">
NL …
NL </FORM></BODY></HTML>
```

With the addition of this code (either of the variations above will work), the login page will include an error message whenever the request contains the query/value pair **LI=Fail**. If request contains a different value for **LI**, or if **LI** is undefined, then the page will contain a simple instructional message.

23 Edit **Authorize.asp** and add a query string (recall Section 25.3.1) to the redirect statement for failed authorization:

```
NL <%
NL strUserName=Request.Form.
   Item("txtUserName")
NL strPassword=Request.Form.
   Item("txtPassword")
NL If strUserName=strPassword Then
NL   Response.Redirect "Menu.html"
NL Else
NL   Response.Redirect
     "Login.asp?LI=Fail"
NL End if
NL %>
```

Remember to change **Login.html** to **Login.asp** in the redirect statement.

24 Test your application to ensure that the conditional heading is working correctly.

By carrying the value of the query string **LI** from one page to the next, a primitive form of memory has been added to the application.

26.3.5 Robust authorization

A robust authorization mechanism is one that cannot easily be defeated or bypassed. As it now stands, our authorization mechanism can be side-stepped by typing **`Menu.html`** into the browser's address window.

It would be straightforward to add a conditional redirect to the start of the menu file using VBSCRIPT:

```
NL  <% If Response.QueryString("LI") <>
    True Then
NL    Response.Redirect "Login.asp"
NL  End If %>
NL  <HTML>
NL  …
NL  </HTML>
```

However, this provides very little additional protection since a knowledgeable user could simply type **`Menu.asp?LI=True`** into the browser's address window.

There are at least two ways around this problem:

1. Use a hash function to generate a value for **`LI`** that is difficult for users to guess. For example, instead of setting **`LI=True`**, the hash function would generate a longer value such as **`LI=FJ910392XZZ381847292873086`** that could also be independently generated and verified on subsequent pages.
2. Use ASP's built-in **`Session`** object to read and write session-level variables on the server.

In the following sections, we will explore the use of hash values and query strings. In Lesson 27, we will use ASP's session-level variables to implement a slightly different authentication scheme.

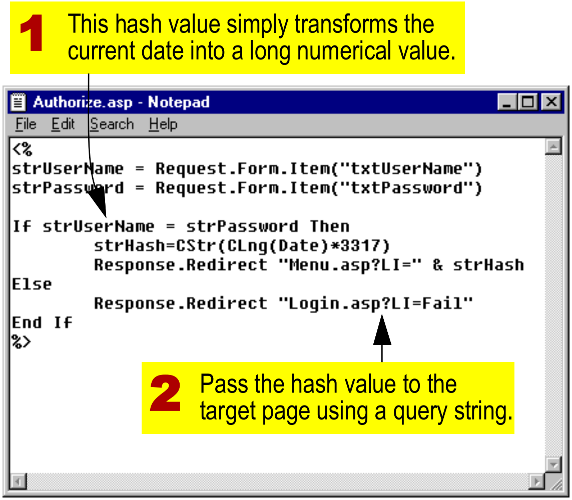
26.3.5.1 Creating a hash function

A hash function is simply a function that takes some known values and transforms them into a jumbled "hash" of digits and/or characters.

25 Make the following changes to **`Authorize.asp`**, as shown in Figure 26.5:

```
NL  <%
NL  strUserName=Request.Form.
    Item("txtUserName")
NL  strPassword=Request.Form.
    Item("txtPassword")
NL  If strUserName=strPassword Then
NL    strHash = CStr(CLng(Date)* 3317)
NL    Response.Redirect "Menu.html?LI="
      & strHash
NL  Else
NL    Response.Redirect
      "Login.asp?LI=Fail
NL  End if
NL  %>
```

FIGURE 26.5: Generate a hash value when authorization is successful.



This code calculates an extremely simple hash value and appends it as a query string to the redirection URL. The elements of this particular hash function are:

- `Date()` is a built-in function that returns the current date. The underlying representation for dates and times is numeric.
- `CLng()` is a built in function that converts a number into a long integer. In this case, it converts the number returned by the `Date()` function.
- 3317 is just an arbitrary multiplier
- `CStr()` is a built-in function that converts any data into a string of characters.

Like all hash functions, this one takes one or more publicly known values (the current date) and transforms it in a (hopefully) non-obvious way into a longer series of digits. Since the same hash function can be calculated on other pages and compared to the value being passed around in the query string, it is possible to identify users who have logged in successfully.

26.3.5.2 Using a hash value to confirm authorization

In this section, you will recalculate the hash value in the menu page and compare it to the value passed in the query string. If they are the same, then it is likely that the user was properly authorized by the code in `Authorize.asp`.

26 Rename `Menu.html` to `Menu.asp` and open the file for editing.

27 Add the following code to the top of the file, as shown in Figure 26.6:

```
NL  <%
NL  strHash=CStr(CLng(Date)*3317)
```

```
NL If Request.QueryString.Item("LI")
   <> strHash Then
NL   Response.Redirect "Login.asp"
NL End If
NL %>
NL <HTML>
NL …
NL </HTML>
```

28 Open `Authorize.asp` for editing and replace the reference to `Menu.html` with `Menu.asp`.

If the hash value passed in the query string is not identical to the hash value calculated on the menu page, the user is redirected to the login page.

If you make a typographical error and the hash function you use in `Authorize.asp` is different from the one in `Menu.asp`, you will never see the menu. Instead, each time you will be immediately redirected to the login page. Such redirection bugs can be tricky to diagnose because you are immediately transferred off the page with the error during testing.

Naturally, an authorization mechanism based on visible hash values is only useful if the hash function is secret and is very difficult to guess. In this example, the hash values change every day. However, the values change in such a predictable and linear manner that it would be very easy for a hacker to induce the hash function and gain unauthorized access to the application.

There is an entire science devoted to the creation of difficult-to-guess hash functions. In general, the function should generate long values that have a highly non-linear relationship with the inputs.

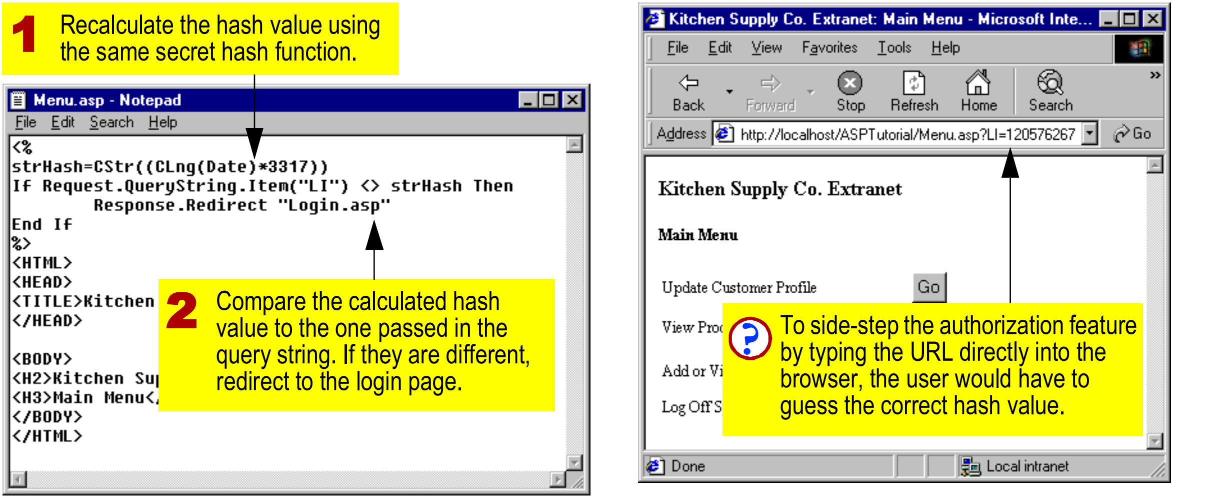
You might be tempted to pass a simple authorization string (e.g., `LI=True`) via a hidden field in a form (revisit Section 25.3.4 for more information on different field types in HTML). Although hidden fields are "less visible" than query strings, they are in plain view in the form's HTML source. As such, they provide no more security than query strings.

26.4 Discussion

26.4.1 Server-side scripting and CGI

The earliest web servers used a mechanism called the Common Gateway Interface (CGI) to allow the web server to interact with other server-side programs. The difference between CGI programming and server side scripting is that in server-side scripting:

FIGURE 26.6: Create a conditional redirect feature that compares a hash value to a value passed in a query string.



- the client requests a document containing special tags and code interspersed within the page's HTML structure; and
- the code is executed on the server before sending the document to the client.

In CGI programming, the URL in the client's HTTP request is not a document at all, but the name of an executable program in a special directory on the server (e.g., `http://<server name>/cgi-bin/run_me`). When the server receives the request from the client, the program is executed. Since the CGI program is not embedded within an HTML document, it is the CGI program's responsibility to "write" a response that can be sent back to the client. That is, the CGI program must do its task (e.g., perform a database lookup) *and* generate all the HTML text to be sent back to the browser.

Although server-side scripting is less wasteful of server resources (a major consideration when the potential exists for thousands of users to access your site at the same time) and is easier to create and deploy than CGI, CGI is still widely used. One reason for its longevity is that it is a vendor-independent standard that is bundled with all web servers. ACTIVE SERVER PAGES is freely bundled, but only with the MICROSOFT's INTERNET INFORMATION SERVER[1]. COLDFUSION is available for a wider range of web servers, but is sold as a separate product. CGI is free.

A number of cross-platform open-source server-side scripting languages have emerged, including PHP (see www.php.net) and JAVA SERVER PAGES (JSP). Although the format of the tags and the syntax of the language varies depending on which scripting infrastructure you use, the basic principles of server-side scripting are identical in ASP, COLDFUSION, PHP, and JSP.

Another reason for CGI's continued use is the shear bulk of legacy code written for CGI applications. Although a CGI program can be written in just about any language (C++, FORTRAN, VISUAL BASIC), a freely available interpreted language called PERL caught on early as an easy way to parse text and dynamically create HTML (remember, the output of a CGI script is sent back to the client, not an intermediate page, as in ASP).

[1] CHILLISOFT sells a product that provides an ASP functionality for non-IIS web servers.

26.5 Application to the project

29 Create a new ASP file called `Menu_process.asp` to process the information passed to the server by the menu page.

HINT: You can use the `If… ElseIf…` construct to check whether specific name/value pairs were passed in the HTTP request. For example:

```
NL  <%
NL  If Request.Form("cmdCustomer")="Go" Then
NL    strRedirect="Customer.asp"
NL  ElseIf Request.Form("cmdProduct")="Go" Then
NL    strRedirect="ProductList.asp"
NL  ElseIf Resquest.Form("cmdOrder")="Go" Then
NL    strRedirect="Order.asp"
NL  ...
NL  End If
```

```
NL  Response.Redirect strRedirect
NL  %>
```

VBSCRIPT also supports a **`Select… Case… End Select`** construct that is functionally identical (but slightly more elegant) than the **`ElseIf`** construct.

Lesson 27: Using sessions

27.1 Introduction: Dealing with statelessness (part 2)

In Section 26.3.4, you used a query string to maintain state information across different pages of your web application. Specifically, when the user entered the correct user name and password, the state variable **`LI`** was set to "True" and passed from page to page in the URL (e.g., **`http://…/ASPTest.asp?LI=True)`**. Thus, it was possible for the server to recognize the users logged-in "state" even though the TCP/IP connection used by the Internet is stateless.

Of course, one downside of using query strings (or even hidden fields) to pass sensitive state information (such as whether the user has been authenticated) is that this information is in plain view. Thus, in order to make it more difficult to bypass the authorization logic, we changed the value of **`LI`** from "True" to something less obvious (a hash value) in Section 26.3.5.

The use of hash values and query strings is extremely common on the Internet (just log in to a site that retains state information such as **`www.expedia.com`** and examine the contents of your browser's URL window). However, server-side scripting environments (such as ASP and JSP) provide a simpler mechanism for storing state information across pages: sessions.

27.1.1 ASP's Session object

In Lesson 26, we used two of ASP's built-in objects: **`Response`** and **`Request`**. These two objects correspond exactly to the HTTP response and request constructs, so they are fairly easy to understand. There is no HTTP counterpart for the **`Session`** object, however (hence the need for it in the first place).

A session is defined as a user's visit to a web-based application. Within the session, the user may visit many different pages within the application and even leave the application briefly and return. The **`Session`** object is simply an "after market" solution to the statelessness of the underlying HTTP protocol. It permits ASP designers to read and write server-side session variables that persist throughout the entire session, regardless of which pages are viewed.

27.1.2 Session variables and scope

The "scope" of normal ASP variables is the page on which they are created. Thus, although you could define a variable on a page:

```
NL   <HTML>
```

```
NL  <BODY>
NL  <% strFavColor="Blue" %>
NL  <P>My favorite color is:
    <%= strFavColor %></P>
NL  </BODY>
NL  </HTML>
```

the value of `strFavColor` would not be available on any other page. Session variables, in contrast, are available on any page in the application for the duration of the session.

27.1.2.1 Declaring a session variable

To create a session variable, you simply give it a name within the collection of variables stored in the `Session` object. For example, the following assignment statement can be used to store the user's favorite color in a session variable called `strFavColor`:

```
NL  <HTML>
NL  <BODY>
NL  <% Session("strFavColor") = "Blue"
    %>
NL  </BODY>
NL  </HTML>
```

27.1.2.2 Using a session variable

Now, on a completely different page (within the same session), the value of the session variable can be recalled:

```
NL  <HTML>
NL  <HEAD>
NL  <TITLE>A Different Page</TITLE>
NL  </HEAD>
NL  <BODY>
NL  <P>Your favorite color is:
    <%= Session("strFavColor") %></P>
NL  </BODY>
NL  </HTML>
```

Unlike a query string, the information in a session variable never gets passed back to the client. Instead, it is stored in the server's memory and can be made invisible to the user. Consequently, session variables can be used to implement a simple-but-robust authorization mechanism.

27.2 Learning objectives

- create session variable
- understand how session variables can be used to simplify authentication
- use Session.Abandon to free-up server resources
- understand the role of cookies in providing session functionality

27.3 Exercises

In this section you will modify the authorization scheme you created in Lesson 26 to use a session variable instead of a query string.

In the exercises that follow, you will replace some of the code you wrote in Lesson 26 with new code based on session variables. If you have a sentimental attachment to the versions of **`Authorize.asp`** and **`Menu.asp`** that use hash values and query strings, you should save a backup copy of these files to a different directory before continuing.

27.3.1 Controlling branching using session variables

1 Edit **`Authorize.asp`** to replace the hash value and query string code with the following code:

```
NL  <%
NL  strUserName=Request.Form.
    Item("txtUserName")
NL  strPassword=Request.Form.
    Item("txtPassword")
NL  If strUserName=strPassword Then
NL    Session("LI") = "True"
NL    Response.Redirect "Menu.asp"
NL  Else
NL    Session("LI") = "Fail"
NL    Response.Redirect "Login.asp"
NL  End if
NL  %>
```

2 Edit **`Menu.asp`** to use the value of the session variable instead of the hash value to control access:

```
NL  <%
NL  If Session("LI") <> "True" Then
NL    Response.Redirect "Login.asp""
NL  End If
NL  %>
NL  <HTML>
NL  …
NL  </HTML>
```

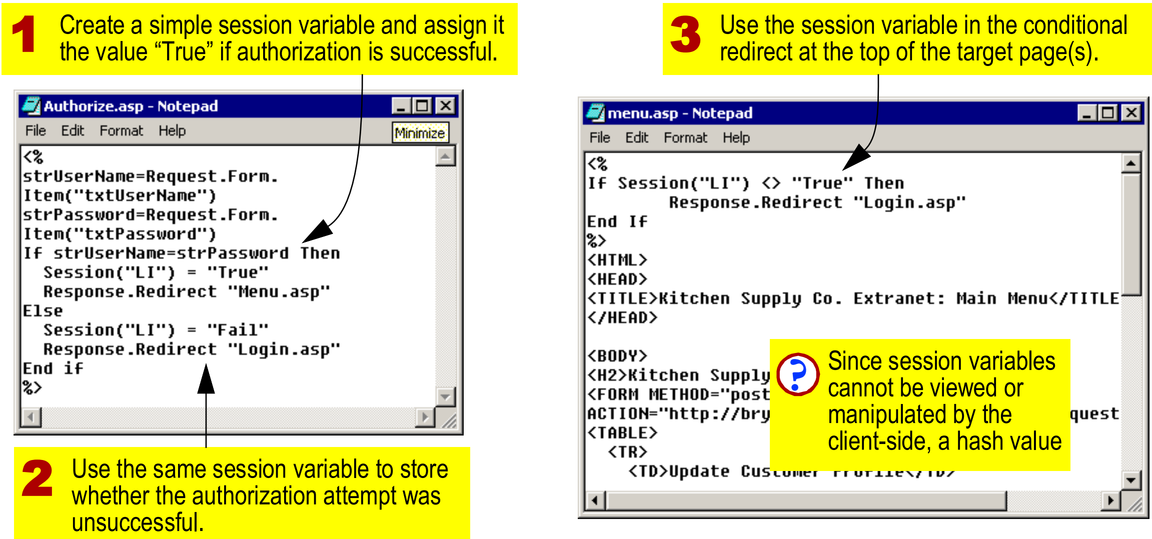
Remember that a **`Redirect`** method will not work if the server has already written something to the HTTP response. Thus, you must ensure that the VBSCRIPT code is located *above* the **`<HTML>`** tag.

The changes to **`Menu.asp`** are shown in Figure 27.1.

3 Edit **`Login.asp`** to use the value of the session variable instead of the query string to display the “login incorrect” message:

```
NL  <HTML>
NL  …
NL  <BODY>
NL  <% If Session("LI") = "Fail" Then
    %>
NL    <H3>Login incorrect: please try
      again</H3>
NL  <% Else %>
```

FIGURE 27.1: Replace the hash value/query string authorization mechanism with one based on session variables.




```
NL   <H3>Please enter your user name
     and password</H3>
NL  <% End if %>
NL  …
NL  </HTML>
```

Clearly, the **`Session`** object makes keeping track of a user’s progress through the application much easier.

27.3.2 Ending a session

Although the server will eventually end a session if a specified amount of time elapses without activity (e.g., 20 minutes), it is important to realize that each session ties up a certain amount of server resources. If there are many thousands of users connected to the site simultaneously, then the server has to keep track of many thousand individual sessions. In extreme circumstances, the requirements of managing all the sessions can overwhelm the web server. Having a logoff mechanism that allows users to explicitly end their session has two advantages:

1. It frees session-specific resources as soon as possible.
2. It eliminates the security risk created by users walking away from machines while an authorized session is still active.

You will notice that most secure web applications (such as on-line banking) have explicit logoff procedures.

The easiest way to end a session in ASP is to transfer the user to a page that contains a **`Session.Abandon`** statement. The **`Abandon`** method destroys the session and any session-level variables stored with the session.

4 Rename your **`Logoff.html`** file to **`Logoff.asp`**.

5 Add a message similar to “Thank you, you are now logged off” to the body of the document.

6 Add the following code to the end of the document (following the **`</HTML>`** tag):

```
NL  <HTML>
NL  …
NL  <P>Thank you, you are now logged
    off</P>
NL  …
NL  </HTML>
NL  <% Session.Abandon %>
```

Once the session is abandoned, the value of **`Session("LI")`** no longer equals “True”. Thus, even if an unauthorized user uses the browser’s **Back** button or opens a cached copy of page, he or she will be unable to get past the conditional redirects that you will place at the top of all your content pages in Section 27.5.

27.3.3 Limiting page caching

To create the illusion of speed, web browsers typically “cache” pages as they are visited. Thus, when you hit the back button or revisit a page in other ways, there is a good chance that you are looking at the copy of the page that is cached on your local hard drive, not the page on the web server. Thus, it is possible so see the contents of the cached pages even after logging out. If you have sensitive data on a page (e.g., banking information) you can use the ASP **`Response`** object to tell browser not to cache the page:

```
NL  <% Response.Expires = 0 %>
NL  <HTML>
NL  …
NL  </HTML
```

Whether the browser actually obeys this directive is beyond your control as a web designer.

27.4 Discussion: Cookies and sessions

The `Session` object is an easy way to achieve persistence of variables within a session. Although the feature may seem complicated, the implementation of the `Session` object is not that different from the query string/hash value approach used in Section 26.3.5. The main difference is that the Session ID (a hash value that uniquely identifies a session) is created automatically by the server and passed back and forth between the server and client as an HTTP **cookie** instead of a query string.

When you send a request to an ASP page for the first time, a number of things occur:

- The server creates a Session ID value that is unique to the session (assume "123" for simplicity's sake).
- Any session variables created during the user's interaction with the web application are stored in the server's memory and tagged as belonging to Session ID = 123 (remember that there may be thousands of concurrent users, each with a unique Session ID and set of session variables).
- The server appends the SessionID to an HTTP header variable called `HTTP_COOKIE`. The header is included in the response sent back to the client.
- The browser extracts the cookie from the server's HTTP response . The data in the cookie is saved to a small text file on the client computer along with sender information that allows the browser to associate the cookie with the name of the server that sent it.
- Whenever the client sends an HTTP request to the server, it checks to see whether it has saved a cookie associated with that particular server. If so, the browser sends the data in the cookie (e.g., Session ID = "123") back to the server.

The server can use the cookie value to "recognize" the session and retrieve any server-side variables assigned to it. In this way, ASP creates the illusion of a contiguous session.

Cookies are site-specific. That is, your browser is responsible for ensuring that a cookie deposited by Server X is not sent to Server Y, and vice-versa.

27.4.1 Session example

To illustrate the use of cookies, do the following:

7 Close your browser. If you have multiple browser windows open, make sure they are all closed. This eliminates any session-level (temporary) cookies you may have open.

8 Re-open a browser windows and send an HTTP request to `EchoRequest.asp` (recall Section 25.3).

At first, the “Server Variables” section (near the bottom of the page) should not contain any information about an ASP session. However, when you hit the **Refresh** button, you should see that a Session ID has been deposited on your computer and sent back to the server. The contents of the cookie should look something like the following:

```
HTTP_COOKIE =
ASPSESSIONIDQQGGQQFB=DIHJOFOANKGNILOAFON
GJIDL
```

27.4.1.1 Cookie expiration

How do the two computers know when the session is over? Each cookie has an expiry date. The Session ID cookies, which are created automatically by the ASP-enabled web server, expire immediately at the end of the session (e.g., when the browser is closed). On the server-side, a session time-out can be set by the server administrator. When there is no activity from the client for 20 minutes (the default), the session is closed and all the server-side session variables are released.

27.4.1.2 Longer-term cookies

Of course, not all cookies are created automatically. In ASP for example, the `Response.Cookies` collection can be used to create cookie key/value pairs and set cookie-level properties such as expiry date. If a cookie’s expiry date is set to a date in the future, the cookie file saved on the client’s hard drive is not deleted at the end of the session. In this way, the cookie values are available for many sessions, even if the browser and/or computer are shut down.

Persistent cookies are used by many web sites to automatically identify you whenever you visit the site. From a user’s perspective, cookies are convenient if they are used to personalize the site or eliminate the need to login manually with each visit. From the site owner’s perspective, cookies are valuable because they permit the behavior of users within the site to be tracked, both within a particular visit and across multiple visits.

For example, if you visit `www.volvo.com` ten times in one week to look at the same car, you may be a promising sales lead for VOLVO. However, without a persistent cookie, there is no reliable way to know that the ten visits came from the same computer. Naturally, the use of cookies in this manner leads to concerns about privacy. And at a more basic level, some people simply do not like the idea of a remote computer writing something to their hard drive.

It is important to realize that a cookie's potential to do harm is constrained by the HTTP protocol on one hand and your browser (which is solely responsible for the creation and management of cookie files) on the other. Cookies are just key/value pairs—nothing more. A cookie cannot erase your hard drive, implant a virus, or spy on you.

27.4.1.3 Cookie design issues

Given the number of times a cookie's contents are sent to the server, it is best to minimize the amount of data stored in the cookie. As such, most cookies contain nothing but an identification value. The identification value (or primary key, if you prefer) is used to access more extensive visitor information that is stored on a server-side database. Where does the more extensive server-side visitor information come from? Typically, you provide it when you register for a site, enter a contest, or fill in any type of form.

If you provide the site with certain key information (such as your postal code or social insurance number), it may be possible for the site to combine data from multiple sources to get a very clear picture of who you are. From a privacy point of view, the problem is not the cookie. Rather, it is the widespread practice of reselling and merging data that you have willingly provided about yourself.

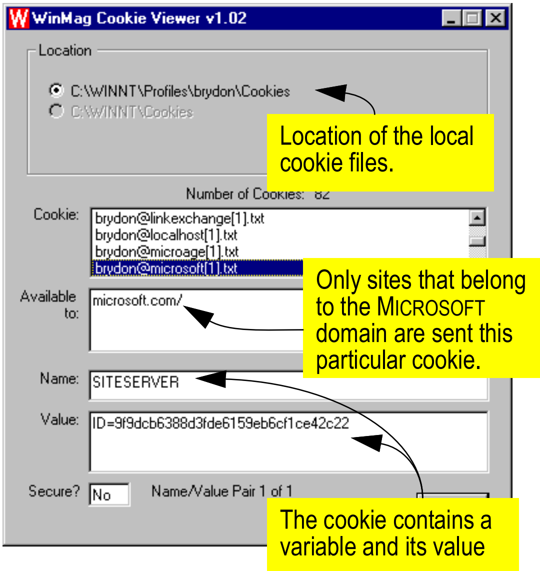
If you are curious about the contents of the cookies on your computer's hard drive, you can download one of the many free "cookie viewers" available over the Internet. For example, Figure 27.2 shows the contents of a cookie left by a MICROSOFT web site.

27.5 Application to the project

It is important that each page of your extranet require authentication. As such, you should include a conditional redirect at the top of each content page to ensure that the only way the page can be accessed is if a session variable is set to some value first. The only way to set the value of the session variable is to login successfully.

9 Add a conditional redirect (similar to that shown on the right in Figure 27.1) to all the

FIGURE 27.2: The contents of a cookie created by a MICROSOFT web site.



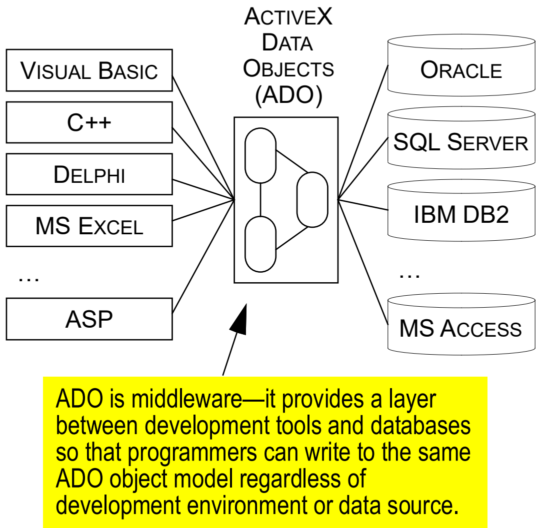
content pages of your web-based order entry system. Of course, the login page should not include this code.

Since you are adding script to the pages, the file names must be changed to end with the .asp extension.

28.1 Introduction: What is ADO?

ACTIVEX DATA OBJECTS (ADO) is a MICROSOFT technology that enables programmers to access data stored in virtually any type of database. The nice thing about ADO is that it provides a *lingua franca* between many different development tools on one side and many different databases on the other. The relationship between development tools, ADO, and databases is shown in Figure 28.1.

FIGURE 28.1: The relationship between development tools, ADO, and databases.



ADO is database middleware and subsumes ODBC, which you saw in Lesson 9 and again in Lesson 23. Although ODBC has been widely adopted and remains well supported, ADO is more powerful and provides a more flexible, object-oriented, means of manipulating a database using a programming language.

The ADO object model consists of three classes of objects. Each object has its own properties and methods that help simplify various aspects of data access:

1. **Connection object** — stores all the information about a connection to a database and provides methods to manage the connection. The database may be file-based (e.g., a MICROSOFT ACCESS .mdb file) or client/server (e.g., ORACLE, SQL SERVER, INTERBASE).
2. **Command object** — stores information about commands sent to the database. Some database commands return data

(e.g., SQL **SELECT** statements) while others perform processing (e.g., ACCESS action queries and stored procedures written using SQL **INSERT**, **UPDATE**, and **DELETE** statements).

3. **Recordset object** — stores the results of a query against a database. Although a recordset is invisible, it has the same structure as a datasheet view of data in ACCESS (records in rows, fields in columns). Changes made to the data in recordset objects can normally be saved to the database.

Since **Recordset** objects contain the data, they are the most important objects in the ADO model for most applications. **Connection** and **Command** objects are used as a means of specifying which data should appear in the recordset.

If you do not have a (recent) version of ADO installed on your machine or if you want to access MICROSOFT's on-line documentation, you should visit the ADO web site at **www.microsoft.com/data/ado**.

28.2 Learning objectives

- understand how ACTIVEX DATA OBJECTS can be used to create dynamic web content
- create ADO Connection, Command, and Recordset objects
- display records from Recordset object
- use ADO to "up-size" a web-based application
- use database look-up for user authorization
- modify values in a database using an HTML form

28.3 Exercises

28.3.1 Creating a Connection object

In this section, you will create a **Connection** object and configure it to point to a web-based version of your ACCESS order entry database.

28.3.1.1 Setting up the source database

You have been provided with an ACCESS database file called **OrderEntryWeb.mdb** in the project package. The database is similar to the one you created in previous lessons for your ACCESS application. However, the **OrderEntryWeb.mdb** file contains a number of queries that are used to help you complete your web-based application in Lesson 29.

1 Copy the `OrderEntryWeb.mdb` file from the project package to a folder on your web server. Make a note of the full path of the file on the server (e.g., `C:\Documents and Settings\brydon\My Documents\KitchenSupply\OrderEntryWeb.mdb`).

It does not matter where in the web server's directory structure you locate the database file. However, if your web server supports MICROSOFT FRONTPAGE extensions, it is best to locate your database file in a folder that is *not* part of a "FRONTPAGE web".

28.3.1.2 Creating the EmployeeList file

2 Create a new ASP file and save it as `EmployeeList.asp`.

3 Add the core HTML tags as well as a title and heading.

4 In the header section (or anywhere near the top of the file), add the following code to create and configure a new `Connection` object:

```
NL  <HEAD>
NL  …
NL  <%
NL  Set objCon =
    Server.CreateObject("ADODB.
    Connection")
NL  objCon.Provider =
    "Microsoft.Jet.OLEDB.4.0"
NL  objCon.ConnectionString =
    "path\OrderEntryWeb.mdb"
NL  objCon.Open
NL  %>
NL  </HEAD>
```

"*path*" in the `ConnectionString` property refers to the *physical* location of your database on the web server. To use ADO with a MICROSOFT JET (i.e., ACCESS) database, you must specify the complete location (including drive letter and path) of the database file.

5 To make sure the database connection is opened successfully, add the following verification code to the body of the document:

```
NL  <HTML>
NL  <HEAD>
NL  <TITLE>Employee List</TITLE>
NL  <%
NL  …
NL  objCon.Open
NL  %>
NL  </HEAD>
NL  <BODY>
```

```
NL   State = <%= objCon.State %>
NL   </BODY>
NL   </HTML>
```

The state property can be used to determine whether the connection is open (`state` = 1) or closed (`state` = 0).

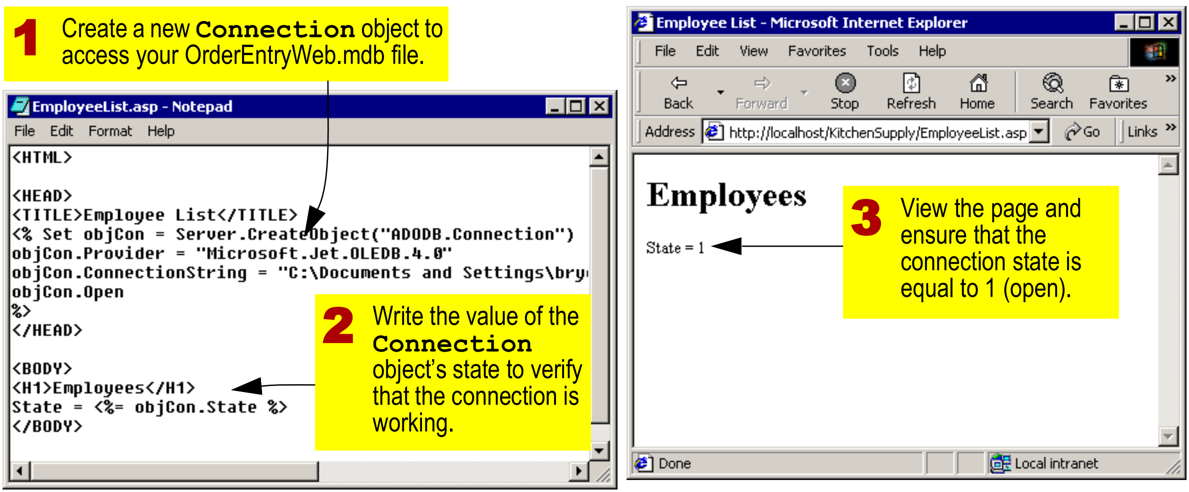
6 Test the `Connection` object, as shown in Figure 28.2.

28.3.1.3 Understanding the ADO code

The code you have just written introduces a few new VBSCRIPT constructs that warrant a brief description.

- The `Set` statement — When you assign an intrinsic data type (e.g., integer, string, etc.) to a variable, you use an equals sign. However, when you set a variable to "point" to an existing object, you must use the `Set` statement. In this case, the variable `objCon` is set to point to a `Connection` object.

FIGURE 28.2: Create and test a *`Connection`* object for a MICROSOFT JET database.



- **`Server.CreateObject("ADODB.Connection")`** — This method tells the web server to create a new **`Connection`** object. The ADODB prefix is required because ActiveX components other than ADO may have an object called "connection".

- **`objCon.Provider`** — The **`Provider`** property of the **`Connection`** object is set to a predefined string for accessing Microsoft Jet databases. You will see in Section 28.3.5 that it is possible to make connections to other types of data providers.

- **`objCon.ConnectionString`** — The name of the database to open is provided in the connection string.

- **`objCon.Open`** — The **`Open`** method opens the connection to the specified database.

In summary, the code in Figure 28.2 tells the server to create a **`Connection`** object, sets a variable (**`objCon`**) to point to the new object, sets the properties of the object so that it knows about a Microsoft Access database file stored on the server, and opens the database connection.

28.3.2 Creating a Command object

A **`Command`** object simply stores a command to be sent to the database using Structured Query Language (SQL). The SQL commands are translated by the **provider** (in this case **`Microsoft.Jet.OLEDB.4.0`**) into a format understood by the data source. Because the database-specific provider takes care of the translation, all you need to know to access data over an ADO connection is a handful of SQL commands.

7 Create a new variable called **`objCmd`** and set it to point to a new **`Command`** object. You can add this to the scripting code at the top of your **`EmployeeList.asp`** file:

```
NL  <HTML>
NL  <HEAD>
NL  …
NL  <%
NL  Set objCon =
    Server.CreateObject("ADODB.
    Connection")
NL  objCon.Provider =
    "Microsoft.Jet.OLEDB.4.0"
NL  objCon.ConnectionString =
    "path\OrderEntryWeb.mdb"
NL  objCon.Open
NL  Set objCmd = Server.CreateObject
    ("ADODB.Command")
NL  %>
NL  </HEAD>
NL  …
NL  </HTML>
```

Although the code above creates a **`Command`** object, the object knows nothing about what it should do, which database it should connect to, and so on. Thus, just as we set the properties of the **Connection** object in Section 28.3.1, we have to set the properties of the **Command** object before using it in subsequent code.

8 Add the following code to specify the SQL statement and the database connection to be used by the **Command** object:

```
NL  <HTML>
NL  <HEAD>
NL  …
NL  <%
NL  …
NL  objCon.Open
NL  Set objCmd = Server.CreateObject
    ("ADODB.Command")
NL  With objCmd
NL    .CommandText = "SELECT * FROM
    Employees"
NL    .CommandType = 1
NL    Set .ActiveConnection = objCon
NL  End With
NL  %>
NL  </HEAD>
NL  …
NL  </HTML>
```

The resulting **EmployeeList.asp** file is shown in Figure 28.3.

Within the **With objCmd… End With** statement, **objCmd** is assumed to be the object being referred to. So instead of writing **objCmd.CommandType**, you can omit the object name and just write **.CommandType**. The **With… End With** construct in VBSCRIPT exists simply to save you some typing.

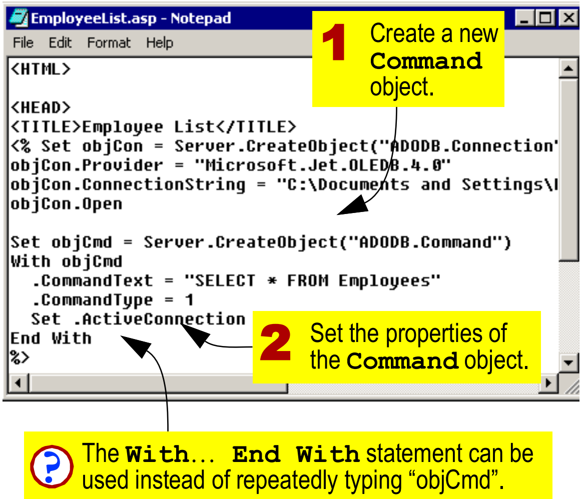
The **CommandText** property is straightforward—it is a SQL **SELECT** statement. The **CommandType = 1** statement tells the target database that the **CommandText** property contains a textual query (versus the name of a table or the name of a stored procedure, etc.). Finally, each **Command** object needs to act on an active database connection. In this case, the **ActiveConnection** property is set to point to the **Connection** object created in Section 28.3.1.

28.3.3 Creating a Recordset object

Once the **Command** object is set up correctly, creating a **Recordset** object based on the results of the embedded SQL query is straightforward.

As is the case with many MICROSOFT technologies, ADO provides many different ways to do the same thing. Although this is powerful and convenient for experienced users, it is frustrating and confusing for new users. The method used

FIGURE 28.3: Create a ***Command*** object based on a SQL ***SELECT*** query.



to create recordsets used here is more complex than it needs to be. However, it permits greater control over the type of **`Recordset`** returned. Controlling the recordset type becomes important if you need to update the data in the database.

9 Create a new **`Recordset`** object called **`rsEmp`** using the now-familiar **`Server.CreateObject`** method (see the code example with the next step if you are stuck).

10 Add the following code to set the properties of the **`Recordset`** object:

```
NL  <HTML>
NL  <HEAD>
NL  …
NL  <%
NL  …
NL    Set .ActiveConnection = objCon
NL  End With
NL  Set rsEmp = Server.CreateObject
    ("ADODB.Recordset")
NL  With rsEmp
NL    .CursorType = 1
NL    .LockType = 3
NL    .Open objCmd
NL  End With
NL  %>
NL  </HEAD>
NL  …
NL  </HTML>
```

Although the **`CreateObject`** method creates a new **`Recordset`** object, the **`Recordset`** object does not contain any data until the **`Open`** method is invoked.

The **`Recordset.Open`** method requires that a valid **`Command`** object be passed as an argument. The **`Command`** object is used when opening the recordset to determine which records are retrieved from the database. And the **Command** object uses its **ActiveConnection** property to know how to connect to the database. In a nutshell, this is how ADO works: the **Recordset** object depends on the **Command** object and the **Command** object depends on the **Connection** object.

11 To ensure that the SQL query in your **Command** object returns data, add the following verification code to the body of your document:

```
NL  <HTML>
NL  <HEAD>
NL  <TITLE>Employee List</TITLE>
NL  <%
NL  …
NL  %>
NL  </HEAD>
NL  <BODY>
NL  <P>State = <%= objCon.State %></P>
NL  <P>End of File = <%= rsEmp.EOF %></
    P>
NL  </BODY>
NL  </HTML>
```

The **EOF** property of a **Recordset** object returns True if the recordset is at the "end of file" marker and False otherwise. When a recordset is opened, its record pointer is set to the first record; however, if the recordset is empty (i.e., there is no first record) the **EOF** property is set to True.

12 Test your code, as shown in Figure 28.4.

28.3.4 Viewing a Recordset's contents

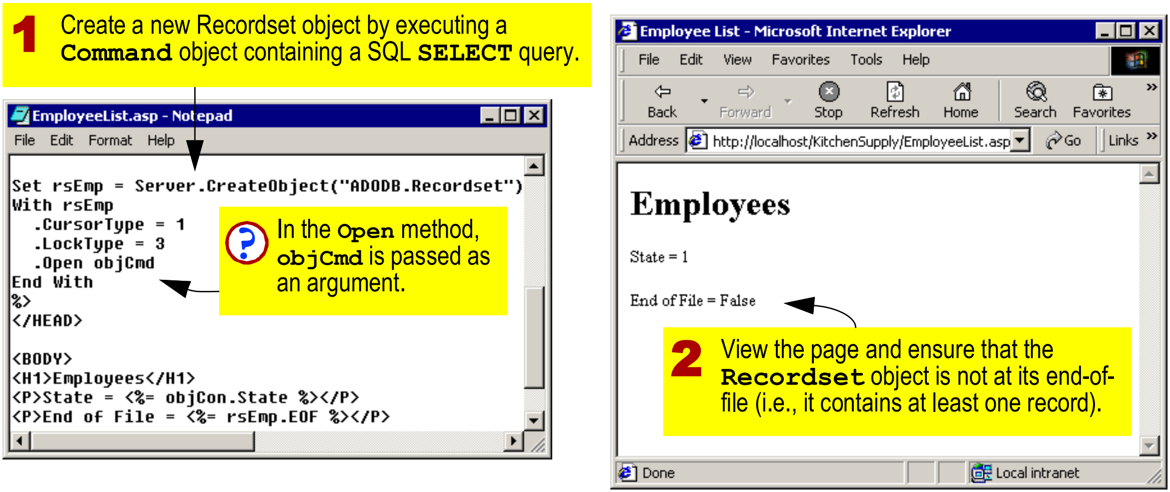
Although an ADO **Recordset** object is invisible, its logical structure is identical to that of a datasheet in ACCESS. The most important thing to know about using a recordset is that there is a record pointer (or recordset cursor, if you prefer) which points to the "current row". To access a particular record, you must use the record pointer to navigate through the individual records.

Although an ADO **Recordset** and a DATA ACCESS OBJECTS (DAO) **Recordset** (recall Lesson 21) have many important differences, the basic concept of a recordset is the same in both cases.

Once you have set the record pointer to a particular row in a recordset, you can access the data in the record by using the **Fields** collection. For example, to get the current value of the **emp_lname** field in the **rsEmps** recordset, you use the following syntax:

```
NL  … rsEmps.Fields("emp_lname") …
```

FIGURE 28.4: Create and test a **`Recordset`** object.



Unlike VBA, VBScript does not support use of the "**!**" shortcut. Thus, the following syntax returns an error: **`rsEmps!emp_lname.`**

13 Delete the verification code in the body of the document. It is no longer required.

14 Add a loop to cycle through all the records in the `Recordset` object:

```
NL  <HTML>
NL  <HEAD>
NL  <TITLE>Employee List</TITLE>
NL  <%
NL  ...
NL  %>
NL  </HEAD>
NL  <BODY>
NL  <UL>
NL  <% Do Until rsEmp.EOF %>
```

```
NL    <LI>
NL    <%= rsEmp.Fields("emp_lname") &
      ", " & rsEmp.Fields("emp_fname")
      %>
NL    </LI>
NL    <% rsEmp.MoveNext
NL  Loop %>
NL  </UL>
NL  </BODY>
NL  </HTML>
```

Note that the record pointer is explicitly moved from one record to the next using the recordset's **MoveNext** method. There are other cursor manipulation methods, such as **MoveFirst**, **MoveLast**, and **MovePrevious**.

If you forget the **MoveNext** method within the loop, **EOF** will never be set to True because the record pointer will never be moved from the first record. In such a case, the web server will become caught in an endless loop.

15 Test the employee list, as shown in Figure 28.5.

Congratulations: you have just used created a dynamic web page that pulls its content from a database.

28.3.5 Swapping data sources

It is generally *not* a good idea to build a web based application on top of an ACCESS database. The JET database engine (which powers ACCESS) is intended for workgroup-size applications. It cannot scale up to a large volume of concurrent users and its security features are inadequate for storing large volumes of confidential or financial transactions.

It is possible to use ACCESS for prototyping your web-based applications. But once you have your prototype working, it is best to "up-size" to an industrial-strength client/server database system. Fortunately, ADO makes this easy.

28.3.5.1 Demonstration mode

In this section, we will revert to demonstration mode. I will show you how to create an ADO connection to a client/server database (the same SQL SERVER database that I used for the demonstration in Section 9.3.2).

28.3.5.2 Modifying the Connection object

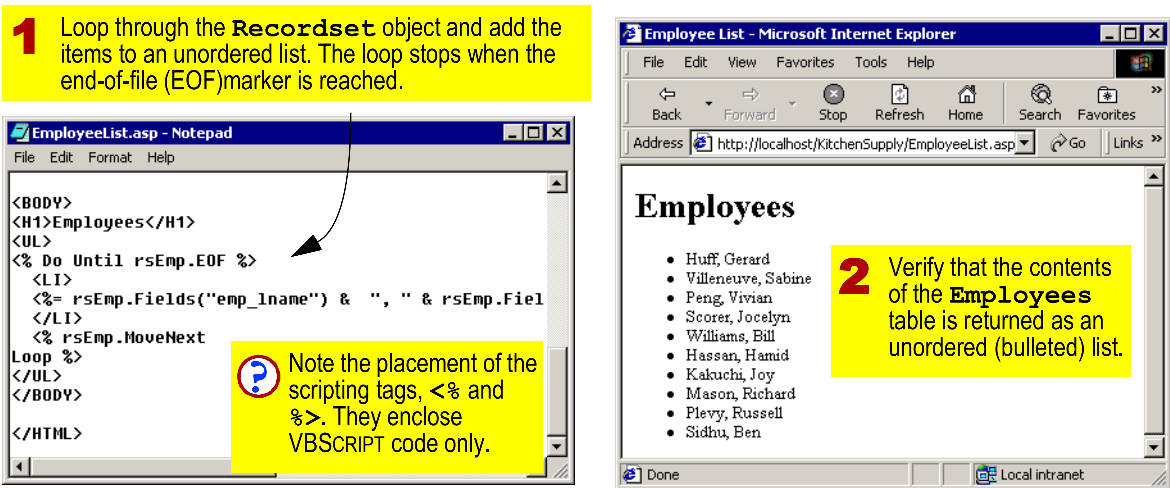
16 I start by modifying the properties of **objCon** so that it connects to a SQL SERVER database using the TCP/IP protocol. The changes are shown below.

```
NL  <%
NL
```

```
NL  <HTML>
NL  <HEAD>
NL  …
NL  Set objCon =
    Server.CreateObject("ADODB.
    Connection")
NL  With objCon
NL    .Provider = "sqloledb"
NL    .ConnectionString =
      "Server=brydon.bus.sfu.ca;
      UID=<username>;pwd=<password>"
NL    .Open
NL  End With
NL  …
NL  %>
NL  </HEAD>
NL  …
NL  </HTML>
```

Each type of data source requires a different **ConnectionString** property. For example, client/server databases require a server name, a user name, and a

FIGURE 28.5: Loop through the ***Recordset*** object to display the items return by the query.



password, whereas ACCESS only requires the location of a *.mdb file.

Once the properties of the `Connection` object have been updated, I am done. It is as simple as that. Since the details of the data source are entirely encapsulated within the `Connection`, no changes have to be made to the rest of my code.[1]

17 I verify that the new data source is working correctly. The results are identical to those shown in Figure 28.5.

28.3.6 Authentication

In Section 26.3.3.2, you created a "mock" authentication routine. A more useful authentication routine consists of two steps:

1. Search the `Customers` table for a user name that matches the value of `txtUserName` passed to the server by login form. If a match is not found, authorization fails.
2. If a matching user name is found, compare the value of `txtPassword` passed to the server to the user's password in the database. If they do not match, authorization fails; otherwise, authorization succeeds.

In this section, you will create an authorization routine that implements the above logic.

18 Open `Authorize.asp` for editing. Cut and paste the connection and command information for your ACCESS database from `EmployeeList.asp` into the top part of the `Authorize.asp` file.

19 To make the code a bit neater, modify the `objCon` section to use the `With… End With` syntax. The results are shown below:

```
NL <%
NL strUserName=Request.Form.Item("txtUserName")
NL strPassword=Request.Form.Item("txtPassword")
NL Set objCon =
   Server.CreateObject("ADODB.Connection")
NL With objCon
NL   .Provider =
   "Microsoft.Jet.OLEDB.4.0"
NL   .ConnectionString = "C:\Documents
   and Settings\brydon\My
   Documents\KitchenSupply\OrderEntryWeb.mdb"
NL   .Open
NL End With
```

[1] In practice, changing databases may be more complex. Since database systems have very different capabilities, problems occur when code that exploits features of one DBMS is executed against a different DBMS that does not support the feature.

```
NL  Set objCmd =
    Server.CreateObject("ADODB.Command"
    )
NL  With objCmd
NL    .CommandText = "SELECT * FROM
    Employees"
NL    .CommandType = 1
NL    Set .ActiveConnection = objCon
NL  End With
NL  If strUserName=strPassword Then
NL    Session("LI") = "True"
NL    Response.Redirect "Menu.asp"
NL  Else
NL    Session("LI") = "Fail"
NL    Response.Redirect "Login.asp"
NL  End if
NL  %>
```

20 Change the **`CommandText`** property to a SQL statement that returns the correct row from the **`Customers`** table:

```
NL  <%
NL  …
NL  Set objCmd =
    Server.CreateObject("ADODB.Command"
    )
NL  With objCmd
NL    .CommandText = "SELECT * FROM
    Customers WHERE UserName = '" &
    strUserName & "'"
NL    .CommandType = 1
NL    Set .ActiveConnection = objCon
NL  End With
NL  …
NL  %>
```

21 Cut and paste the **`Recordset`** code from **`EmployeeList.asp`** to **`Authorize.asp`**.

22 Change the name of the **`Recordset`** object from **`rsEmp`** to **`rsCust`** wherever **`rsEmp`** is used.

These changes are shown in Figure 28.6. If the username entered by the user is in the **`Customers`** table, a recordset consisting of a single record is returned to **`rsCust`**. Otherwise, an empty recordset is returned.

If you do not make the changes shown in Figure 28.6, the authorization routine will fail. Since debugging ASP scripts is difficult and time consuming, it is important to look very carefully at your code for typos and silly mistakes before publishing the files to the web server.

23 Modify the **`If… Then`** statement to recognize the three possible authorization outcomes. The code is shown below and in Figure 28.7.

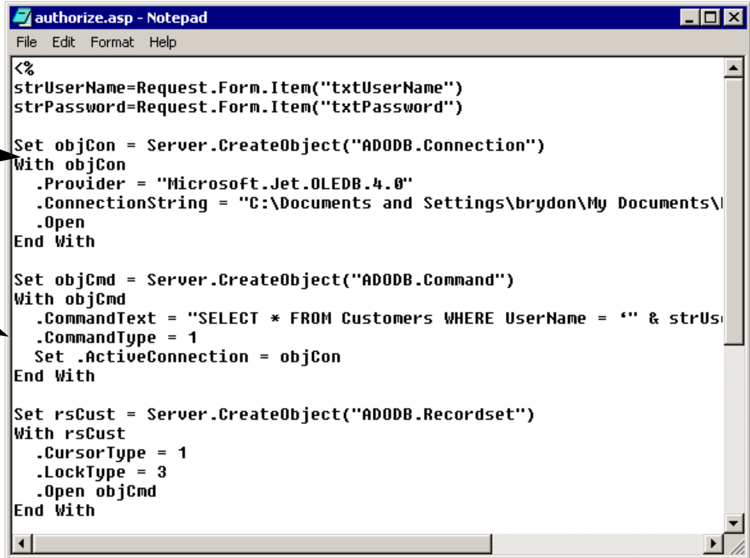
```
NL  <%
NL  …
NL  If rsCust.EOF Then
NL    Session("LI") = "FAIL_USER"
NL    Response.Redirect "Login.asp"
```

FIGURE 28.6: Create **`Connection`**, **`Command`**, and **`Recordset`** objects in **`Authorize.asp`** to support an authorization routine.

1 Cut and paste the connection and command information. Clean up the code a bit by using the **`With…`** **`End With`** construct.

3 Create a **`Recordset`** object. It should consist of one record if the user exists in **`Customers`** table.

? Since the SQL statement requires nested quotation marks, the single quotes are enclosed within double quotes to give a result of the form: **`“SELECT * FROM Customers WHERE UserName = ‘sammy’”`**.



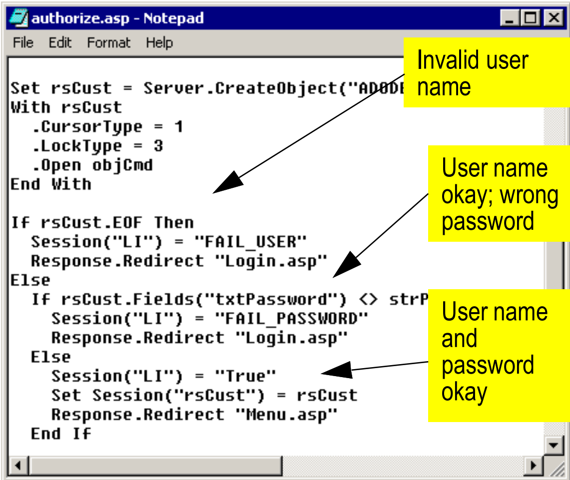
```
authorize.asp - Notepad
File  Edit  Format  Help
<%
strUserName=Request.Form.Item("txtUserName")
strPassword=Request.Form.Item("txtPassword")

Set objCon = Server.CreateObject("ADODB.Connection")
With objCon
   .Provider = "Microsoft.Jet.OLEDB.4.0"
   .ConnectionString = "C:\Documents and Settings\brydon\My Documents\
   .Open
End With

Set objCmd = Server.CreateObject("ADODB.Command")
With objCmd
   .CommandText = "SELECT * FROM Customers WHERE UserName = '" & strUs
   .CommandType = 1
   Set .ActiveConnection = objCon
End With

Set rsCust = Server.CreateObject("ADODB.Recordset")
With rsCust
   .CursorType = 1
   .LockType = 3
   .Open objCmd
End With
```

```
NL  Else
NL    If rsCust.Fields("Password") <>
      strPassword Then
NL      Session("LI") = "FAIL_PASSWORD"
NL      Response.Redirect "Login.asp"
NL    Else
NL      Session("LI") = "True"
NL      Response.Redirect "Menu.asp"
NL    End If
NL  End If
NL  %>
```

24 Modify **`Login.asp`** to provide meaningful error messages when the user enters the wrong username or password:

```
NL  <HTML>
NL  …
NL  <H3>
```

FIGURE 28.7: Implement the authorization logic using nested *If… Then* statements.




```
NL  <% If Session("LI")="FAIL_USER"
    Then %>
NL    User name incorrect: please try
      again
NL  <% ElseIf
    Session("LI")="FAIL_PASSWORD"
NL  Then %>
NL    Password incorrect: Please try
      again
NL  <% End If %>
NL  </H3>
NL  …
NL  </HTML>
```

25 Test the authorization procedure using different user names and passwords. When the valid user name/password combination “sammy”/“sam” is entered, you should be transferred to the menu page.

28.3.7 Updating data

To this point, you have only *displayed* data contained in **Recordset** objects. However, it is possible to use the **Recordset** object to make changes (append, delete, update) to the underlying data. In this section, you will create a “customer profile” form. Your on-line customers can use this form to view and edit certain information about themselves, including their billing address, contact information, and so on.

The **rsCust** recordset you used in the preceding section will be used for two purposes:

1. provide the initial values to show on the form (via the textboxes’ **VALUE** attribute), and
2. accept and save the new values of the **VALUE** attribute entered into the form by users.

To save the users’ changes to the data, you have to do a bit of plumbing. Specifically, you have to write a script that takes the values from the form (via the `Request.Form` collection) and use them to update the customer record. As a consequence, you will require two ASP files:

- `Customer.asp` — the visible form used to display customer data and permit users to make changes; and,
- `Customer_Process.asp` — an invisible script that writes any changes made by users to the database and then returns users to the updated `Customer.asp` page.

28.3.7.1 Reusing the customers recordset

The `rsCust` recordset already contains all the customer information that you need. However, recall that the scope of ASP variables is limited to the page on which the variables were created. In other words, once the user leaves the `Authorize.asp` page, the `rsCust` object is destroyed.

To get around this, you are have two options:

1. Create a new customer `Recordset` object whenever you need it. To create the correct SQL query each time, you need to pass some information about the customer (e.g., `CustID)` from page to page using a query string or a session variable.
2. Save the `Recordset` object as a session variable.

Neither option is always better than the other. Much depends on how often the `Recordset` in question will be used by other pages. However, Option 2 is certainly easier to implement, so we will stick with it.

The amount of memory required to store a simple value (such as `CustID`) is many times smaller than the amount of memory required to store a complex object such as an ADO `Recordset`. Since each user requires a unique session, the number of ADO objects that the server has to store in memory can become very large if there are many simultaneous users. Since we are more interested in broad concepts at this stage, we will ignore memory optimization. However, you should be aware of the scalability issues that arise when you save objects to session variables.

26 Add the following code to `Authorize.asp` to save a reference to the customers recordset in a session variable:

```
NL  <%
NL  …
NL  If rsCust.EOF Then
NL    Session("LI") = "FAIL_USER"
NL    Response.Redirect = "Login.asp"
NL  Else
```

```
NL   If rsCust.Fields("Password") <>
     strPassword Then
NL     Session("LI") = "FAIL_PASSWORD"
NL     Response.Redirect "Login.asp"
NL   Else
NL     Session("LI") = "True"
NL     Set Session("rsCust") = rsCust
NL     Response.Redirect "Menu.asp"
NL   End If
NL End If
NL %>
```

Whenever the user logs in successfully, a session-level pointer to the **Recordset** object is created that is available in all pages in the application for the duration of the session.

As soon as there are no variables pointing to the object, the object is destroyed and marked for garbage collection.

28.3.7.2 Creating a customer profile form

27 Rename the **Customer.html** file you created in Section 24.5 to **Customer.asp**.

28 In the header of **Customer.asp**, create a local variable which points to the session-level customer **Recordset** object. This is done to save some typing later.

```
NL <HTML>
NL <HEAD>
NL <% Set rsCust=Session("rsCust") %>
NL ...
NL </HEAD>
NL ...
NL </HTML>
```

29 Add form tags to the body of **Customer.asp**. As always, the **METHOD**, should be "Post". The **ACTION** value should be **Customer_Process.asp** (which you will create shortly).

In order to simplify the layout of multiple textboxes, it is worthwhile to put the form elements within an invisible table.

30 Nest **<TABLE>**… **</TABLE>** tags inside of the **<FORM>**… **</FORM>** tags.

31 Add the following code to create a row with two columns:

```
NL ...
NL <FORM ACTION="customer_process.asp"
   METHOD="post">
NL <TABLE cellPadding=1 cellSpacing=1
   width="75%">
NL <TR>
NL <TD>Name: <INPUT TYPE="Text"
   NAME="txtCustomerName" VALUE="<%=
   rsCust.Fields("CustName") %>"></TD>
NL <TD>Contact person: <INPUT
   TYPE="Text" NAME="txtContactPerson"
   VALUE="<%=
```

```
      rsCust.Fields("ContactPerson")
      %>"></TD>
NL    </TR>
NL    …
NL    </TABLE>
NL    </FORM>
NL    …
```

32 Add a second row containing textboxes with the following attribute/value pairs:

Textbox name	Field name
txtBillingAddresss	rsCust.BillingAddress
txtContactPhone	rsCust.ContactPhone

33 Add a third row with a **Submit** and **Reset** buttons:

```
NL    <TABLE>
NL    …
NL    <TR>
NL    <TD><INPUT TYPE="Submit"
      NAME="cmdSubmit" VALUE="Submit"></
      TD>
NL    <TD><INPUT TYPE="Reset"
      NAME="cmdReset" VALUE="Reset"></TD>
NL    </TR>
NL    </TABLE>
NL    </FORM>
NL    …
```

To test the form, use the menu buttons you created in Section 25.3.6 and enabled with scripts in Section 26.5.

34 Bring up the login page in your browser and log in as "sammy"/"sam".

35 Click the "Go" button for "Update Customer Profile." You should see a form similar to that shown in Figure 28.8.

28.3.7.3 Creating a form processing script

With the HTTP protocol, you do not have access to the form values until the submit button is pressed and the browser sends the web server an HTTP request. In ASP, the target of the HTTP request is generally a script-only page that executes and then transfers the user to a page with visible content.

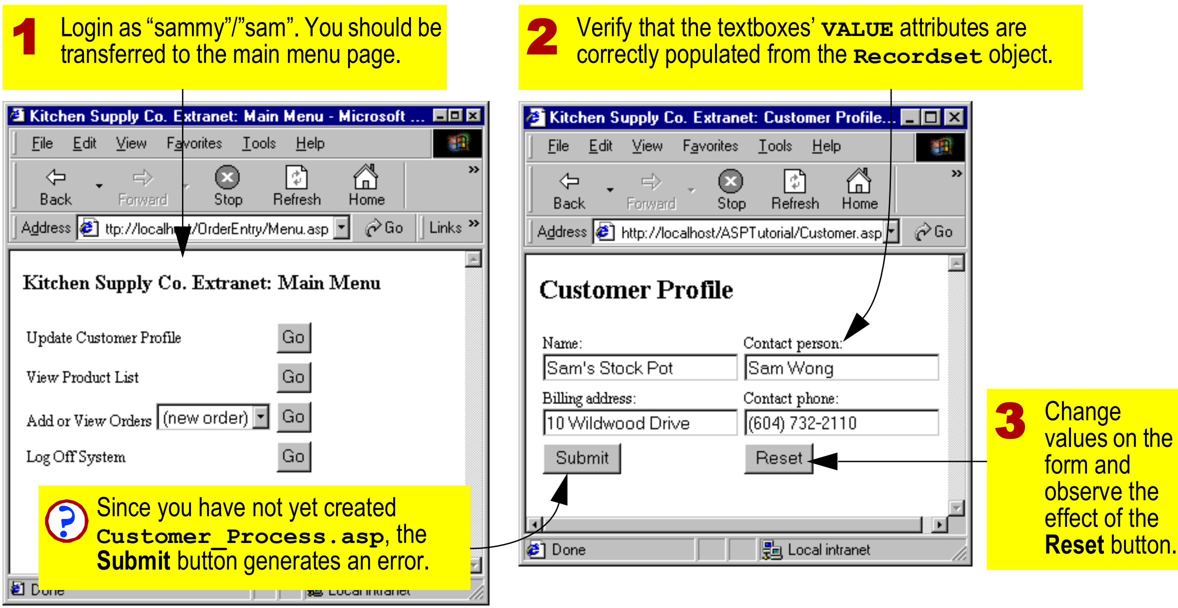
In this section, you will create a processing script to write changes made by the user to the database and then transfer the user back to the customer profile page.

36 Create a new file called `Customer_Process.asp`. Since the file will not contain any HTML, you do not need to add the core HTML tags.

37 Add the following code to transfer the values from the form to the table fields of the `Recordset` object:

```
NL    <%
NL    With Session("rsCust")
```

FIGURE 28.8: Test the customer profile form by logging in successfully and clicking on the hyperlink on the menu page.




```
NL      .Fields("CustName")=
    Request.Form("txtCustomerName")
NL      .Fields("BillingAddress")=
    Request.Form("txtBillingAddress")
NL      .Fields("ContactPerson")=
    Request.Form("txtContactPerson")
NL      .Fields("ContactPhone")=
    Request.Form("txtContactPhone")
NL      .Update
NL    End With
NL    Response.Redirect "Customer.asp"
NL    %>
```

If you forget the **`Update`** method, the changes to the **`Recordset`** object will not be saved to the database.

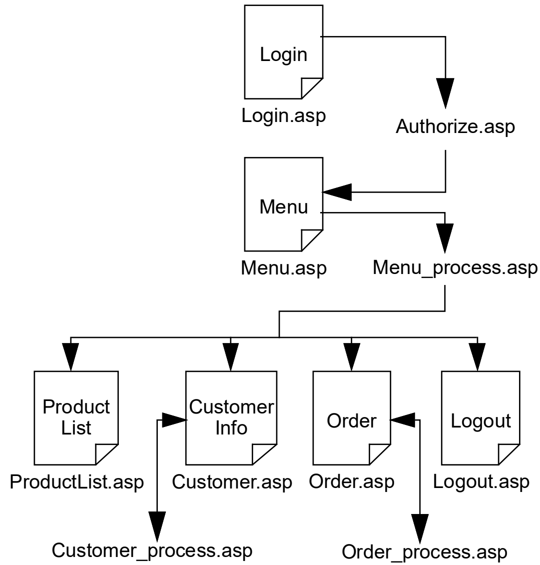
38 Change values in the customer profile form to ensure that they are saved to the database.

28.4 Discussion

The two-file approach you used to process database updates for customers is fairly typical in ASP application development. One file (**`Customer.asp`**) contains HTML form elements and allows the user to enter new values for customer fields. The second file (**`Customer_process.asp`**) contains executable script only and never shows up in the user's browser. Instead, the code to update the database is executed and the **`Response.Redirect`** method is used to transfer the user back to the (updated) customer profile.

Your web-enabled order entry system requires a number of invisible script-only pages to process the different types of information entered by your users. The relationship of the script-only pages to the content pages that you have already created is shown in Figure 28.9.

FIGURE 28.9: The updated structure of the web-based order entry system.



28.5 Application to the project

28.5.1 Touch-ups

39 Ensure you have implemented the login and customer profiles pages described in this lesson.

40 If you wish, you may add more information to the customer profile form, although it is certainly not a requirement (if you know how to do four HTML textboxes, you know how to do a thousand).

41 To save time when testing your application in subsequent lessons, set the default values for the user name and password textboxes on your login form to appropriate values. For example, set the `VALUE` attribute of `txtUserName` to “sam”. In this way, you can login with one click and do not have to keep typing “sam”/ “sammy”.

Obviously, you should remove the `VALUE` attributes from both textboxes on your login page when you put the application into production.

28.5.2 Persistence pays

As Figure 28.9 shows, your application requires a large number of ASP pages to function properly. Many of these pages require access to the database either to:

- extract and display data (e.g., `Products.asp`),
- search the database (e.g., `Authorize.asp`), or
- update the database (`Customer_process.asp`).

Rather than create a separate ADO connection for each page, it is much simpler to create a session-level reference to the `Connection` object when it is first created.

42 Add the following code to your `Authorize.asp` file. The code should only execute when user authentication is successful.

```
NL  Set Session("objCon") = objCon
```

28.5.3 Creating a dynamic list of products

43 Use an ADO recordset to complete the product list page you started to populate manually in Section 24.3.5. This task requires the same skills as the employee list in Section 28.3.4.

In ADO, you can use shortcuts to create recordsets. Specifically, you can create a recordset without first creating a command object or setting the recordsets properties. The following code creates a recordset containing a list of products sorted by the `ProductID` field:

```
NL  Set rsProducts =
    Server.CreateObject
    ("ADODB.Recordset")
```

NL
```
rsProducts.Open "SELECT * FROM
Products ORDER BY ProductID",
Session("objCon"), 0, 1
```

Three arguments are applied to the recordset's `Open` method:

1. **Source** (`SELECT * …`)— source can be an ADO `Command` object, the name of a table, or (as in this case) an SQL statement.
2. **Connection** (`Session("objCon")`)— if the source is an ADO `Command` object (with its `ActiveConnection` property set), then the connection argument is optional. However, in this case, no `Command` object is used and thus a `Connection` object must be supplied.
3. **Cursor type** (`0`)— since the product list is read-only, the simplest type of recordset ("forward only") is used. A forward-only recordset is a read-only snap shot in which the only permitted method is `MoveNext` or `MoveLast` (`MovePrev` and `MoveFirst` violate the forward-only constraint). Use of this type of recordset greatly reduces the amount of server resources required to show the product list.
4. **Lock type** (`1`)— the lock type is set to read-only. Given that the cursor type is read-only already, this property does not need to be set.

29.1 Introduction: Modularity using COM objects

ORDEROBJECTS is a COM object that can be used to simplify the construction of certain parts of your web-based order entry system. What is a COM Object? COM stands for COMPONENT OBJECT MODEL and is a MICROSOFT-initiated standard that specifies how WINDOWS software components written by different people using different languages can work together.

Over the years, COM has undergone many changes in name and functionality. Previous incarnations include OLE (OBJECT EMBEDDING AND LINKING) and ACTIVEX. There is also DCOM (Distributed COM) for having software components on different machines interact over networks and COM+, which is an upgraded version of COM. Confused yet? You should be. See MICROSOFT's web site for more recent news about the COM family of technologies.

29.1.1 Shared libraries

The COM standard allows programmers to create specialized routines using tools such as VISUAL C++, VISUAL BASIC, or BORLAND'S DELPHI and store the routines in dynamic link library (DLL) files. Code stored in an COM-compliant DLL can be accessed by any program that supports the COM standard (i.e., most major WINDOWS programs including OFFICE applications and ACTIVE SERVER PAGES). COM is immensely powerful because it means that anyone who can write a few lines of code can create routines to extend the functionality of commercial software.

To illustrate, assume that you want to use a combo box to allow users to select from a list of countries. However, rather than just show the name of the country, you want to show the name plus a small image of each country's flag. Assuming you know how, you could use the COM standard and a language like C++ to create a flag-enabled combo box component. Once the flag-enabled combo box is created and installed on your machine, it can be used like any other interface control in applications such as ACCESS. In addition, you can make the component available to others in your organization or even sell it.

As it turns out, there is a healthy third-party market for specialized COM components (e.g., **`www.componentsource.com`**).

29.1.2 The OrderObjects component

It is important to point out that COM is not limited to interface elements. For example, the ADO object model you used in Lesson 28 is provided by a COM component that snaps into Windows. Unlike a flag-enabled combo box, ADO is invisible. It works in the background to provide services like linking to databases.

The OrderObjects component works in a similar manner (albeit on a much smaller scale than ADO). The component was written in Visual Basic and compiled to a COM DLL. Naturally, OrderObjects is not a commercial-grade COM component. Instead, it is intended to give you some exposure to component-based development and simplify the completion of your web-based order entry system.

In order for your ASP pages to use the functionality provided by the OrderObjects component, the component must be installed on the web server. The installation process is discussed in Section 29.3.1.

29.1.3 OrderObjects versus ADO

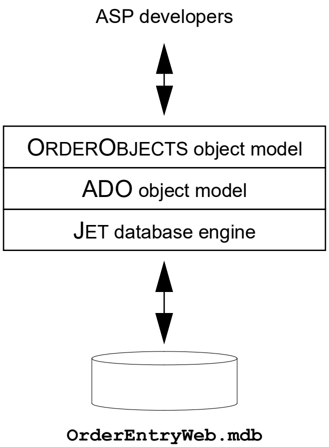
Recall that the ADO object model is meant to insulate developers from the complexity of dealing with many database systems (and keep in mind that database systems are meant to insulate developers from the complexities of physical data storage). The `Connection`, `Command`, and `Recordset` objects that you created in Lesson 28 gave you easy access to the contents of your Access database file. And with a few minor modifications, it was shown how the same code could be used to access a SQL Server database.

Continuing on this theme, the OrderObjects object model is meant to insulate developers (i.e., you) from the complexities of ADO. The component allows you to create two classes of objects: `Order` and `OrderDetails`. Each object encapsulates data from your database and provides methods for retrieving and updating information about customer orders.

For example, the `Order.ProcessOrder` method saves the details of an order from the HTML form to the database and updates inventory levels to reflect quantities earmarked for shipping. In addition, the `ProcessOrder` method contains business rules such as “You cannot ship what you do not have.” In other words, the logic for determining the quantity to ship is already defined within the `Order` object. All you have to do is set up an HTML order form and call the appropriate method to process the order.

The relationship between the ORDEROBJECTS layer, ADO, and the JET database is shown in Figure 29.1.

FIGURE 29.1: Insulating developers from complexity.



29.2 Learning objectives

- understand how business rules can be embedded in compiled object libraries
- register a COM component with the operating system (if required)
- use the ORDEROBJECTS component to simplify the creation of an on-line order form
- understand how third-party COM components make your life as a developer easier

29.3 Exercises

29.3.1 Installing the ORDEROBJECTS component

In my courses, I set up a web server for my students and install the ORDEROBJECTS component on the server for them. If you have someone doing this for you, you may skip ahead to Section 29.3.3.

In this section, you will put the ORDEROBJECTS component DLL file on your web server and notify WINDOWS that the component exists. Two different methods are provided:

1. direct registration of component with the operation system, and
2. installation via a generic setup routine.

The outcome is the same in both cases, except that the installation routine does more than simply register the component: it makes sure you have a relatively recent version of ADO installed on your machine and checks other dependencies.

29.3.1.1 Direct registration

When you use the `Server.CreateObject()` method in your VBSCRIPT code, you are asking the server's operating system to create an instance of the specified object type. For example, in Section 28.3.1 you created an ADO `Connection` object using: `CreateObject("ADODB.Connection")`.

You may ask yourself: "How does the operating system know how to create an ADO `Connection` object?" The answer is that the operating system's registry (which is like a database) contains a pointer to the DLL file that contains the ADO functionality. The trick is to inform the registry of the location of the DLL file in the first place.

1. Copy the `OrderObjects.dll` file (from the project package) to a suitable location on your web server's hard disk (e.g., `C:\Documents and Settings\brydon\My Documents\KitchenSupply\`).

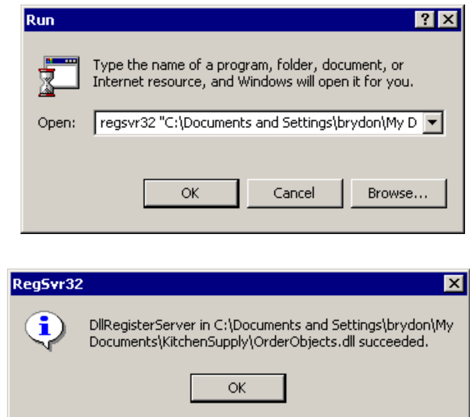
2. Select **Start** → **Run** from the WINDOWS task bar and use the `RegSvr32` command to add the location of the component to the registry:

NL
```
Regsvr32 "C:\Documents and Settings\brydon\My Documents\KitchenSupply\OrderObjects.dll"
```

If the name of the path to the DLL file contains spaces, you must enclose the file name in quotation marks (as shown above).

If you have typed in the path correctly, you should get the message box shown in Figure 29.2.

FIGURE 29.2: Use ***RegSvr32*** to register the ORDEROBJECTS component.



To unregister the component, reverse the process using `RegSvr32`'s "-u" switch:

NL
```
Regsvr32 "C:\Documents and
Settings\brydon\My
```

`Documents\KitchenSupply\OrderObjects.dll" -u`

Once the component is unregistered, you can safely delete the DLL from the hard drive and you are back to where you started.

29.3.1.2 Installation via the installer

One problem with direct installation is that the code in **`OrderObjects.dll`** makes certain assumptions about the versions of ADO and ASP installed on the web server. If the server is using earlier versions of these components, then the ORDEROBJECTS component may not work correctly or work at all (issues surrounding DLL versions and component installation are discussed briefly in Section 29.4.2).

The installation program in the project package copies the DLL file to the directory you specify and registers the component. It also checks the installed versions of ADO and ASP and updates them as required (hence the size of the installation files). With the administrative work done, you and all other developers on your web server can use the ORDEROBJECTS component.

The utility used to create the install routine for ORDEROBJECTS is bundled with MICROSOFT VISUAL STUDIO 6.0. In my experience, it is well-behaved—that is, it leaves newer versions of components intact and provides an uninstall option through **Control Panel → Add/Remove Software**.

3 Find the setup directory for ORDEROBJECTS in the project package and copy the three files (**`Setup.exe`**, **`OrderObjects.cab`**, and **`Setup.lst`**) to a folder on the web server.

Since you will delete the setup files once the installation is complete, it does not matter which folder you store the files in.

4 Double-click the **`Setup.exe`** file.

5 Specify an installation directory (e.g., **`C:\Program Files\OrderObjects`**) as shown in Figure 29.3.

When the installation routine completes, you should get a message box similar to the one shown in Figure 29.4. If you look in the installation directory (e.g., **`C:\Program Files\OrderObjects`**), you will see the **`OrderObjects.dll`** file and an uninstall log file.

To uninstall the ORDEROBJECTS component, do not simply delete the **`OrderObjects`** folder. Doing so deletes the install log and renders the **Control Panel → Add/Remove Software** feature useless. Having said this, it is important to keep in mind that the uninstall program does nothing

FIGURE 29.3: Use ORDEROBJECTS installer to install the component.

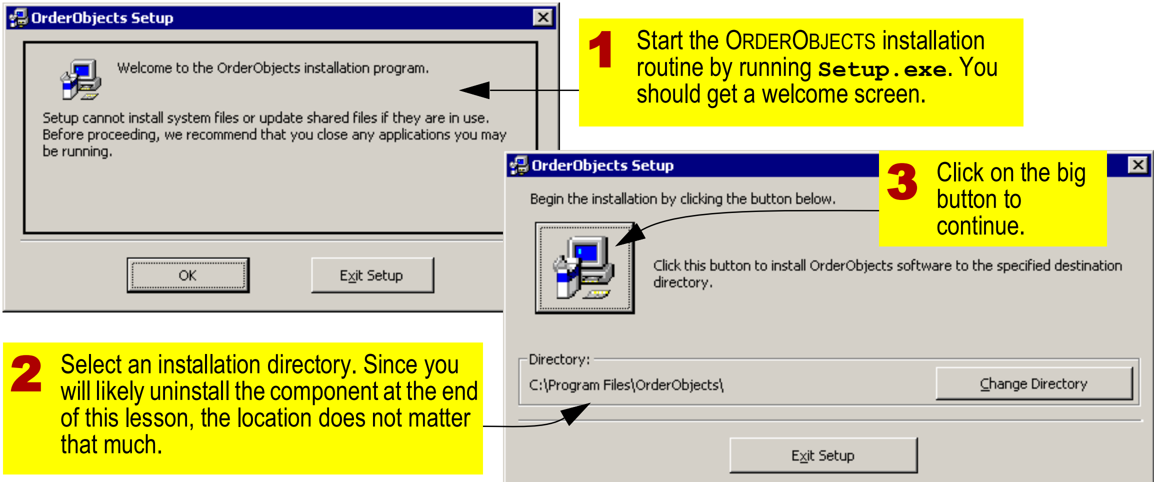
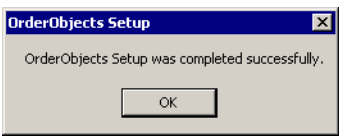


FIGURE 29.4: The installation routine completes successfully.



magical: it simply removes the registry entries for the ORDEROBJECTS objects and deletes the DLL file.

6 Return to the temporary folder containing the three installation files (`Setup.exe`, `OrderObjects.cab`, and `Setup.lst`) and delete them.

29.3.2 Reading the component's documentation

Before using any third-party component, you should read the documentation provided with the component and make sure you understand what objects, methods, and properties are available for use. Unfortunately, the documentation supplied with third-party controls tends to be sparse and the brief overview of ORDEROBJECTS included as Section 29.4.1 conforms to this generalization.

7 Scan Section 29.4.1 and ensure you understand the basic architecture of the Order and OrderDetails objects and how they relate to one-another.

8 Make a mental note of the methods and properties that are exposed by each object.

29.3.3 Creating and initializing the object

In this section, you will create an instance of the `Order` class (i.e., you are going to create an `Order` object). A good place to create and initialize the object is in the `Authorize.asp` file. In this way, the object is created and initialized only if the user logs in successfully.

You will start by creating a new `Order` object called by using the `Server.CreateObject(`*<class name>*`)` method. Then, you will "initialize" the object by calling the `objOrder.Initialize` method and passing it two arguments:

- a valid ADO `Connection` object to an `OrderEntryWeb.mdb` database; and,
- the `CustID` value of the customer.

Passing an existing connection to the object means that you do not have to configure the `Order` object for a particular database—everything the object needs to connect to the data is already encapsulated in `objCon`. The `CustID` is needed so that the `Order` object knows which records from the `Orders` table to retrieve (i.e., those orders belonging to the customer that is currently logged in).

9 Add the following to your `Authorize.asp` file:

```
NL  <%
NL  If rsCust.EOF Then
NL    …
NL  Else
NL    If rsCust.Fields("txtPassword")
    <> strPassword Then
NL      Session("LI") = "FAIL_PASSWORD"
NL      Response.Redirect "Login.asp"
NL    Else
NL      Session("LI") = "True"
NL      Set Session("rsCust") = rsCust
```

```
NL      Set Session("objOrder") =
    Server.CreateObject
        ("OrderObjects.Order")
NL      Session("objOrder").Initialize
        objCon, rsCust.Fields("CustID")
NL      Response.Redirect "Menu.asp"
NL    End If
NL  End If
NL  %>
```

A reference to the **Order** object is assigned to a session-level variable. In this way, the **Order** object can be used on other pages of the application.

29.3.4 Selecting an order from the menu

The order form in this application has two basic functions:

1. allow the user to create a new order, and
2. allow the user to view or change an existing order.

The combo box on the menu page is used so that the user can indicate whether she wants to create a new order or show an existing order.

In this section, you will modify the **cboOrderID** combo box you started in Section 25.3.6.2 and create a script to process the user's selection.

29.3.4.1 Retrieving a list of existing orders

10 Open **Menu.asp** for editing and create a local reference to the **Order** object in the header of the document:

```
NL  <%
NL  If Session("LI") <> "True" Then
NL    Response.Redirect "Login.asp"
NL  End If
NL  %>
NL  <HTML>
NL  <HEAD>
NL  <TITLE>Kitchen Supply Co. Extranet:
    Main Menu</TITLE>
NL  <% Set objOrder =
    Session("objOrder") %>
NL  </HEAD>
NL  …
```

Creating a local reference to a session-level object allows you to use the object without continually typing "Session(…)".

The first option in the combo box is for a new order (**OrderID** = 0) and should be left intact. However, the remaining rows in the combo box should be populated dynamically based on the existing orders placed by the customer. The special order listing features of the **Order** object can be used for this purpose.

29.3.4.2 Looping through the orders

The procedure to loop though the list of orders using the **`Order`** object is slightly different than the procedure you used to loop through an ADO recordset in Section 28.3.4:

- The **`objOrder.MoveFirst`** and **`objOrder.MoveNext`** methods return True if they are successful and False if the list is empty or the end of the list is encountered. You can use the value returned by the **`MoveFirst`**/**`MoveNext`** methods instead of checking the recordset's **`EOF`** property after each move.
- Since the desired values are properties of the object, the ***`<Recordset>`*****`.Fields("`*****`<Field name>`*****`")`** notation does not have to be used. Instead, the more compact and familiar ***`<object>`*****`.`*****`<property>`*** notation can be used (e.g., **`objOrder.OrderID`**).
- A special property called **`OrderName`** is provided to help users identify a particular order by **`OrderID`** and date. **`OrderName`** is much like a calculated field, except that it is calculated within the **`Order`** object.

11 Modify your **`cboOrderID`** combo box so that it is populated dynamically when the page is created:

```
NL  …
NL  <TD>Add or View Orders
NL  <SELECT NAME="cboOrderID">
NL  <OPTION VALUE=0>(new order)</
    OPTION>
NL  <% If objOrder.MoveFirst Then
NL    Do %>
NL      <OPTION VALUE="<%=
    objOrder.OrderID %>">
        <%= objOrder.OrderName %></
    OPTION>
NL    <% Loop While objOrder.MoveNext
NL  End If %>
NL  </SELECT></TD>
NL  …
```

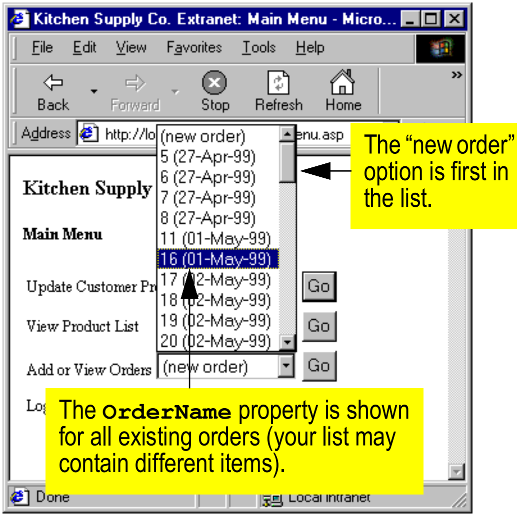
Note that the **`VALUE`** attribute (the part that is passed to the server) for each option is the **`OrderID`** but the value shown in the combo box is **`OrderName`**.

12 Test the combo box, as shown in Figure 29.5.

29.3.4.3 Finding an existing order

Once the user has selected an order from the combo box on the menu page and pressed the corresponding **Go** button, the **`GetOrder`** method of the **`Order`** object can be used to locate the order in the database. The syntax of the **`GetOrder`** method is: **`objOrder.GetOrder`** ***`<OrderID>`***. The **`OrderID`** of the order that the user wishes to view is passed to the server in

FIGURE 29.5: Use the **`Order`** object to dynamically populate the combo box.



the HTTP request as `cboOrderID`. Thus, **`Request.Form("cboOrderID")`** can be used as the argument for the **`GetOrder`** method.

13 Open the **`Menu_Process.asp`** file you created in Section 26.5 for editing.

14 Add the following code to find the order specified by the user:

```
NL  <%
NL  …
NL  ElseIf
    Request.Form("cmdOrder")="Go" Then
NL    strRedirect="Order.asp"
NL    If Request.Form("cboOrderID")=0
    Then
NL       'create new order
NL    Else
NL       'locate an existing order
NL       Session("objOrder").GetOrder
         Request.Form("cboOrderID")
NL    End If
NL  …
NL  End If
NL  Response.Redirect strRedirect
NL  %>
```

To this point, you have only entered a comment line for the situation in which **`OrderID`** = 0. You will complete the "new order" branch momentarily.

29.3.4.4 Testing the GetOrder method

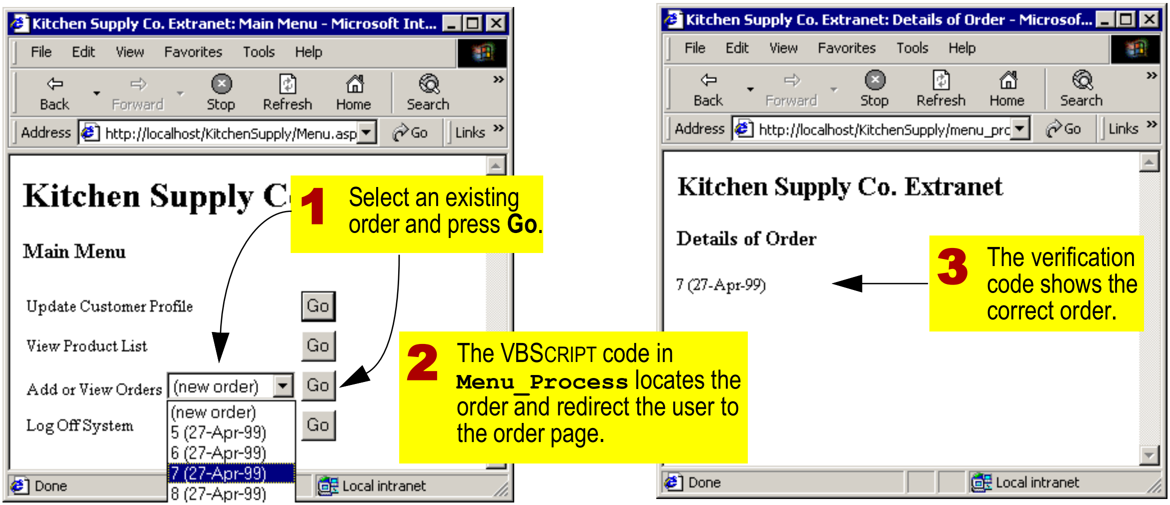
15 Open **`Order.asp`** for editing and create a local reference to the session-level **`Order`** object in the header section of the document.

16 In the body of the document, add the following verification code to ensure that the correct order is being retrieved:

NL `<%= objOrder.OrderName %>`

17 Test the menu, as shown in Figure 29.6. When you are satisfied that the `GetOrder` method is working, delete the verification code.

FIGURE 29.6: Verify that the correct order is being located before the order form is shown.



29.3.5 Displaying a customer order

The `Order` object sitting on the web server expects to interact with the HTML order form in a particular way. Specifically, when creating and modifying an order, *all* the products available for ordering are shown on the order form. To place an order, the user does the following:

1. Change the `QtyOrdered` value for desired items from zero to some other value.

2. Press the **Process Order** button when the desired quantities for all products have been entered.

When the order is processed by the `Order.ProcessOrder` method, all items with `QtyOrdered` = 0 are dropped from the order. Thus, the code encapsulated inside the `Order` object takes care of ensuring that only non-zero order details are stored in the `OrderDetails` table. The rationale for this particular ordering interface is discussed in more detail in Section 29.4.3.

29.3.5.1 Creating an order header

18 Create a form in the body of `Order.asp` and set its `ACTION` attribute to `Order_Process.asp`.

Like the ACCESS order form you created in Lesson 14, the web-based order form should have an order header (showing information about the order) and an order detail section (showing the items in the order).

19 Create a table to simplify the layout of the order header information.

20 Add textboxes for each of the order properties that you wish to show. A list of properties available from the `Order` object is provided in Section 29.4.1.1. An example of the type of HTML and ASP code used to create a basic header is shown in Figure 29.7.

The `SIZE` attribute can be used within an `INPUT` tag to control the size of the textbox.

29.3.5.2 Processed and unprocessed order

The appearance and behavior of the order form depends on whether the order has been processed. If the order has been processed, the customer cannot change the order in any way.

To indicate read-only controls visually, the HTML `DISABLED` attribute can be used. In most browsers, a disabled HTML control cannot receive the focus and is greyed-out. For example, to disable the `txtOrderID` textbox, use an `<INPUT>` tag similar to the following:

```
NL <INPUT NAME=txtOrderID VALUE="<%=
   objOrder.OrderID %>" SIZE=5
   DISABLED>
```

Although support for the `DISABLED` attribute in browsers is spotty (see Section 29.4.4), we will use it here to keep things simple.

21 Lock the form controls that the customer is not permitted to change (e.g., `txtOrderID`).

FIGURE 29.7: Implement an order header using HTML form elements and a table for formatting.

order.asp - Notepad

```
<TABLE border=1 cellPadding=1 cellSpacing=1 width="80%">
  <TR>
    <TD><STRONG>Order ID</STRONG></TD>
    <TD><INPUT NAME=txtOrderID VALUE="<%= objOrder.OrderID %>"
      SIZE=5 DISABLED></TD>
    <TD><STRONG>Order Date</STRONG></TD>
    <TD><INPUT NAME=txtOrderDate VALUE="<%= objOrder.OrderDate %>"
      SIZE=5></TD>
    <TD><STRONG>Processed?</STRONG></TD>
    <TD><INPUT NAME=chkProcessed TYPE=checkbox DISABLED
      <% If objOrder.Processed Then %> checked <% End If %>></TD></TR>
    <% If Not objOrder.Processed Then %>
      <TR>
        <TD colSpan=6>
        <P align=right><INPUT NAME=cmdSubmit TYPE=submit
          VALUE="Process Order"></P></TD></TR>
      <TR>
        <TD colSpan=6>
        <P align=right><INPUT NAME=cmdReset TYPE=reset
          VALUE="Abandon Changes"></P></TD>
      </TR>
    <% End If %>
</TABLE>
<P>
```

1 Create a table to format the header.

2 Use the `Order.Processed` property to determine whether to set the checkbox's `CHECKED` attribute.

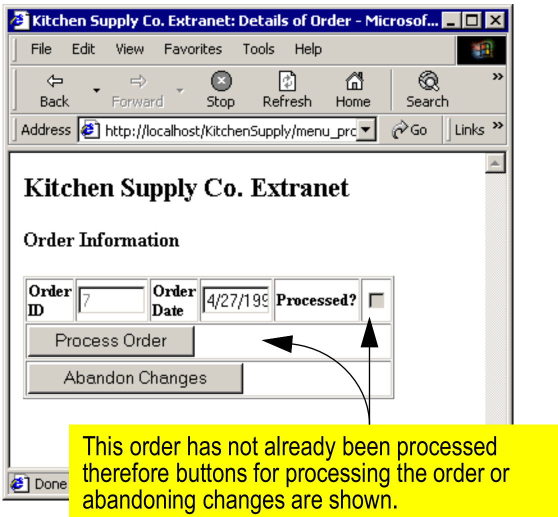
3 Use the `Order.Processed` property to determine whether to show buttons for processing the order.

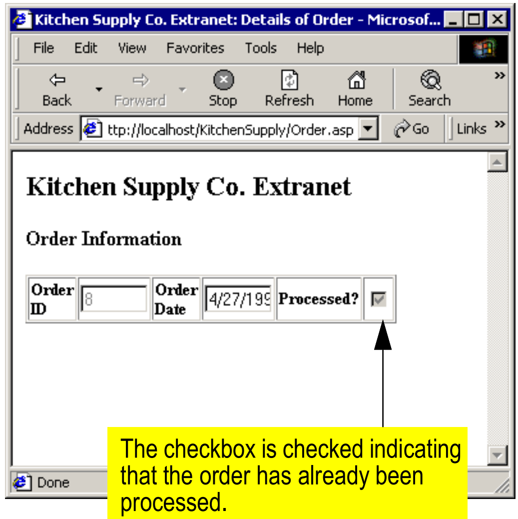
22 Test the order header. The differences between a processed order (e.g., `OrderID` = 7) and an unprocessed order (e.g., `OrderID` = 8) are shown in Figure 29.8.

29.3.5.3 Displaying the order details

The `Order.GetOrder` method not only retrieves the correct properties for the selected order, it *synchronizes* its internal `OrderDetails` object. In other words, when the `GetOrder` method is executed, the `OrderDetails` object is updated so that it contains the order details of the selected order only.

FIGURE 29.8: The appearance of the order header depends on whether the order in question has already been processed.





Accessing the individual order details using the **OrderDetails** object is similar to accessing the items in an ADO **Recordset** object: you start at the top of the list and iterate through the list until you reach its end.

The basic steps required to list an order's order details are the following:

1. Create a local reference to the **OrderDetails** object. To do this, you need to know two things:
   a) The **Order** object encapsulates a synchronized **OrderDetails** object.

b) You can access the `OrderDetails` object by using an **accessor method** called `OrderDetails` provided by the `Orders` object.

Thus, the syntax required to create a local reference called `objDetails` to the `OrderDetails` object inside of `objOrder` is **`Set objDetails = objOrder.OrderDetails`**

2. Use the `MoveFirst` method to move to the first order detail and to ensure that at least one order detail exists.
3. Use the `MoveNext` method to step through the items in the list and recognize the end of the list.

In this section, you will create a table to format the details of the order and use the `OrderDetails` object to dynamically populate the table.

23 Create a second table on the order form under the order header.

24 Use the table header tags, `<TH>` and `</TH>`, to create a row of column labels, as shown in the code below:

```
NL  …
NL  <TABLE …>
NL    order header
NL  </TABLE>
NL  <% If objDetails.MoveFirst Then %>
NL  <TABLE border=1 cellPadding=1
    cellSpacing=1 width="80%">
NL  <TR>
NL    <TH>ProductID</TH>
NL    <TH>Description</TH>
NL    <TH>Unit</TH>
NL    <TH><P align=right>Price</P></TH>
NL    <TH><P align=right><% If
    objOrder.Processed Then %>
NL      Qty<BR>Shipped
NL    <% Else %>
NL      Qty On<BR>Hand
NL    <% End If %></P></TH>
NL    <TH><P
    align=right>Qty<BR>Ordered</
    STRONG></P></TH>
NL    <TH><P
    align=right>Extended<BR>Price</
    STRONG></P></TH>
NL  </TR>
NL  </TABLE>
NL  <% End If %>
```

The value of `objOrder.Processed` is used to control the headings shown for the details part of the order. Naturally, if the order has already been processed, the current quantity on hand for each product in the order is irrelevant and should not be shown. Similarly, if the order has not been processed, the quantity shipped has not yet been determined and should not be shown.

29.3.5.4 Multiple QtyOrdered values

When the user presses the **Process Order** button, an HTTP request containing the **txtQtyOrdered** values entered by the user is sent to the web server. However, a single form field cannot contain the quantity ordered information for multiple products. As a consequence, the order form must generate unique **txtQtyOrdered** = *value* pairs for each product in the order.

The **Order** object requires that a special convention be used for the name of the "quantity ordered" textbox on your order form. Specifically, the name of the textbox must be **txtQtyOrdered** followed by an underscore and the **ProductID** of the item being ordered. Thus, the field/value pair "**txtQtyOrdered_51 5012**" = 12 indicates that the quantity ordered for product 51 5012 should be set to 12 units.

25 Use VBScript to construct the name of the **QtyOrdered** textboxes dynamically:

```
NL  <INPUT TYPE="Text" NAME=
    "<%= "txtQtyOrdered_" &
    objDetails.ProductID %>">
```

The **Order** object's **ProcessOrder** method contains all the logic required to extract the **ProductID** from the HTTP field/value pair and write the changes to the database.

26 Add code to iterate through the items in the **OrderDetails** object. An example is shown in below:

```
NL  …
NL  <% If objDetails.MoveFirst Then %>
NL  <TABLE border=1 cellPadding=1
    cellSpacing=1 width="80%">
NL  <TR>
NL    …
NL  </TR>
NL  <% Do %>
NL  <TR>
NL    <TD><INPUT disabled
    name="txtProductID" value="<%=
    objDetails.ProductID %>" size=10></
    TD>
NL    <TD><INPUT disabled
    name="txtDescription" value="<%=
    objDetails.Description %>"
    size=15></TD>
NL    <TD><INPUT disabled
    name="txtUnit" value="<%=
    objDetails.Unit %>" size=3></TD>
NL    <TD><INPUT disabled
    name="txtPrice" value="<%=
    FormatCurrency(objDetails.ActualPri
    ce) %>" size=5></TD>
NL    <TD><INPUT disabled size=5
NL    <% If objOrder.Processed Then %>
NL      name="txtQtyShipped" value="<%=
    objDetails.QtyShipped %>"
NL    <% Else %>
```

```
NL      name="txtQtyOnHand" value="<%=
    objDetails.QtyOnHand %>"
NL    <% End If %>></TD>
NL    <TD><INPUT <% If
    objOrder.Processed Then %> disabled
    <% End If %> name="<%=
    "txtQtyOrdered_" &
    objDetails.ProductID %>" value="<%=
    objDetails.QtyOrdered %>" size=5></
    TD>
NL    <TD><INPUT disabled
    name="txtExtendedPrice" value="<%=
    FormatCurrency(objDetails.ExtendedP
    rice) %>" size=5></TD></TR>
NL  <% Loop While objDetails.MoveNext
    %>
NL  </TABLE>
NL  <% End If %>
```

The `objOrder.Processed` property is also used to control whether certain textboxes are enabled. If the order has already been processed, *all* the fields on the form should be disabled. If the order has not been processed, then the user should be able to change the quantity ordered for each product.

29.3.5.5 Formatting issues

Note that the price values are not formatted as currency by default in HTML. To fix this problem, use VBSCRIPT's `FormatCurrency()` function.

27 Ensure you have used VBSCRIPT's built-in `FormatCurrency()` function to displace monetary values:

```
NL  <TD><%= FormatCurrency
     (objDetails.UnitPrice) %></TD>
```

28 Test the list, as shown in Figure 29.9.

29.3.6 Processing the order

To process the order form, you must pass the entire ASP `Request` object (which contains the HTTP field/value pairs) to the `order` object's `ProcessOrder` method.

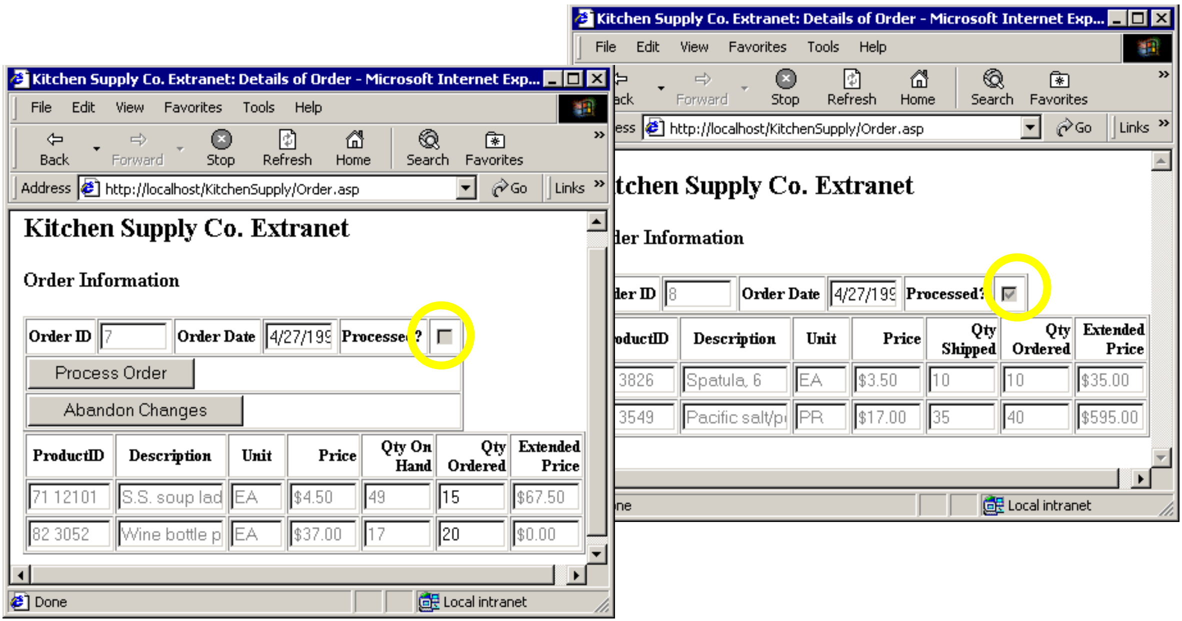
29 Add a **Process order** button to the order form. It should send the HTTP request to `Order_Process.asp`.

30 Create a new ASP file called `Order_Process.asp`

31 Add the following code to the file:

```
NL  <%
NL  Session("objOrder").ProcessOrder
    Request
NL  Response.Redirect "Order.asp"
NL  %>
```

FIGURE 29.9: The appearance of the completed order form depends on whether the order in question has been processed.



Note that the **ProcessOrder** method requires that a reference to the built-in ASP **Request** object be passed as a parameter. Also, recall from Section 18.4.3 that the parameter should *not* be enclosed in brackets since **ProcessOrder** does not return a value.

Processing an order is as easy as that. The **Order** object does all the work required to process the order and permits you, as the application designer, to focus on interface and functionality issues.

29.3.7 Creating a new order

The procedure for creating a new order is very similar to that for viewing or updating an existing order. The only difference is that rather than finding an existing order, the **Orders** object must

- create a new order in the **Orders** table, and
- create a default set of order detail records for the order.

As discussed in Section 29.4.3, the default set of order details used in this system is simply the set of all products in the **Products** table.

32 Add the following code to the "new order" branch in **Menu_Process.asp**:

```
NL  'create a new order
NL  Session("objOrder").AddOrder
```

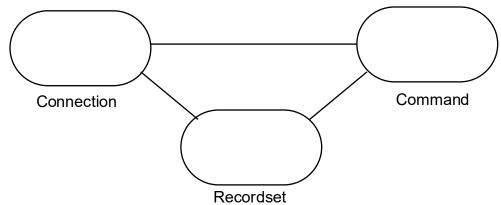
29.4 Discussion

29.4.1 The OrderObjects object model

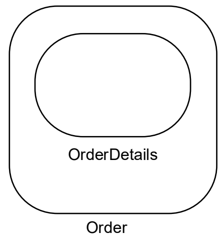
Unlike the ADO object model, the ORDEROBJECTS object model is hierarchical—the **orderDetails** object is completely contained within the **Order** object. A comparison of the two object models is shown in Figure 29.10. In the following sections, a brief overview of both the objects in the **OrderObjects** model is provided and the important properties and methods of each are listed.

FIGURE 29.10: A comparison of the ADO and ORDEROBJECTS object models

The ADO object model



The OrderObjects object model



29.4.1.1 Order

The **Order** object can be used to access any order placed by the customer or to create a new order in the customer's name. The object has the following public properties:

Property	Data type
OrderID	Long
OrderName[a]	String
CustomerID	Long
OrderDate	Date
Processed	Boolean
OrderDetails	reference to OrderDetail object

a. **OrderName** is simply the concatenation of **OrderID** and **OrderDate** in (DD-MMM-YY) format.

Before accessing the properties of the **Order** object, the **GetOrder** method must be used locate the desired order. The following methods are provided by the **Order** object to simplify retrieval and processing of orders:

- **Initialize(objCon as Connection, lngCustID as Long)** —returns a Boolean value: The **Order** object must be passed a valid ADO connection object and the **CustID** of the currently logged-in customer. If the object cannot be initialized, False is returned.

- **GetOrder(lngOrderID as Long)** — returns a Boolean value: Calling this method sets the object to point at the order specified by the **lngOrderID** argument. If the value of **lngOrderID** is not found in the underlying **Orders** table, False is returned.

- **NewOrder(lngCustID as Long)** — returns nothing: This method creates a new **Order** record for the customer in question.

- **ProcessOrder(objRequest as Request)** — returns nothing: This method simplifies the processing of new or changed orders. All that is required by the method is that it be passed an ASP **Request** object containing field/value pairs of the form **txtQtyOrdered_*<ProductID>*** = *value*. The method takes care of processing the order against the **Products** table so that the inventory values are up to date.

29.4.1.2 OrderDetails

The **OrderDetails** object simply provides a list of order details in the current order. It has the following public properties:

Property	Data type
OrderID	Long
ProductID	String
Description	String
Unit	String
QtyOnHand	Integer

Property	Data type
ActualPrice	Currency
QtyOrdered	Integer
QtyShipped	Integer
ExtendedPrice	Currency

`OrderDetails` provides the following public methods for iterating through the underlying recordset:

- **`MoveFirst`** — returns a Boolean value (True/False): The **`MoveFirst`** methods moves to the first order detail. If the list of order details is empty or undefined, the method returns False.
- **`MoveNext`** — returns a Boolean value (True/False): The **`MoveNext`** method moves to the next order detail in the list. If the record pointer is already at the end of file (EOF) marker, or the next record is the EOF marker, the method returns False.

29.4.2 Updating components

Software components can be a great time saver when developing applications. The component can be developed and tested in a controlled environment and then used in many different applications. In the case of web-based applications, components are especially useful because server-side scripting languages (e.g., VBSCRIPT) and programming tools for scripting and debugging are much less sophisticated than stand-alone development environments like VISUAL BASIC PROFESSIONAL or DELPHI.

The problem with component software is that the many different components from many different vendors sometimes conflict with each other. If certain programs break when a component is upgraded, or if an installation routine replaces the current version of a component with an older version, the result is generally known as “DLL Hell”. When the conflicts occur on a mission-critical web server, then the problem can affect thousands of users and multiple applications.

An important stipulation of the COM standard is that if a program works with a version of a component, it will work with all subsequent versions of that component. Thus, nothing should ever break by updating a component.[1] Moreover, an installation routine should *never* replace a component with an older version of the component.

29.4.3 Shopping carts and other interfaces

The approach you used in this lesson to create and process an order is very different from the

[1] Unfortunately, experience suggests that this stipulation is not always respected by software developers.

"shopping cart" metaphor used at many retail on-line stores. However, the assumption is made that users of this site will be ordering a large proportion of the items in the product list. Rather than add items one-by-one as done in a shopping carts, users simply pick what they want from an exhaustive list of products.

The HTTP protocol requires a "round trip" (client → server → client) to update the information on the user's screen. Because of the time required for the round trip, it is inefficient to replicate the interface of the stand-alone order form you created using ACCESS in Lesson 14. Client-side technologies (such as JAVA and DHTML) are giving web-based application designers more control over the user interface. However, these technologies are still relatively immature and are beyond the scope of this lesson.

29.4.4 Use of the DISABLED attribute in HTML

Not all browsers support the `DISABLED` attribute. For example, some versions of NETSCAPE NAVIGATOR simply ignore it. Since you have very little control over the browsers used by your customers, there is very little you can do about minor problems such as this.

In the case of the forms you developed in this lesson, the "disabled" status of a textbox is simply an aesthetic issue. As such, there are many different ways of making a clear demarcation between the data the user is expected to change and the data that is simply displayed for the users benefit. For example, textboxes could be used for values that can be changed by the user and plain text used for all other items.

The important thing to keep in mind is that unlike the textboxes you created in ACCESS, the HTML textboxes are not automatically bound to the underlying database. Thus, the only way that a change made to the data on an HTML form can be propagated to the database is through an update procedure written in VBSCRIPT. Since you have not written such scripts for the read-only fields, the integrity of the data in the database is not at risk regardless of the look and feel of your form.

29.5 Application to the project

33 Complete your order form using the `Order` and `OrderDetails` objects.

34 Test your application.