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

© Michael Brydon (mjbrydon@sfu.ca)
Last update: 02-May-01

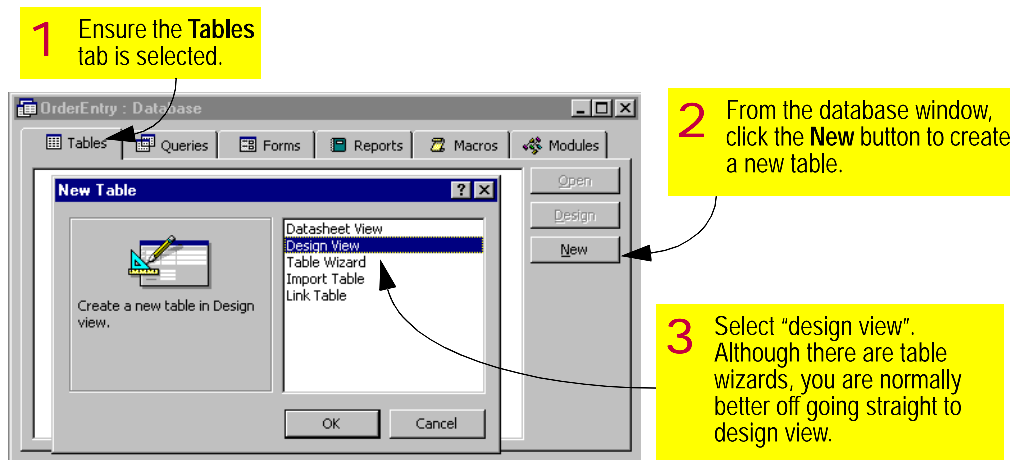
5.3 Exercises

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

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

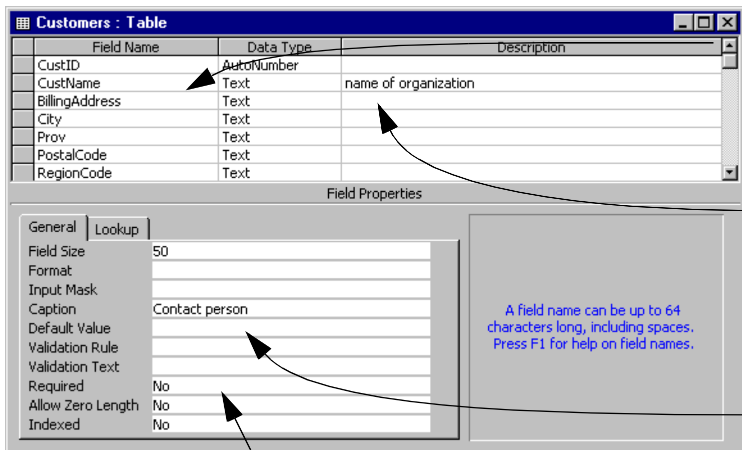
➡ Create the **Customers** table using the field names and data types shown in Table 5.1.

⚠ 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.



1 Enter the field names and data types for the customer attributes.

? The "description" column allows you to enter a short comment about the field. This information is not processed in any way by ACCESS, it simply allows you to document your design decisions.

? The "field properties" section allows you to enter information about the field and constraints on the values for the field.

2 Specify appropriate properties for each field. For now, you should leave the **Indexed** property set to its default value.

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`**.

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

- 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.

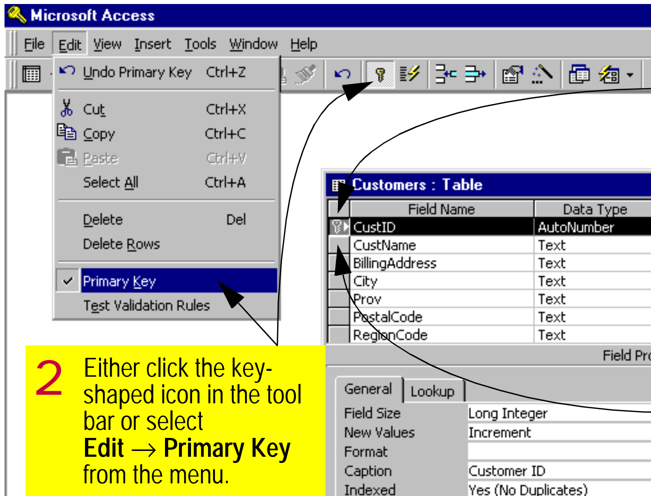
5.3.3 Specifying optional field properties

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.

- Set the **Caption** property for all fields that have "user *un*-friendly" names. For example, the caption for the **`CustName`** field

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



1 Click on the grey box beside the field (or fields) that form the primary key.

For a "concatenated" primary key (see Section 5.4.1), hold down the **Ctrl** key to select more than one field.

If the primary key is set properly, a small key icon will appear next to the field(s).

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.

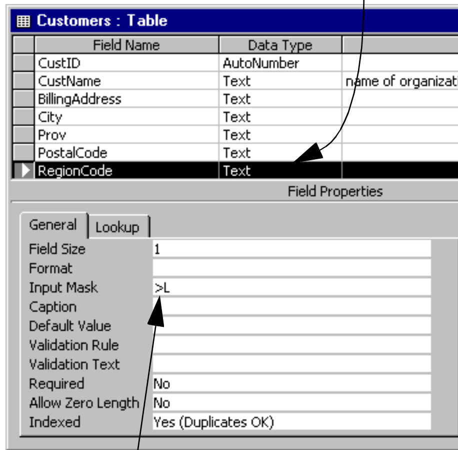
If you specify a caption for a field, ACCESS uses the caption instead of the field's name when displaying your data. This feature allows you to use short, compact field names, but present end-users with more meaningful field name aliases.

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

An input mask allows you to specify a "template" that controls the type of data that can be entered into the field. For

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

1 Click anywhere on the **RegionCode** field.



2 Enter “**>L**” in the **Input Mask** property. To learn more about the input mask symbols, press **F1** while editing the property.

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.

- 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.

(lesson5-1.avi)

- Bring up the **Customers** table in design view.
- 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.
- Follow the instructions provided by the wizard to create a phone number input mask.
- 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.

➡ 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.

➡ Set the input mask for **`RegionCode`** to "**`>L`**" as you did for the **`Customers`** table. In this way, the format of **`RegionCode`** data will be identical in both tables.

➡ Do not specify a caption for **`RegionCode`**. We will use this field in a subsequent lesson

---

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.

to illustrate the extra work that is created by ignoring the caption property.

➡ 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.

➡ 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.

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

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

➡ 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 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

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

The field names are shown in the “field selectors” across the top of the columns.

The records are shown as rows.

The grey boxes are “record selectors”.

The “navigation buttons” at the bottom of the window indicate the current record number and allow you to go directly to the first, previous, next, last, or new record.

You can resize the **RegionName** column by clicking near the right side of the field selector and dragging the border to the right. Double clicking the field selector boarder automatically sizes the column to fit its contents.

You can temporarily sort the records in a particular order by right-clicking any of the field selectors.

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 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

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

The black triangle indicates the “current record”.

The asterisk(*) indicates a place holder for a new record.

	RegionCode	RegionName
✎	N	North
*		

	RegionCode	RegionName
	N	North
✎	S	Sou
*		

	RegionCode	RegionName
	N	North
▶	S	South
*		

The small pencil means that the record buffer has been changed, but not yet saved to the database.

When the contents of the record buffer have been written to the database, the pencil changes back into a triangle

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 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

---

[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.

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

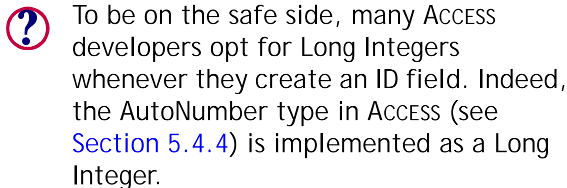
Numeric data types are typically divided into two major subtypes: **integer** and **real** (or **floating point**) numbers. Recall that integers are whole numbers (e.g., 6, -21) whereas real numbers have fractional parts expressed using decimal notation (e.g., 2.389, -813.2).

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].

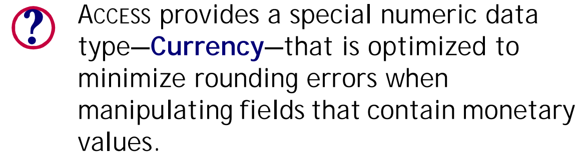
To address this problem, ACCESS provides a **Long Integer** data type. Since a Long Integer allocates twice as many bytes of storage as an Integer, it is capable of storing $2^{4 \times 8}$ unique values (numbers from -2,147,483,648 to 2,147,483,647).



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.

[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.

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**

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).

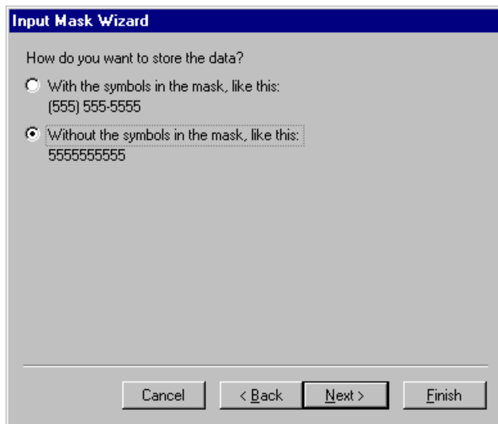
When you use a literal character in an input mask, the second argument determines whether the literal value is simply displayed or displayed *and stored* in the table as part of the data.

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?

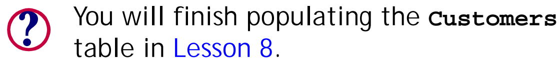
The bottom line is this: once a key is assigned, it should never be reused, even if the record to which it is assigned is subsequently deleted.

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

➡ Add the remaining values from Table 5.2 to the **Regions** table.



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