Teach Yourself InterBase

This tutorial takes you step-by-step through the process of creating and using a database using the InterBase Windows ISQL dialog. You learn to create data structures that enforce referential integrity constraints and maintain security. You populate your tables, create triggers and stored procedures, and learn a number of techniques for retrieving the data with precision.

There are five parts in this tutorial.

- In Part I, you learn how to use this tutorial, you start the InterBase server and log in to it, and you create a user and a database.

- In Part II, you learn the fundamentals of database design and how to work in the InterBase Windows ISQL environment; you create the data structures for your database, and you learn a little about how to recover from errors.

- In Part III, you put data into the database (you *populate* it).

- In Part IV, you get the data out again (you *access* or *retrieve* it).

- In Part IV, you work with database security, and create some triggers and stored procedures to automate some of your database tasks.

Part I **Getting Started**

In Part I, you perform the following actions:

- Check whether the Local InterBase server is running

- Start the Local InterBase server

- Log on to a server from the InterBase Server Manager and create a new user on the server

- Open InterBase Windows ISQL and create a new database on a server

Using this tutorial

Throughout this tutorial, you are instructed to enter SQL statements manually at the beginning of each new topic in order to give you hands-on experience with it. Then you are instructed to read in one of the SQL scripts that accompany this tutorial document. Following these steps allows you to create a database that is complex enough to be interesting without excessive keyboarding. The database that you create in this tutorial is, in fact, the EMPLOYEE database that is used as the Example database for InterBase and that is referenced throughout the InterBase document set.

Finding the files you need

As you reach the places in this tutorial that tell you to read in a script file, use the script files (**sql*) that are in the *\doc\Tutorial* directory on your InterBase CDROM.

This tutorial document and the accompanying SQL script files are also available on the InterBase web site at **http://www.interbase.com/**.

Typographic conventions

This tutorial document and the SQL scripts that accompany it use the following typographic conventions:

· Database names, keywords, and domain names are in ALL CAPS.

· Table names have initial caps and are in *italic*.

· Names of columns, indexes, stored procedures, and triggers are lowercase *italic*.

· File and path names are in *italic*.

Reading and typing capitals

Type of entry	Case sensitivity
SQL statements	When you're entering SQL statements into InterBase Windows ISQL, you can ignore the capitalization. The conventions listed above are to make it easy to read and understand the examples. You can enter the exercises in all lower case if you prefer.
Strings	Strings (anything inside of quotation marks) *are* case sensitive. There are a lot of strings in single quotes in this tutorial, and you must enter the case exactly as it's shown.
External references	When you refer to something outside of InterBase, such as a filename, the reference is case sensitive.

TABLE 1

Line breaks

- Line breaks are added within example statements to make them easy to read and understand. They are not required.

- When you're entering statements in InterBase Windows ISQL you don't have to follow the line breaks in the examples. Enter ones that make it easy for you to keep track of what you're doing. InterBase ignores line breaks within input SQL statements.

Understanding which parts to do

This tutorial contains some code examples that you are not supposed to enter into the TUTORIAL database. In other places, it gives the text of code that you *are* supposed to enter: these are your action items. To make things clear, parts that you are supposed to actually enter are all preceded by headings with a ▶ symbol, as in the following example:

▶ **Example of an action item head**

Actions that you are supposed to perform are always preceded by a head like the one above. Don't enter examples that are not preceded by this type of heading.

Starting the Local InterBase server

Only one instance of the InterBase server can run at a time, so to work on this tutorial, you need to check whether InterBase is running and start it if necessary.

▶ **Start the server**

To check whether InterBase is running On Windows 95 platforms, an icon appears in the tray when InterBase is running. This is also true when InterBase is running as an application on Windows NT. When InterBase running as a service on Windows NT platforms, there is no icon. To check whether InterBase is running as a service on NT, right-click on a blank area of the Windows Taskbar, choose Task Manager from the menu, and look for *ibguard.exe* or *ibserver.exe* in the Processes pane.

To start the Local InterBase server To start the Local InterBase server running as an application on either Windows 95 or Windows NT, choose InterBase Guardian from the InterBase folder of the Start menu. To start Local InterBase as a service on Windows NT platforms, double-click Services in the Control Panel, highlight the InterBase Guardian entry, and click Start.

Connecting to a server from Server Manager

The text of this tutorial assumes that you are working on the Local InterBase server. If you want to work on a remote server, you must have the password for a valid InterBase user on that server.

Note In this part of the tutorial, you connect to a server using the InterBase Server Manager, because your next task is to create a new user on that server. You must be working in Server Manager to create a user. Throughout the rest of this tutorial, you will connect to a server from the InterBase Windows IQSL dialog, since that's where you do most of the work of creating, populating, using, and maintaining a database.

In this exercise, you connect as the SYSDBA user, since that is the only user who can create new user accounts. You create a user called TUTOR, which is the account you will use for the rest of these exercises.
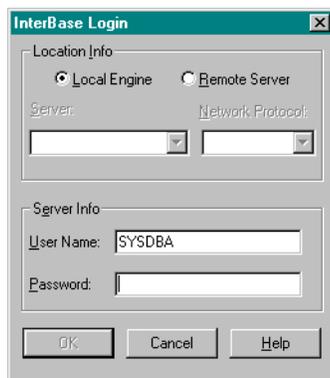
▶ **Log in to a server from Server Manager**

1. Open the InterBase Server Manager by choosing it from the InterBase folder on the Start menu.
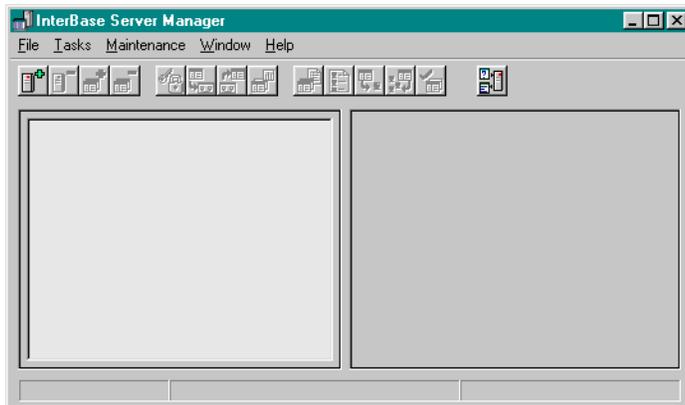

Server Login
button

2. To log in to the Local InterBase server, click the Server Login button or choose **File | Server Login** to display the InterBase Login dialog.

3. Click the Local Engine radio button and fill in the password for the SYSDBA user. By default, this password is `masterkey`. If you have changed the password (highly recommended!), use the current password. Click OK. In either case, you must log on as SYSDBA in order to create a new user account.

An icon for the local server appears in the left pane of Server Manager.
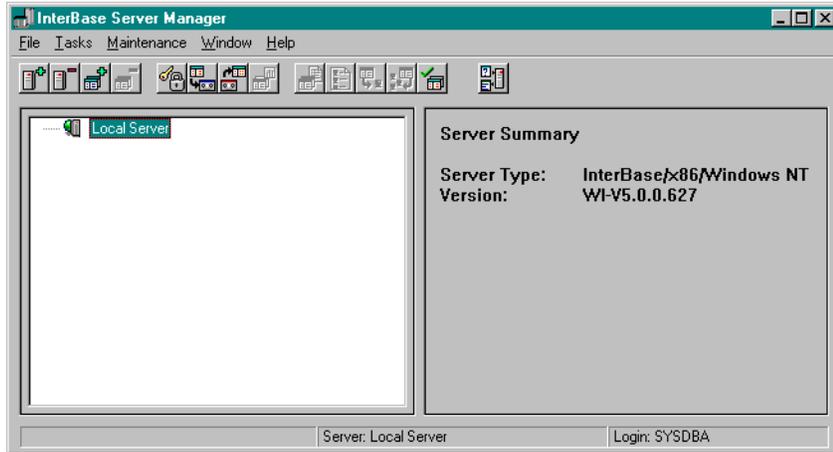
Creating a new user

The rest of this tutorial assumes that you are user TUTOR and that your password is `tutor4ib`. In this next exercise, you create user TUTOR.

▶ **Create a new user**

In the previous section, you opened Server Manager and connected to a server as an existing user. Now you create a new user, TUTOR. **Note** InterBase ships by default with one user, SYSDBA, defined.
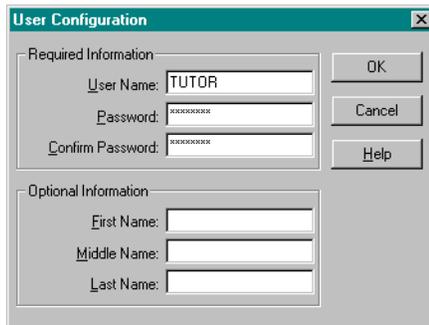
The User
Security
button

1. In Server Manager, choose **Tasks | User Security** to display the InterBase Security dialog or click the User Security button.

2. In the InterBase Security dialog, click the Add User button to display the User Configuration dialog.

3. Type TUTOR in the User Name field and tutor4ib in the Password and Confirm Password fields.

4. Click OK to close the User Configuration dialog. Click OK to close the InterBase Security dialog.

   You have now created a user on the server you're logged into. Users are defined server-wide and can connect to any database that resides on that server. Tables within these databases have additional security, however. Being able to connect to a database won't do you much good if you don't have privileges on any of its tables.

5. Now choose **File|Exit**. Choose Yes when InterBase asks you if you want to log out from all servers and exit Server Manager.
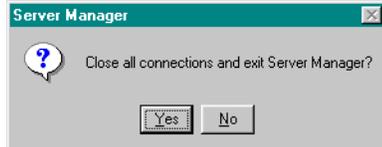
Note If you want to logout from a server without exiting Server Manager, choose **File|Server Logout** or click the Server Logout button ▯.

Creating a database

Now that you have used Server Manager to create a valid user name, you are ready to use InterBase Windows ISQL to create the TUTORIAL database that you will use for the exercises in this tutorial.

InterBase databases are stored in files that, by convention, have a *.gdb* extension.

▶ **Create the TUTORIAL database**

1. Open InterBase Windows ISQL by choosing it from the InterBase folder on the Start menu.

   Note If you're working in Server Manager, you can open InterBase Windows ISQL by choosing **Tasks | Interactive ISQL** or clicking the [icon] button.

FIGURE 1   The InterBase Windows ISQL dialog

Hold the mouse pointer over a Toolbar button to get Tooltips (balloon help) for that button

SQL Statement Area

SQL Output Area

Status Bar
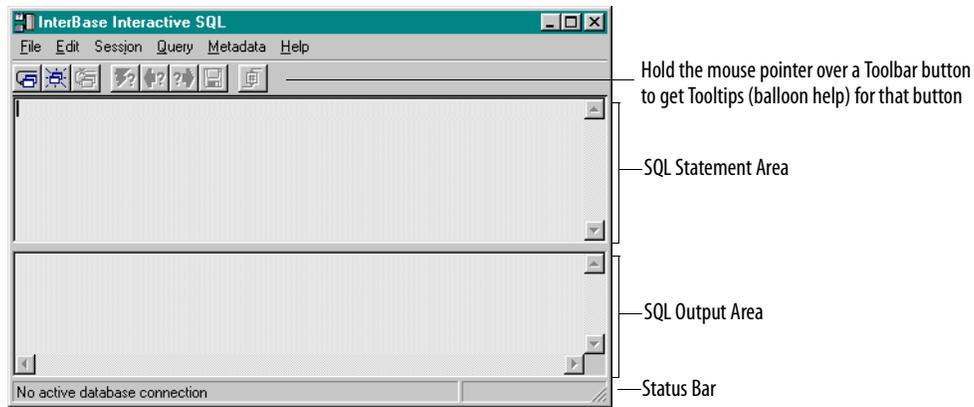
This dialog is described in more detail in "Using InterBase Windows ISQL" on page 12.

2. Choose a location for your TUTORIAL database. This example and the SQL script files use *C:\interbase5\tutorial\tutorial.gdb*. If you choose a different location, you must edit the CONNECT statement in the SQL script files to reflect the new location. (You're instructed how to do this a little later.)

3. Choose **File | Create Database** or click the Create Database [icon] button to display the Create Database dialog.

Create Database button
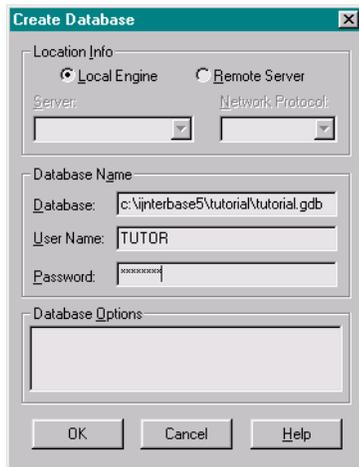
NOTE: Be sure you log in as user TUTOR for the remaining exercises in this tutorial.

Notice that this looks a lot like the dialog you saw when you connected to a server from Server Manager. But there's a difference: InterBase Windows ISQL combines logging in to a server with connecting to a database. Creating a database is a special case of this: you log in to a server, create a database, and connect to the new database all in one dialog.

4. Choose Local Engine.

5. In the Database field, type the full path to your new database, including the file name:

   `C:\interbase5\tutorial\tutorial.gdb`

6. Type TUTOR in the User Name field. You can type it in lower case; it displays in caps anyway. The User Name field displays the login of the last user who connected, so in the future, you may find this field already filled in correctly.

7. Enter the `tutor4ib` password in the Password field. You can ignore the Database Options field.

8. Click OK to create the TUTORIAL database. Its filename is *tutorial.gdb*.

You've now created a database named TUTORIAL that belongs to user TUTOR. You're also logged in to the server and connected to the new database. Look at the Status Bar at the bottom of the InterBase Windows ISQL window: you should see the path and name of the database you just created. Whenever you're connected to a database, the name and path appear in the Status Bar.

Database: c:\interbase5\tutorial\tutorial.gdb   Local Server
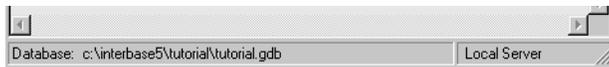
▶ **Disconnecting from a database**

1. Choose **File | Disconnect from Database** or click the Disconnect 📇 button.

2. When InterBase queries whether you want to disconnect from the database, choose OK.

Disconnect button

The Status Bar now tells you that there is no active database connection.

No active database connection

Part II Data Definition

In Part II, you perform the following actions:

- Take a quick look at data modeling
- Create some domains
- Execute SQL scripts
- Create three tables and a view
- View object definitions
- Alter a table
- Create, modify and drop indexes

Database design

The crucial first step in constructing any database is *database design*. This step is so important that volumes have been written about it. You can't create a functional, efficient database without first thinking through its components and desired functionality in great detail. Chapter 2 of the *Data Definition Guide* provides a good introduction to the topic.

A quick look at data modeling

This following list provides a brief and simple overview of the process of designing a database:

1. Determine data content.

   What information needs to be stored? In thinking about this, look at it from the point of view of the end users: What groups of end users will access the database? What information will they need to retrieve? What questions will they be asking of the database?

2. Group types of data together.

   Information items tend to group naturally together. Later, when you create tables in the database, you create one table for each group of data items. The granularity with which you divide the mass of information into groups depends on factors such as the quantity and complexity of the information your database must handle. The goal is to have each item of information in only one place. The process of identifying such groups is called *normalization*. Identify *entities* and their *attributes*. In this tutorial, for example, one type of entity is the project. A project's attributes are its ID number, name, description, leader's name, and product. Later in this tutorial, you will see that there is a table named *Project* that has columns named *proj_id*, *proj_name*, *proj_desc*, *team_leader*, and *product*.

3. Design the tables.

   Determine what tables you will create, what columns will be in each table, and what type of data each column will contain. If you have identified your entities and their attributes carefully, each entity will correspond to a table and each attribute will be a column in that table. This is the point where you decide on the datatype for each column, as well. Is the data numeric or text? If it's numeric, what is the expected range of values? If it's text, how long a string do you need to accommodate? Identify an appropriate datatype for each column. InterBase's supported datatypes are discussed in Chapter 4 of the *Data Definition Guide*.

4. Consider the interdependencies of your table columns.

   You can't sell an item, for example, unless you have it in inventory. You can't deliver it unless it's in stock. You create primary keys and foreign keys to maintain these dependencies. This is called *maintaining database integrity*. Other mechanisms for maintaining database integrity and security include CHECK constraints, and using GRANT and ROLE statements to control access to tables.

The TUTORIAL database

The TUTORIAL database that you create in this tutorial is, in fact, an exact copy of the EMPLOYEE database that is used for examples throughout the InterBase document set. The TUTORIAL database is a generic business database. Imagine, for the purposes of this tutorial, that you are responsible for creating a database for this company. In the data modeling phase, you identify the following entities (information groups):

departments	jobs	countries	customers
employees	projects	employee projects	sales
department budgets for each project	salary history for each employee		

You will see, as you progress through this tutorial, that the TUTORIAL database contains ten tables that correspond exactly to the ten items above. To get an overview now, you can look at an article about the EMPLOYEE database, since the EMPLOYEE database and the finished TUTORIAL database are identical. Go to **http://www.interbase.com/tech/exampledb/exampledb.html** on the InterBase website.

An overview of SQL

SQL statements fall into two major categories:

- Data definition language (DDL) statements
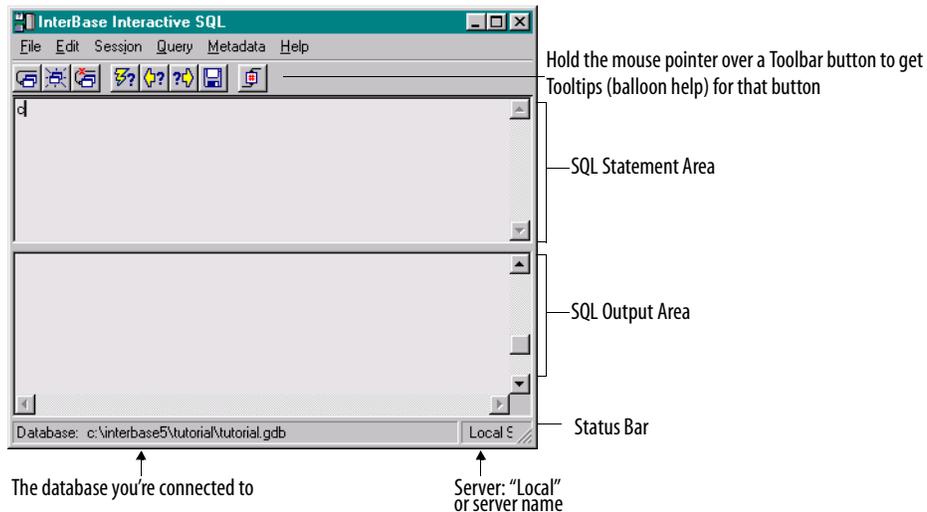- Data manipulation language (DML) statements

DDL statements define, change, and delete the structures that hold data. These include the database itself, tables, and other elements that are part of the database such as domains, indexes, triggers, stored procedures, roles, and shadows. Collectively, the objects defined with DDL statements are known as *metadata*. DDL statements begin with the keywords CREATE, ALTER, and DROP. For example, CREATE TABLE defines a table, ALTER TABLE modifies an existing table, and DROP TABLE deletes a table.

DML statements manipulate data within these data structures. The four fundamental DML statements are INSERT, UPDATE, DELETE, and SELECT. INSERT adds data to a table, UPDATE modifies existing data, and DELETE removes data. The SELECT statement retrieves or *queries* information from the database. It is the most important—and most complex—of all the SQL statements, because it is the means by which you access all the information that you have so meticulously stored.

In Part II of this tutorial, you use several DDL statements—CREATE DOMAIN, CREATE TABLE, ALTER TABLE, CREATE VIEW, and CREATE INDEX—to create data structures for your TUTORIAL database. In Part III, you use the DML statements INSERT, UPDATE, and DELETE to add data to your database and modify it. Part IV teaches you the all-important SELECT statement—also a DML statement. In Part V, you learn two advanced DDL statements, CREATE PROCEDURE and CREATE TRIGGER.

Using InterBase Windows ISQL

This section describes how to use the InterBase Windows ISQL graphical interface to enter, execute, and commit SQL statements.

Hold the mouse pointer over a Toolbar button to get Tooltips (balloon help) for that button

SQL Statement Area

SQL Output Area

Status Bar

The database you're connected to

Server: "Local" or server name

Entering SQL statements in the SQL Statement Area

You enter SQL statements by typing them in the SQL Statement area.

▪ You don't need to end statements with a terminator, such as the semicolon, when you are using InterBase Windows ISQL. However, the terminator is not prohibited and doesn't cause a problem. You can, for example, copy and paste statements from scripts (where they must have terminators) and run them without removing the terminators.

▪ InterBase SQL statements are not case sensitive. You can enter all SQL statements in lowercase if you prefer.

▪ *Anything inside quotation marks is case sensitive* and must be entered as shown.

▪ InterBase ignores line breaks within statements. They are for your convenience only.

▪ You can use spaces to indent lines to make them easier to read. InterBase ignores these spaces. You cannot indent using tabs.

▪ You must execute each statement before entering the next one.

Executing statements

There are three ways to execute a statement in InterBase Windows ISQL:

▪ You can choose **Query | Execute**

▪ You can click the Execute Query ⚡ button

▪ You can press Ctrl - Enter

You must execute each statement before entering the next one.

Note Although this method of entering SQL statements by hand one at a time is an option in InterBase, users often use SQL scripts (data definition files) as a more convenient way of entering data. See "Running an SQL script" on page 17 for more information.

Committing your work

Until you commit your work, your transaction is said to be *active*. Work associated with an active transaction is not yet visible to other users. You often want to enter an entire group of related items before committing, so that misleading intermediate states are never visible. When you commit, your transaction changes to a committed state and the work you did in that transaction becomes visible to other users. When you are creating metadata, however, you usually want to commit each data structure as you complete the DDL statement. (See page 12 for a discussion of DDL and DML statements.)

The InterBase Windows ISQL environment provides an Autocommit feature that automatically commits any DDL statement when you execute it. The Autocommit feature does not apply to DML statements such as INSERT, UPDATE, DELETE, and SELECT. The Autocommit feature is enabled by default in InterBase.

▶ **Check session settings**

To check the status of the Autocommit feature and other session settings, follow these steps:

1. Choose **Session | Basic Settings** to display the Basic ISQL Set Options dialog.
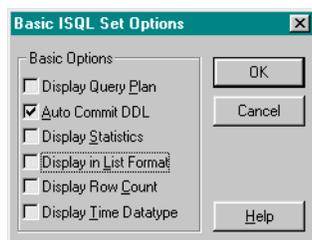
2. Check to see that the Auto Commit DDL box is checked. If necessary, enable the feature.

For further information on how to use InterBase Windows ISQL, see the *Operations Guide*.

Creating domains

A *domain* is a customized column definition that you use to create tables. When you create a table, you specify the characteristics of each column in the table. Often, across the tables in a database, there will be several columns that have the same characteristics. Rather than entering the same complex definition for each column, you can create a name for the collection of characteristics. This named set of column characteristics is called a domain. You can use this domain name in a column definition rather than typing out the full definition.

IMPORTANT   If you are going to use domains in your column definitions, you must create the domains before you use them in table definitions.

Column characteristics include:

- Datatype
- Default value: a literal value, NULL, or the name of the current user (USER)
- Nullability: NOT NULL prohibits NULL values in the column (columns are nullable by default)
- CHECK constraints: checks that the value being entered meets specified criteria
- Character set and optional collation order (output sort order for CHAR and VARCHAR columns)
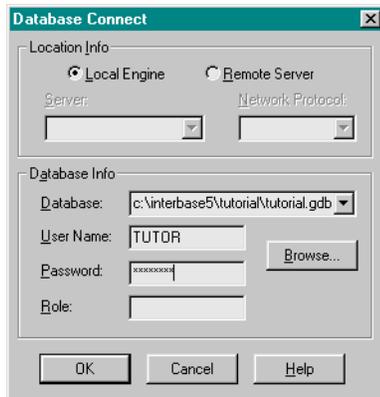
When you specify a column, only a column name and datatype are required. All other characteristics are optional.

▶ **Connect to the TUTORIAL database**

1. In InterBase Windows ISQL, choose **File | Connect to Database** or click the Connect ⬚ button.

The Connect
button

2. Choose Local Engine.

3. In the Database field, type the complete path to the database:

   `C:\interbase5\tutorial\tutorial.gdb`

   Database Connect

   ─ Location Info ─
   ● Local Engine   ○ Remote Server
   Server: Network Protocol:
   [▼]   [▼]

   ─ Database Info ─
   Database: [c:\interbase5\tutorial\tutorial.gdb ▼]
   User Name: [TUTOR]   [Browse...]
   Password: [xxxxxxxx]
   Role: [   ]

   [OK]   [Cancel]   [Help]

   Shortcuts You can skip this step if the correct path and name is already showing. If another name is showing, but you have connected to this database recently, you can choose it from the dropdown Database list. If it's not on the dropdown list and you've forgotten the exact name and path of the database, you can click the Browse button to locate it.

4. Type TUTOR in the User Name field and type tutor4ib in the Password field.

   The Role field does not require an entry. The Role parameter is optional and is discussed in "Access privileges" on page 63.

5. Choose OK.

TIP   Look in the Status Bar to confirm that you are connected to the TUTORIAL database.

Database: c:\interbase5\tutorial\tutorial.gdb   Local Server

Entering metadata statements

In this exercise, you use the CREATE DOMAIN statement to create domains that you will use later to specify column datatypes.

▶ **Create some domains**

In the following exercise you define four domains. The first three specify only a datatype. The fourth one is more complex. In each case, the domain will be useful for several different columns, not just the column for which it is named.

You should be connected to the TUTORIAL database when you begin this exercise.

1. Type the following code in the SQL Statement Area to define a domain called FIRSTNAME that has a datatype of VARCHAR(15).

```
CREATE DOMAIN FIRSTNAME AS VARCHAR(15)
```

2. Execute the statement: click the 🔁 button, press Ctrl-Enter, or choose **Query|Execute**.

3. Now create two more domains, LASTNAME and EMPNO. Execute each statement before entering the next one.

```
CREATE DOMAIN LASTNAME AS VARCHAR(20)

CREATE DOMAIN EMPNO AS SMALLINT
```

Each statement appears in the SQL Output Area after it executes.

4. Next, enter and execute the following code to define a domain for department numbers. The domain is defined as a three-character string. In addition to the datatype, it includes check constraints to ensure that the department number is either "000", alphabetically between "0" and "999", or NULL. Pay attention to parentheses and quotes as you enter this:

```
CREATE DOMAIN DEPTNO AS CHAR(3)
   CHECK (VALUE = '000'
       OR (VALUE > '0' AND VALUE <= '999')
       OR VALUE IS NULL)
```

TIP   When you're typing an SQL statement that has parentheses, take a moment to count the left parentheses and the right parentheses and make sure that there are the same number of each. Mismatched parentheses are a major source of errors in SQL code. In the example above, there are nested parentheses: the CHECK clause is enclosed in parentheses because it contains three parts ("A OR B OR C") and the second part of the clause has parentheses because it also contains multiple parts ("A AND B").

The CREATE DOMAIN statement above is divided into several lines to make it easy for humans to follow the syntax. InterBase ignores the line breaks when parsing the statement. Enter the whole statement before executing it.

Note You don't need to commit your work, because CREATE statements are DDL (data definition language) statements. You turned **Autocommit DDL** on in the Session Settings earlier in this tutorial, so all these DDL statements have been committed automatically.

Data definition files

Since you've already created several domains, you can use an SQL script—also called a *data definition file*—to create the rest of the domains. A data definition file is a text file that contains SQL statements. It can be executed in InterBase Windows ISQL and is typically created with a text editor such as Notepad.

It is often convenient to create a data definition file rather than typing each statement directly into InterBase Windows ISQL, because the text editor provides you with an editing environment and the script provides a reusable record of what was entered. In practice, most data definition is accomplished using data definition files.

IMPORTANT   Every SQL script must begin with a CONNECT statement. The CONNECT statement specifies a database name including the complete path, a user name, and password. The SQL scripts that accompany this tutorial begin with the following CONNECT statement:

```
CONNECT 'C:\interbase5\tutorial\tutorial.gdb'
USER 'TUTOR' PASSWORD 'tutor4ib'
```

If this is not the correct information for you, you must edit each SQL file and make the CONNECT string correct. The SQL scripts are text files that you can modify in any text editor. If you use an application that saves by default in a proprietary format, be sure to save the files as text.

Running an SQL script

Before running an ISQL script, it is good practice to open the file in a text editor and check that the CONNECT statement provides the correct server, database, user name, and password. Edit the information if necessary.

The remaining domain definitions that are needed for your TUTORIAL database are in the *Domains.sql* data definition file. In this section of the tutorial, you execute that file to create the remaining domain definitions.

▶ **Run the *Domains.sql* script**

1. Open *Domains.sql* in a text editor and make sure that the CONNECT statement specifies the correct path, database name, user name, and password.

2. Notice that the file contains many descriptive passages that are commented out. The convention for comments is exactly like that for the C language: comments begin with `/*` and end with `*/`.

3. Look through the file and notice that each SQL statement ends with a semicolon (`;`). Semicolons are required at the end of each statement in a data definition file. They are not required when you type statements directly into InterBase Windows ISQL.

4. In InterBase Windows ISQL, choose **File | Run an ISQL Script**. InterBase Windows ISQL asks if you want to commit your work. Choose Yes. The Execute ISQL Script File dialog displays:
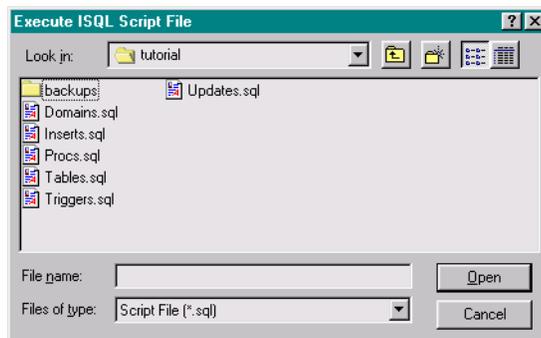
5. Navigate to the location where you have stored the tutorial SQL files and highlight *Domains.sql*. Click Open.

6. A dialog box appears, asking if you want to save the results to a file. Click No, since you want to see the results in the SQL Output Area of the ISQL window.

    (If you choose Yes, InterBase Windows ISQL prompts you for a filename in which to store the output. No output appears in the SQL Output Area.)

    InterBase reads the file and executes each statement and posts a message that says "Script completed successfully."

Troubleshooting If you receive a message stating that there are errors, click the Details button and read the information; it is often useful. Also check that the CONNECT information (database, user, and password) in the script file is correct.

7. To confirm that the domains now exist, choose **Metadata | Show**, select Domain from the View Information On list, and click OK.
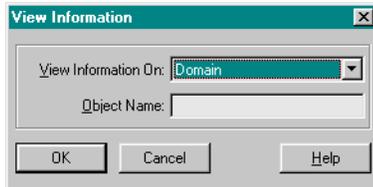
You should see all the domains defined for the database displayed in the SQL Output area. (Note: If you were not connected to *tutorial.gdb* before running the script, you will need to connect now before the **Show** command is available.) You should see the following domains:
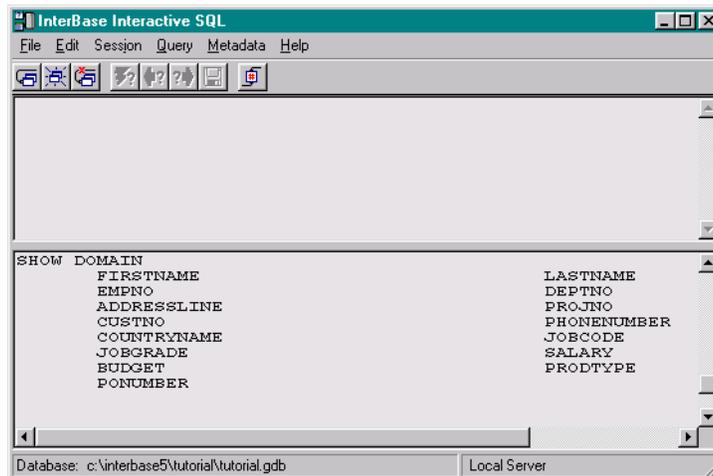
Creating tables

A table is a data structure consisting of an unordered set of rows, each containing a specific number of columns. Conceptually, a database table is like an ordinary table. Much of the power of relational databases comes from defining the *relations* among the tables.

The CREATE TABLE statement has the following general form:

```
CREATE TABLE tablename (colname1 characteristics[, colname2 characteristics, …]
   [, tableconstraint …])
```

▪ C*haracteristics* must include a datatype and can also include several other things. See "Creating domains" on page 14 for a list of column characteristics.

▪ A *table constraint* can be a CHECK, UNIQUE, FOREIGN KEY, or PRIMARY KEY constraint on one or more columns.

For the full syntax of the CREATE TABLE statement, see the *Language Reference*.

In the following steps, you create three of the ten tables for the TUTORIAL database.

▶ **Create the *Country* table**

The first table—the *Country* table—has only two columns. The column definitions are separated by commas. For each column, the first word is the column name and the following words are characteristics. The first column, *country*, has the COUNTRYNAME domain and in addition is NOT NULL and a primary key. Primary keys are discussed in a following section of this tutorial.

1. In InterBase Windows ISQL, connect (attach) to *tutorial.gdb* as TUTOR, if you are not connected already.

2. Enter the following CREATE TABLE statement. The layout below is for ease of reading; the line endings are not required:

```
CREATE TABLE Country (
    country COUNTRYNAME NOT NULL PRIMARY KEY,
    currency VARCHAR(10) NOT NULL)
```

Notice that the collection of column definitions is surrounded by parentheses and that the columns are separated by commas.

3. Execute the statement (click the 🛂 button, press Ctrl-Enter or choose **Query|Execute**). If you entered the code without errors, it appears in the SQL Output Area.

▶ **Create the *Department* table**

Next, you create the *Department* table. This table has only two columns to begin with. Later, you use the ALTER TABLE command to add to it. Type in the complete SQL statement and then execute it:

```
CREATE TABLE Department (
    dept_no DEPTNO NOT NULL PRIMARY KEY,
    department VARCHAR(25) NOT NULL UNIQUE)
```

The *dept_no* column is the primary key for the table and is therefore UNIQUE. Primary keys are discussed on page 21. Notice that the *department* column value must be unique and that neither column can be null.

▶ Create the *Job* table

Now you create the more complex *Job* table. This definition includes CHECK constraints, PRIMARY KEY and FOREIGN KEY constraints and a BLOB datatype for storing descriptive text. The text following the code discusses these new elements.

1. Enter the following CREATE TABLE statement in the SQL Statement Area. Type in the whole statement and then execute it.

```
CREATE TABLE Job (job_code JOBCODE NOT NULL,
    job_grade JOBGRADE NOT NULL,
    job_country COUNTRYNAME NOT NULL,
    job_title VARCHAR(25) NOT NULL,
    min_salary SALARY NOT NULL,
    max_salary SALARY NOT NULL,
    job_requirement BLOB SUB_TYPE TEXT SEGMENT SIZE 400,
    language_req VARCHAR(15)[1:5],
CONSTRAINT pkjob PRIMARY KEY (job_code, job_grade, job_country),
CONSTRAINT fkjob FOREIGN KEY (job_country) REFERENCES Country (country),
CHECK (min_salary < max_salary))
```

The CHECK constraint at the end checks that the minimum salary is less than the maximum salary.

The three-column primary key guarantees that the combination of the three columns identifies a unique row in the table.

The foreign key checks that any country listed in the *Job* table also exists in the *Country* table.

The BLOB datatype used for the *job_requirement* column is a dynamically sizable datatype that has no specified size and encoding. It is used to store large amounts of data such as text, images, sounds, and other multimedia content.

2. To check that the tables now exist in the database, choose **Metadata|Show**, choose Table from the View Information On list, and choose OK. The SQL Output Area should list three tables: *Country*, *Department*, and *Job*.

Primary keys and foreign keys

The *Job* table definition includes a primary key and a foreign key.

■ A *primary key* is a column or group of columns that uniquely identify a row. Every table should have a primary key. No table can have more than one primary key. The PRIMARY KEY characteristic can be specified as part of a column definition, as in the first column of the *Country* table, or it can be specified as a separate clause of the CREATE TABLE statement, as in the statement that creates the *Job* table. The primary key in the *Job* table is made up of three columns: *job_code*, *job_grade*, and *job_country*. While a value can appear more than once in any of these columns taken individually, the fact that they are collectively a primary key means that the three values taken together cannot occur in more than one row.

■ A *foreign key* is a column or set of columns in one table whose values *must* have matching values in the primary key of another (or the same) table. A foreign key is said to *reference* its primary key. Foreign keys are a mechanism for maintaining data integrity. In the *Job* table, for example, any country listed in the *job_country* column must also exist in the *Country* table. By stating that the *job_country* column of the *Job* table is a foreign key that references the *country* column of the *Country* table, you are guaranteeing this, because InterBase will return an error if a value is entered the *job_country* column that does not have a matching entry in the *country* column of the *Country* table.

You can declare a constraint such as UNIQUE, FOREIGN KEY, or PRIMARY KEY either as part of a column definition, or as a *table constraint* following the column definitions. The syntax varies slightly depending on which you choose. See the *Language Reference* for details.

You declared the primary key constraint as part of a column definition in the *Country* and *Department* tables. For the *Job* table, you declared the primary key, foreign key, and check constraints at the table level. Functionally, the effect is the same.

Cascading referential integrity constraints

When you create a foreign key, you are saying that the value in the foreign key must also exist in the primary key that it references. What happens if later, the value in the referenced primary key changes or is deleted? The *cascading referential integrity constraints*, new in InterBase 5, let you specify what should happen. Your choices are to take no action, to propagate (cascade) the change to the foreign key column, to set the foreign key to its default, or to set it to NULL.

If you are specifying the foreign key as part of the column definition, the syntax is this:

```
CREATE TABLE table_name (column_name datatype FOREIGN KEY
   REFERENCES other_table(columns)
      [ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
      [ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}],
   [, more columns defs])
```

If you are specifying the foreign key as a table-level constraint, the syntax is nearly the same except that you have to identify the column for which it is being defined, so the syntax becomes:

```
CREATE TABLE table_name (column_defs,
   FOREIGN KEY (column_name) REFERENCES other_table(columns)
      [ON UPDATE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}]
      [ON DELETE {NO ACTION | CASCADE | SET DEFAULT | SET NULL}],
   [, more table constraints])
```

A little later, you will use ALTER TABLE to add columns and table constraints to the *Department* table, including some cascading referential integrity constraints.

Naming constraints

When you declare a constraint at either the column level or the table level, you have the option of naming the constraint using the optional CONSTRAINT keyword, followed by a constraint name. When a constraint is named, you can drop it using the ALTER TABLE statement. In the *Job* table definition, two of the constraints have names "pkjob" and "fkjob"), but the CHECK constraint does not have a name (although it *could* have). When you alter the *Department* table a little later, you will add two named constraints.

Computed columns

When you are creating a table, you can define columns whose value is based on the values of one or more other columns in the table. The computation can include any arithmetic operations that are appropriate to the datatypes of the columns. Open *Tables.sql* and look at the following column definition for the *Employee* table:

```
full_name COMPUTED BY (last_name || ', ' || first_name)
```

The value of the *full-name* column consists of the value in the same row of the *last_name* column plus a comma plus the value of the *first_name* column.

Look at the *new_salary* column of the *Salary_history* table.

```
new_salary COMPUTED BY (old_salary + old_salary * percent_change / 100)
```

To find the value of *new_salary*, InterBase multiplies the value of *old_salary* by the value of *percent_change*, divides the result by 100, and adds that to the original value of *old_salary*.

Backing up a database

This is a good time to back up your database, because you've finished entering some tables. In the next part of the tutorial, you run a script to create more tables. Throughout this tutorial, you will be instructed to back up your database frequently. That way, if you run into difficulties, you can restore the last correct version and try again.

In a production database, a full backup and restore performs several functions:

- It preserves your data by making a copy of both the data and the data structures (metadata).
- It improves database performance by balancing indexes and performing garbage collection on outdated records.
- It reclaims space occupied by deleted records, and packs the remaining data.
- When you restore, it gives you the option of changing the database page size and of distributing the database among multiple files or disks.

IMPORTANT   If you restore a database to a name that is already in use, be sure that no users are connected to it at the time you restore. For this tutorial, it is sufficient to close InterBase Windows ISQL and make sure that there are no connections to the database from Server Manager.

▶ **Back up your database**

Before you begin, create a subdirectory called *backups*. If you are using the recommended directory path for this tutorial, your backups would be in *C:\interbase5\tutorial\backups*.

1. Open Server Manager by choosing it from the InterBase folder on the Start menu.

2. Log in as TUTOR to the server where your TUTORIAL database is located. (See "Connecting to a server from Server Manager" on page 5 if you've forgotten how to do this.)

3. Click the Backup ⊞ button or choose **Tasks|Backup** to display the Database Backup dialog.

```
┌─ Database Backup ──────────────────────────────── [X] ┐
│ ┌─ Backup Source ─────────┐ ┌─ Options ─────────────┐ │
│ │ Server:  Local Server   │ │ ☐ Transportable Format │ │
│ │            [ Remote... ] │ │                        │ │
│ │ Database Path:          │ │ ☐ Backup Metadata Only │ │
│ │ [c:\interbase5\tutorial\tutorial.gdb  ▼] │ ☐ Disable Garbage Collection │ │
│ │                         │ │ ☐ Ignore Transactions In Limbo │ │
│ │ ┌─ Backup Destination ──┐ │ ☐ Ignore Checksums    │ │
│ │ │ Server:  Local Server │ │ ☑ Verbose Output      │ │
│ │ │ Backup File or Device:│ └────────────────────────┘ │
│ │ │ [c:\interbase5\tutorial\backups\tutor1.gbk] │       │
│ │              [ OK ]  [ Cancel ]  [ Help ]            │
└───────────────────────────────────────────────────────┘
```
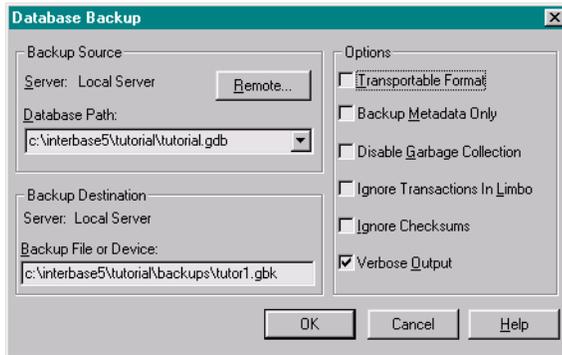
4. In the Backup File or Device field, name your backup file *Tutor1.gbk* and include the complete path to it. Your entry should look like this:

    ```
    C:\interbase5\tutorial\backups\tutor1.gbk
    ```

    By convention, backups have a *.gbk* extent, but it is not required. Enable Verbose Output in order to see a detailed description of what InterBase does when it backs up a database. Choose OK.

5. InterBase posts a dialog describing its progress. When the process is complete, dismiss the dialog and choose **File|Exit** to exit Server Manager.

Creating tables with a script

You've created several tables manually now and begin to understand what's involved. To avoid having to type in all of the table definitions, you should now run the *Tables.sql* script.

▶ **Run the *Tables.sql* script**

Before you leave the topic of tables, look over the remaining table definitions in *Tables.sql* to be sure that you understand them. Pay particular attention to the *Employee* table, which is complex and is central to this database. Notice, in particular, the complex CHECK constraint on the *salary* column in that table. It states that the salary entered for an employee has to be greater than the minimum salary for the employee's job (specified by *job_code*, *job_grade*, and *job_country*) and less than the corresponding maximum.

1. Run the *Tables.sql* script to enter the remaining table definitions into the TUTORIAL database. As before, check first that the database path, user name, and password are correct in the CONNECT statement at the beginning of the file. Then choose **File | Run an ISQL Script**, select the *Tables.sql* file, and choose Open.

2. Use the **Metadata | Show** command to check that you now have ten tables in the TUTORIAL database.
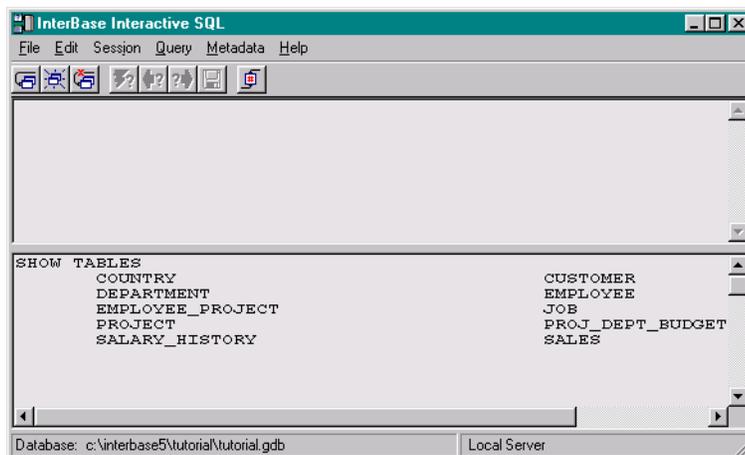
Time to back up If you have successfully run *Tables.sql*, this is a good time to back up your database to *Tutor2.gbk*.

Troubleshooting

If you made any typing mistakes when you were entering the domain definitions, you'll get an error message when you run the *Tables.sql* script or when you are defining the tables manually. InterBase posts a message that looks like this:

Look in the SQL Output area for more information. The SQL Output area echoes the contents of the script. If there were problems with a particular table, the SQL code for that table is followed by an error message such as the following:

```
Statement failed, SQLCODE = -607
Dynamic SQL Error
-SQL error code = -607
-Invalid command
-Specified domain or source column does not exist
```

To understand the problem, follow these steps:

1. Read the error text. In this case, it says that the specified domain does not exist. You probably made an error in typing the domain name.

2. Choose **Metadata | Show** and choose Domains in the View Information On list. Choose OK. InterBase lists the domains in the SQL Output Area.

3. You defined four domains by hand: FIRSTNAME, LASTNAME, EMPNO, and DEPTNO. Look for each of these in the list of domains and make sure that their names are spelled correctly. It's likely that you will find one that is misspelled.

4. Drop the incorrect domain by entering and executing the following command:

    ```
    DROP DOMAIN domainname
    ```

5. Recreate the domain using the CREATE DOMAIN statement.

6. Run the *Tables.sql* script again.

If this isn't the problem, continue with these steps:

7. Look right above the message text to see which table has the problem. Note which domains are used in that table. Do they include any of the four domains that you entered by hand?

8. Choose **Metadata | Show** and choose Domains in the View Information On list. Type the name of the first hand-entered domain that you noted in step 7 in the Object Name box and choose OK. InterBase displays the domain's definition in the SQL Output Area.

9. Compare the displayed definition with the definition given in this document (the one that you typed). Continue checking each of the four hand-entered domains until you find one that has a problem.

10. Drop the domain as described in step 4 above, and then re-enter it correctly. Run the *Tables.sql* script again.

Viewing an object definition

You can see the definition of any object in a database using the Show command from the Metadata menu. You've already used **Metadata|Show** to list what domains and tables exist. Now you use it in a different way to get information about a specific object.

▶ **View the definition of the *Department* table**

To refresh your memory of the current *Department* table definition, follow these steps:

1. Choose **Metadata|Show** from the InterBase Windows ISQL menus.

2. Choose Table from the View Information On list

3. Type `department` in the Object Name field. Case doesn't matter.

4. Click OK. InterBase displays the definition of the *Department* table in the SQL Output Area.

Altering tables

You can change the structure of existing tables with the ALTER TABLE statement. In the previous section of the tutorial, you created a simple *Department* table. Now you use the ALTER TABLE statement to add to this table. The syntax for altering a table—in simplified form—is:

```
ALTER TABLE table_name operation [, operation]
```

where each *operation* is one of the following:

```
ADD column
ADD tableconstraint
DROP column
DROP CONSTRAINT constraintname
```

Notice that you can drop a constraint only if you gave it a name at the time you created it.

▶ **Alter the *Department* table**

You now add five new columns (*head_dept*, *mngr_no*, *budget*, *location*, and *phone_no*) and two foreign key constraints to the *Department* table that you created earlier.

1. Type the following code into the SQL Statement Area of the InterBase Windows ISQL dialog and then execute it:

```
ALTER TABLE Department
   ADD head_dept DEPTNO,
   ADD mngr_no EMPNO,
   ADD budget BUDGET,
   ADD location VARCHAR(15),
   ADD phone_no PHONENUMBER DEFAULT '555-1234',
   ADD FOREIGN KEY (mngr_no)
      REFERENCES Employee (emp_no) ON DELETE CASCADE ON UPDATE CASCADE,
   ADD CONSTRAINT fkdept FOREIGN KEY (head_dept)
      REFERENCES Department (dept_no) ON DELETE CASCADE ON UPDATE CASCADE
```

2. Use **Metadata|Show** ->Table->Department to see the new table definition.

More troubleshooting

If you receive error messages when you are altering tables or inserting data, use the **Metadata | Show** command as your resource.

▪ Show the definition for each table that you entered by hand and compare the output to the SQL code that this document instructs you to enter.

▪ When you find a problem, you can either drop the table and recreate it, or use ALTER TABLE to drop a column and then add the column again with the correct definition. If you misspelled the name of the table itself, you must drop the table and recreate it.

▪ The DROP TABLE statement has the following syntax:

```
DROP TABLE tablename
```

▪ To change a column definition, first drop it using the ALTER TABLE statement:

```
ALTER TABLE tablename DROP columnname
```

▪ Then add the column back in using the ALTER TABLE statement again:

```
ALTER TABLE tablename ADD columnname columndef
```

If you made any typing errors when creating the domains and tables, you will get errors when you try to insert data by hand or to run the *Inserts.sql* and *Update.sql* scripts. If you follow the steps above, you will be able to fix your errors and run the scripts successfully. The remainder of the tutorial is less demanding, in that it focuses on the SELECT command. Once you detect and fix any errors in the domain and table definitions, you will get the correct results from your SELECT statements.

Creating views

A view is a virtual table that contains selected rows and columns from one or more tables or views. InterBase stores only the definition of a view. The contents of a view are essentially pointers to data in the underlying tables. When you create a view, you are not copying data from the source tables to the view. You are looking at the original data.

A view often functions as a security device, because you can give people permissions on a view but not on the underlying tables. Thus, the people can access a defined part of the data (the part defined in the view), but the rest of the data remains private.

In the following exercise, you use the CREATE VIEW statement to create a phone list by choosing the employee number, first name, last name, and phone extension from the *Employee* table and the employee's location and department phone number from the *Department* table. Views are frequently created to store an often-used query or set of queries in the database.

You can select from a view just as you can from a table. Other operations are more restricted. See "Working with Views" in the *Data Definition Guide* for more on views.

▶ **Create the *Phone_list* view**

1. Enter the following statement to create the *Phone_list* view from selected columns in the *Employee* and *Department* tables.

```
CREATE VIEW Phone_list AS
    SELECT emp_no, first_name, last_name, phone_ext, location, phone_no
    FROM Employee, Department
    WHERE Employee.dept_no = Department.dept_no
```

The WHERE clause tells InterBase how to connect the rows: the *dept_no* column in the *Department* table is a foreign key that references the *dept_no* column in the *Employee* table. Both columns are UNIQUE and NOT NULL, so the *dept_no* value in a *Department* row uniquely identifies a row in the *Employee* table. (In case you're wondering, the *dept_no* column in the *Employee* table is UNIQUE because all primary key columns automatically acquire the UNIQUE property.)

Notice that when the same column name appears in two tables in a query, you reference the columns by giving both the table name and the column name, joined by a period:

```
table_name.column_name
```

2. Now look at the structure of the *Phone_list* view by choosing **Metadata | Show**->View and type `phone_list` in the Object Name field. You should see the following output:
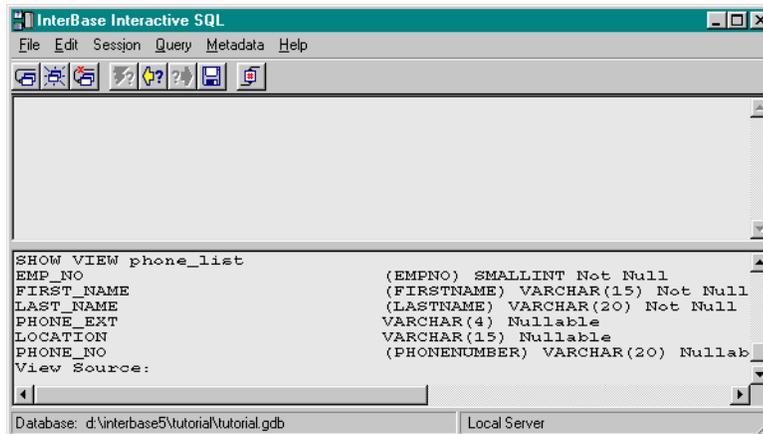
Creating Indexes

An index is based on one or more columns in a table. It orders the contents of the specified columns and stores that information on disk in order to speed up access to those columns. Although they improve the performance of data retrievals, indexes also take up disk space and can slow inserts and updates, so they are typically used on frequently queried columns. Indexes can also enforce uniqueness and referential integrity constraints.

InterBase automatically generates indexes on UNIQUE and PRIMARY KEY columns. See the *Data Definition Guide* for more information about constraints.

You use the CREATE INDEX statement to create an index. The simplified syntax is as follows:

```
CREATE INDEX name ON table (columns)
```

Optionally, you can add one or more of the ASCENDING, DESCENDING, or UNIQUE keywords following the CREATE INDEX keywords.

▶ **Create the *namex* index**

Define an index for the *Employee* table, by entering the following code:

```
CREATE INDEX namex ON Employee (last_name, first_name)
```

This statement defines an index called *namex* for the *last_name* and *first_name* columns in the *Employee* table.

Preventing duplicate index entries

To define an index that eliminates duplicate entries, include the UNIQUE keyword in CREATE INDEX. After a UNIQUE index is defined, users cannot insert or update values in indexed columns if the same values already exist there.

For unique indexes defined on multiple columns, such as *prodtypex* in the example below, the same value can be entered within individual columns, but the combination of values entered in all columns of the index must be unique for each row

You cannot create a UNIQUE index on columns that already contain non-unique values.

▶ **Create a UNIQUE index**

Create a unique index named *prodtypex*, on the *Project* table by entering the following:

```
CREATE UNIQUE INDEX prodtypex ON Project (product, proj_name)
```

Specifying index sort order

By default, SQL stores an index in ascending order. To make a descending sort on a column or group of columns more efficient, use the DESCENDING keyword to define the index.

▶ **Create the *budgetx* DESCENDING index**

Enter and execute the following code to create an index called *budgetx* that is in descending order:

```
CREATE DESCENDING INDEX budgetx ON Department (budget)
```

Modifying indexes

To change an index definition—which columns are indexed, sort order, or UNIQUE requirement—you must first drop the index and then create a new index.

▶ **Alter the *namex* index**

Begin by viewing the current definition of the *namex* index: choose **Metadata|Show**->Index and type namex in the Object Name field.

In the following steps, you redefine the *namex* index that you created earlier to include the UNIQUE keyword.

1. Enter and execute the following DROP INDEX statement:

```
DROP INDEX namex
```

2. Enter and execute the following line to redefine *namex* so that it includes the UNIQUE keyword:

```
CREATE UNIQUE INDEX namex ON Employee (last_name, first_name)
```

Time to back up If you have successfully altered the *Department* table definition, created the *phone_list* view, created the three indexes, and altered the *namex* index, this is a good time to back up your database to *Tutor3.gbk*.

Part III **Populating the Database**

In the preceding exercises, you created the structure of your database: domains, tables, a view, and three indexes. In the following exercises, you use the INSERT statement to *populate* (add data to) the database that you created in previous steps. Then you use UPDATE and DELETE statements to manipulate the data.

Inserting data

The INSERT statement is the mechanism by which you store one or more rows of data in an existing table. In its simplest form, the syntax is:

```
INSERT INTO table_name [(columns)] VALUES (values)
```

If you don't specify column names, InterBase inserts the supplied values into columns in the order in which they were defined, and there must be as many values as there are columns in the table. When you specify columns, you supply the values in the order you name the columns. Columns not specified are given default values or NULL values, depending on the column definitions.

The values supplied can be constants or can be calculated. In embedded SQL, they can also be variables.

An important variation of this syntax is one that allows you to add rows to a table by selecting rows from another table. The two tables must have columns occurring in the same order for this to work. The syntax for this form is:

```
INSERT INTO table_name (columns) SELECT columns FROM table_name WHERE conditions
```

See the *Language Reference* for a full description of INSERT.

▶ **Insert data using column values**

1. Enter and execute the following code to add a row to the *Country* table:

   ```
   INSERT INTO Country(country, currency) VALUES ('USA', 'Dollar')
   ```

   Reminder Anything you type inside the quotation marks is case sensitive.

2. Enter and execute the following line to add a row to the *Department* table:

   ```
   INSERT INTO Department
           (dept_no, department, head_dept, budget, location, phone_no)
       VALUES ('000', 'Corporate Headquarters', NULL, 1000000, 'Monterey',
           '(408) 555-1234')
   ```

   Notice that strings are all enclosed in single quotes, while numeric values are not. The department number and default phone number, for example, are strings, not numeric values.

3. The next row of data for the *Department* table is similar to the previous one. To simplify entry, click the Previous Query [⇦?] button. This redisplays the previous query in the SQL Statement Area.

The Previous Query button

4. Now substitute into the previous query so that it reads as follows and execute the statement.

```
INSERT INTO Department
        (dept_no, department, head_dept, budget, location, phone_no)
    VALUES ('100', 'Sales and Marketing', '000', 200000, 'San Francisco',
            '(415) 555-1234')
```

Notice that the new value for *head_dept* is a string, not a numeric value.

5. Check the accuracy of your insertions by entering and executing each of the following statements in turn. Examine the output to make sure it matches the instructions above.

```
SELECT * from Country
```

```
SELECT * from Department
```

In Part IV of this tutorial, you learn (much!) more about the important SELECT statement.

▶ **Read in the remaining data**

1. To read the remaining data into the *Country*, *Job*, *Department*, and *Employee* tables, open *Inserts.sql* in a text editor, make sure that the CONNECT statement has the correct information, and read it into the database using **File | Run an ISQL Script**.

2. Now enter and execute each following statement in turn to confirm that data has been entered into each table.

```
SELECT * FROM Country
```

There should be 14 entries in the *Country* table. If this one is correct, the others probably are, too. Now run three more SELECT statements. Remember, you must execute each one before proceeding to the next.

```
SELECT * FROM Job
SELECT * FROM Employee
SELECT * FROM Department
```

Time to back up If you have successfully entered three INSERTs and run the *Inserts.sql* script, this is a good time to back up your database to *Tutor4.gbk*.

Updating data

You use UPDATE statements to change values for one or more rows of data in existing tables.

Using UPDATE

A simple update has the following syntax:

```
UPDATE table
   SET column = value
   WHERE condition
```

The UPDATE statement changes values for columns specified in the SET clause; columns not listed in the SET clause are not changed. To update more than one column, list each column assignment in the SET clause, separated by a comma. The WHERE clause determines which rows to update. If there is no WHERE clause, all rows are updated.

For example, the following statement would increase the salary of salespeople by $2,000, by updating the *salary* column of the *Employee* table for rows where the value in the *job_code* column is "sales." (Don't do this yet.)

```
UPDATE Employee
   SET salary = salary + 2000
   WHERE job_code = 'Sales'
```

▶ **Executing and committing**

■ For the rest of this tutorial, execute each statement after entering it. You will no longer be explicitly instructed to do so.

■ In addition, execute a COMMIT statement after executing a DML statement (INSERT, DELETE, UPDATE, and SELECT). DDL statements—CREATE, ALTER, and DROP—don't need manual commits because you have enabled Auto Commit DDL in InterBase Windows ISQL's Session Basic Settings.

▶ **Update data in the *Employee* table**

To make a more specific update, make the WHERE clause more restrictive. Enter the following code to increase the salaries only of salespeople hired before January 1, 1992:

```
UPDATE Employee
   SET salary = salary + 2000
   WHERE job_code = 'Sales' AND hire_date < '01-JAN-1992'
```

A WHERE clause is not required for an update. If the previous statements did not include a WHERE clause, the update would increase the salary of all employees in the *Employee* table.

▶ **Run the *Updates.sql* script**

1. Open the *Updates.sql* file in a text editor and look it over. It contains UPDATE statements that set values for the *mngr_no* column in the *Department* table, it creates some salary history records for the *Employee* table, and it updates the *Customer* table by setting the status of two customers to "on hold" by entering an asterisk in the *on_hold* column. Close the file when you have finished examining it.

2. In InterBase Windows ISQL, choose **File | Run an ISQL script** and run the *Updates.sql* file. As always, choose Yes when asked if you want to commit previous work and No to saving the output to a file.

Time to back up If you have successfully run the *Updates.sql* script and performed the manual update, this is a good time to back up your database to *Tutor5.gbk*.

Updating to a NULL value

Sometimes data needs to be updated before all the new values are available. You can indicate unknown data by setting values to NULL. This works only if a column is *nullable*, meaning that it is not defined as NOT NULL.

Suppose that in the previous example, the department number of salespeople hired before 1992 is changing but the new number is not yet known. You would update salaries and department numbers as follows:

```
UPDATE Employee
   SET salary = salary + 2000, dept_no = NULL          Note: Don't do this yet!
   WHERE job_code = 'Sales'
   AND hire_date < '01-Jan-1992'
```

Using a subquery to update

The search condition of a WHERE clause can be a subquery. Suppose you want to change the manager of all employees in the same department as Katherine Young. One way to do this is to first determine Katherine Young's department number (don't do this yet):

```
SELECT dept_no FROM Employee
   WHERE full_name = 'Young, Katherine'
```

This query returns "623" as the department. Then, using 623 as the search condition in an UPDATE, you change the manager number of all the employees in the department with the following statement:

```
UPDATE Department
   SET mngr_no = 107                           Note: Don't do this yet!
   WHERE dept_no = '623'
```

A more efficient way to perform the update is to combine the two previous statements using a subquery. A subquery is one in which a SELECT clause is used within the WHERE clause to determine which rows to update.

▶ **Update *Department* using a subquery**

1. So that you can see the results of this exercise, begin by entering the following query to show you the manager number of Katherine's department before you make the update:

```
SELECT mngr_no FROM department WHERE dept_no = '623'
```

This returns 15. (If you select *first_name* and *last_name* from the *Employee* table where *emp_no* equals 15, you will see that the manager of department 623 is Katherine herself.)

2. Enter the following UPDATE statement with a subquery to simultaneously find out Katherine's department number and assign a new manager number to that department:

```
UPDATE Department
    SET mngr_no = 107
    WHERE dept_no = (SELECT dept_no FROM Employee
        WHERE full_name = 'Young, Katherine')
```

The rows returned by the SELECT statement within the parentheses are the rows that the UPDATE statement acts on.

3. Execute and commit the UPDATE statement you just entered.

4. Now run the query in step 1 again to see the change. The manager of department 623 is manager number 107, rather than 15.

5. This isn't a change we want to keep, so enter and execute the following statement to reinstate Katherine Young as manager of department 123:

```
UPDATE Department SET mngr_no = 15 WHERE dept_no = '623'
```

Deleting data

To remove one or more rows of data from a table, use the DELETE statement. A simple DELETE has the following syntax:

```
DELETE FROM table
    WHERE condition
```

As with UPDATE, the WHERE clause specifies a search condition that determines the rows to delete. Search conditions can be combined or can be formed using a subquery.

IMPORTANT   The DELETE statement does not require a WHERE clause. However, if you do not include a WHERE clause, you delete *all* the rows of a table.

▶ **Delete a row from *Sales* (and put it back)**

In this exercise, you first look to see what orders are older than a certain date. Then you delete those sales from the *Sales* table and check to see that they are gone.

1. In InterBase Windows ISQL, choose **File | Commit Work** to commit your work to date.

2. Enter the following SELECT statement to see what sales were ordered prior to 1992:

```
SELECT * FROM Sales WHERE order_date < '31-DEC-1991'
```

There should be only one order returned. Notice that the SELECT statement requires that you specify columns. You can also use "*" to specify all columns.

3. Enter the following SMALL DELETE statement. To make it easier, you can display the previous SELECT statement and substitute DELETE for "SELECT *". You can use either the Previous Statement 🔖 button or ⌷Ctrl⌷-P to display previous statements:

```
DELETE FROM Sales
     WHERE order_date < '31-DEC-1991'
```
Note: Don't do this yet!

Notice that the DELETE statement does not take any column specification. That's because it deletes all columns for the rows you have specified.

4. Now repeat your original SELECT query. There should be no rows returned.

5. Oops. You just realized that you didn't want to delete that data after all. Fortunately, you committed previous work before executing this statement, so choose **File | Rollback Work** and click OK at the prompt. This "undoes" all statements that were executed since the last Commit.

6. Perform the SELECT again to see that the deleted row is back.

Time to back up Now that you have created your database and its tables and finished inserting and updating data, this is a good time to back up your database to *Tutor6.gbk*.

Deleting more precisely

You can restrict deletions further by combining search conditions. For example, enter the following statement to delete records of everyone in the sales department hired before January 1, 1992:

```
DELETE FROM Employee
   WHERE job_code = 'Sales'
      AND hire_date < '01-Jan-1992'
```
Note: Don't do this yet!

You can try entering this statement, but you'll get an error because there's a foreign key column in the *Employee_project* table that references the *Employee* table. If you were to delete these rows, some values in the *Employee_project* table would no longer have matching values in the *Employee* table, violating the foreign key constraint that says any employee who has a project must also have an entry in the *Employee* table.

In addition, you can use subqueries to delete data, just as you use them to update. The following statement would delete all rows from the *Employee* table where the employees were in the same department as Katherine Young.

```
DELETE FROM Employee
      WHERE dept_no = (SELECT dept_no FROM Employee
          WHERE full_name = 'Young, Katherine')
```
Note: Don't do this yet!

Again, you cannot actually execute this statement because it would violate foreign key constraints on other tables.

Part IV **Retrieving Data**

The SELECT statement lies at the heart of SQL because it is how you retrieve the information you have stored. There is no use in creating and populating data structures if you cannot get the data out again in usable form. You've seen some simple forms of the SELECT statement in earlier exercises. In this part of the tutorial you get further practice with the SELECT statement.

Overview of SELECT

Part III presented the simplest form of the SELECT statement. The full syntax is much more complex. Take a minute to look at the entry for SELECT in the *Language Reference*. Much of SELECT's power comes from its rich syntax.

In this chapter, you learn a core version of the SELECT syntax:

```
SELECT [DISTINCT] columns
   FROM tables
   WHERE <search_conditions>
   [GROUP BY column [HAVING <search_condition>]]
   [ORDER BY <order_list>]
```

The SELECT syntax above has six main keywords. A keyword and its associated information is called a *clause*. The clauses above are:

Clause	Description
SELECT *columns*	Lists columns to retrieve
DISTINCT	Optional keyword that eliminates duplicate rows
FROM *tables*	Identifies the tables to search for values
WHERE *<search_conditions>*	Specifies the search conditions used to limit retrieved rows to a subset of all available rows
GROUP BY *column*	Groups rows retrieved according the value of the specified column
HAVING *<search_conditions>*	Specifies search condition to use with GROUP BY clause
ORDER BY *<order_list>*	Orders the output of a SELECT statement by the specified columns

TABLE 2   Seven important SELECT clauses

You have already used SELECT statements to retrieve data from single tables. However, SELECT can also retrieve data from multiple tables, by listing the table names in the FROM clause, separated by commas.

▶ **Retrieve data from two tables at once**

In this example, you want to know the name and employee number of the person who manages the Engineering department. The *Department* table contains the manager number (*mngr_no*) for each department. That manager number matches an employee number (*emp_no*) in the *Employee* table, which has a first and last name in the record with the employee number. You link the corresponding records of the two tables by using the WHERE clause to specify the foreign key of one (*mngr_no*) as equal to the primary key (*emp_no*) of the other. Since the primary key is guaranteed to be unique, you are specifying a unique row in the second table. Neither key has to be part of the SELECT clause. In this example, the referenced primary key is part of the SELECT clause but the foreign key is not.

To get the information described above, execute the following SQL statement in InterBase Windows ISQL:

```
SELECT department, last_name, first_name, emp_no
   FROM Department, Employee
   WHERE department = 'Engineering' AND mngr_no = emp_no
```

This statement retrieves the following information:

```
DEPARTMENT            LAST_NAME        FIRST_NAME   EMP_NO
============== ================== =============== ======
Engineering              Nelson           Robert      2
```

Removing duplicate rows with DISTINCT

Columns often contain duplicate entries (assuming that they do not have PRIMARY KEY or UNIQUE constraints on them. Sometimes you want to see only one instance of each value in a column. The DISTINCT keyword gives you exactly that.

▶ **Select one of each**

1. Suppose you want to retrieve a list of all the valid job codes in the TUTORIAL database. Begin by entering this query:

```
SELECT job_code FROM Job
```

As you can see, the results of this query are rather long, and some job codes are repeated a number of times. What you really want is a list of job codes where each value returned is distinct from the others. To eliminate duplicate values, use the DISTINCT keyword.

2. Re-enter the previous query with the DISTINCT keyword:

```
SELECT DISTINCT job_code FROM Job
```

This produces the desired results: each job code is listed only once in the results.

3. What happens if you specify another column when using DISTINCT? Enter the following SELECT statement:

```
SELECT DISTINCT job_code, job_grade FROM Job
```

This query returns:

```
job_code job_grade
======== =========
Accnt           4
Admin           4
Admin           5
CEO             1
CFO             1
Dir             2
Doc             3
Doc             5
Eng             2
Eng             3
Eng             4
Eng             5
. . . (21 rows total)
```

DISTINCT applies to all columns listed in a SELECT statement. In this case, duplicate job codes are retrieved. However, DISTINCT treats the job code and job grade together, so the *combination* of values is distinct.

Using the WHERE clause

The WHERE clause follows the SELECT and FROM clauses. It must precede the ORDER BY clause if one is used. The WHERE clause tests data to see whether it meets certain conditions, and the SELECT statement returns only the rows that meet the WHERE condition. The WHERE clause lies at the heart of database usage, because it is the point at which you state exactly what you want. It seems complex at first glance, but the complexity exists to allow you to be precise in your requests for data.

▶ **Using WHERE**

1. Enter the following statement to return only rows for which "Green" is the value in the *last_name* column.

```
SELECT last_name, first_name, phone_ext
    FROM Employee
    WHERE last_name = 'Green'
```

The query should return one row:

```
Green                   T.J.                218
```

2. Now display the statement again (use the Previous 🔁 button or Ctrl-P) and change the equal sign to a greater than sign. This retrieves rows for which the last name is alphabetically greater than (after) "Green." There should be 29 rows.

Something extra To make the results more readable, execute the last query once again, but add an ORDER BY clause. This is just a preview: the ORDER BY clause is discussed starting on page 52.

```
SELECT last_name, first_name, phone_ext
    FROM Employee
    WHERE last_name > 'Green'
    ORDER BY last_name
```

Search conditions

The text following the WHERE keyword is called a *search condition*, because a SELECT statement searches for rows that meet the condition. Search conditions consist of a **column name** (such as "last_name"), an **operator** (such as "="), and a **value** (such as "Green"). Thus, WHERE clauses have the following general form:

```
WHERE column_name operator value
```

In general, *column* is the column name in the table being queried, *operator* is a comparison operator (Table 3)*,* and *value* is a value or a range of values compared against the column. Table 4 describes the kinds of values you can specify.

Comparison operators

Search conditions use the following operators. Note that for two-character operators, there is no space between the operators.

Operator	Description
Comparison operators	Used to compare data in a column to a value in the search condition. Examples include <, >, <=, >=, =, !=, and < >. Other operators include BETWEEN, CONTAINING, IN, IS NULL, LIKE, and STARTING WITH.
Arithmetic operators	Used to calculate and evaluate search condition values. The operators are +, −, *, and /.
Logical operators	Used to combine search conditions or to negate a condition. The keywords are NOT, AND, and OR.

TABLE 3   Search condition operators

Search condition values

The values in a search condition can be literal, or they can be calculated (derived). In addition, the value can be the return value of a subquery. A subquery is a nested SELECT statement.

Values that are text literals must be placed in quotes. The approaching standard will require single quotes. Currently (1998), double quotes are also allowed. Numeric literals must *not* be quoted.

IMPORTANT   String comparisons are case sensitive.

Types of Values	Description
Literal values	Numbers and text strings whose value you want to test literally; for example, the number 1138 or the string "Smith"
Derived values	Functions and arithmetic expressions; for example salary * 2 or last_name ‖ first_name
Subqueries	A nested SELECT statement that returns one or more values. The returned values are used in testing the search condition.

TABLE 4   Types of values used in search conditions

When a row is compared to a search condition, one of three values is returned:

- *True*: A row meets the conditions specified in the WHERE clause.

- *False*: A row does not meet the conditions specified in the WHERE clause.

- *Unknown*: A field in the WHERE clause contains an NULL state that could not be evaluated.

▶ **Find the deadbeats**

Execute the following SELECT statement into the SQL Statement Area of InterBase Windows ISQL to query the *Sales* table for all the customers who ordered before January 1, 1994, received their shipment, and still haven't paid. Notice that there are three search conditions in the WHERE clause, which are joined together with the AND operator.

```
SELECT * from Sales
   WHERE order_date < '1-JAN-1994' AND order_status = 'shipped' AND paid = 'n'
```

You should get two rows, one for PO number V93F3088 and one for PO number V93N5822.

Negation

You can negate any expression with the negation operators !, ^, and ~. These operators are all synonyms for NOT.

▶ **Find what's NOT**

Suppose you just want to find what customers are not in the United States. Execute the following SELECT statement:

```
SELECT customer, country FROM Customer
   WHERE NOT country = 'USA'
```

You should get a list of ten customers.

There are other ways to achieve exactly this result. To prove to yourself that these all produce the same results as the previous query, execute each of the following forms of it:

```
SELECT customer, country FROM Customer
   WHERE country != 'USA'
```

```
SELECT customer, country FROM Customer
   WHERE country ~= 'USA'
```

```
SELECT customer, country FROM Customer
   WHERE country ^= 'USA'
```

Pattern matching

Besides comparing values, search conditions can also test character strings for a particular pattern. If data is found that matches a given pattern, the row is retrieved.

Wildcards Use a percent sign (%) to match zero or more characters. Use an underscore (_) to match a single character.

Table 5 gives examples of some common patterns. Only CONTAINING is not case sensitive.

WHERE	Matches
last_name LIKE '%q%'	Last names containing at least one "q"
last_name STARTING WITH 'Sm'	Last names beginning with the letters "Sm."
last_name CONTAINING 'q'	Last name contains at least one "q," either uppercase or lowercase.
last_name BETWEEN 'M' AND 'T'	Last name beginning with letters M through S

TABLE 5   Pattern matching examples

▶ **Find what's LIKE a value**

1. LIKE is case sensitive and takes wildcards. Execute this statement to find all employees whose last name ends in "an":

```
SELECT last_name, first_name, emp_no FROM Employee
    WHERE last_name LIKE '%an'
```

The result set should look like this:

```
LAST_NAME            FIRST_NAME      EMP_NO
=================== =============== ======

Ramanathan           Ashok               45
Steadman             Walter              46
```

2. Now enter the following SELECT statement to find employees whose last names begin with "M", have exactly two more characters and then a "D", followed by anything else in the remainder of the name:

```
SELECT last_name, first_name, emp_no FROM Employee
    WHERE last_name LIKE 'M__D%'
```

This returns

```
MacDonald            Mary S.             85
```

but would ignore names like McDonald.

3. The "%" matches zero or more characters. The following query returns rows for last names Burbank, Bender, and Brown.

```
SELECT last_name, first_name, emp_no FROM Employee
    WHERE last_name LIKE 'B%r%'
```

This returns:

```
LAST_NAME            FIRST_NAME      EMP_NO
=================== =============== ======
Burbank              Jennifer M.         71
Bender               Oliver H.          105
Brown                Kelly              109
```

▶ **Find things STARTING WITH**

The STARTING WITH operator tests whether a value starts with a particular character or sequence of characters. STARTING WITH is case sensitive, but does not support wildcard characters.

1. Execute the following statement to retrieve two employee last names that start with "Le":

```
SELECT last_name, first_name FROM Employee
    WHERE last_name STARTING WITH 'Le'
```

The result set is:

```
LAST_NAME            FIRST_NAME
=================== ===============
Lee                  Terri
Leung                Luke
```

2. To negate the STARTING WITH operator, precede it with the logical operator NOT. (*Note*: That's a "one" in the quotes at the end of the statement.)

```
SELECT dept_no, department, location from department
    WHERE dept_no NOT starting with '1'
```

This query should return the following 10 rows:

```
DEPT_NO DEPARTMENT                 LOCATION
======= ========================= ===============
000     Corporate Headquarters    Monterey
600     Engineering               Monterey
900     Finance                   Monterey
620     Software Products Div.     Monterey
621     Software Development       Monterey
622     Quality Assurance          Monterey
623     Customer Support           Monterey
670     Consumer Electronics Div.  Burlington, VT
671     Research and Development    Burlington, VT
672     Customer Services          Burlington, VT
```

▶ **Find something CONTAINING a value**

The CONTAINING operator is similar to STARTING WITH, except it matches strings containing the specified string *anywhere* within the string. CONTAINING is *not* case sensitive and does not support or require wildcards.

1. Execute the following statement to find last names that have a "g" or "G" anywhere in them.

```
SELECT last_name, first_name FROM Employee
    WHERE last_name CONTAINING 'G'
```

You should get the following result set:

```
LAST_NAME            FIRST_NAME
==================== ===============
Young                Bruce
Young                Katherine
Phong                Leslie
Leung                Luke
Page                 Mary
Glon                 Jacques
Green                T.J.
Montgomery           John
Guckenheimer         Mark
```

2. Now execute the same query, except substitute a lower-case "g." You should get exactly the same result set.

```
SELECT last_name, first_name FROM Employee
    WHERE last_name CONTAINING 'g'
```

Testing for an unknown value

Another type of comparison tests for the absence or presence of a value. Use the IS NULL operator to test whether a value is unknown. To test for the *presence* of any value, use IS NOT NULL.

▶ **Test for NULL**

1. Execute the following query to retrieve the names of employees who do not have phone extensions:

   ```
   SELECT last_name, first_name, phone_ext FROM Employee
       WHERE phone_ext IS NULL
   ```

   The query should return rows for last names Sutherland, Glon, and Osborne.

2. Now execute the statement using IS NOT NULL to retrieve the names of employees who *do* have phone extensions:

   ```
   SELECT last_name, first_name, phone_ext FROM Employee
       WHERE phone_ext IS NOT NULL
   ```

   There should be 39 rows in the result set.

Comparing against a range or list of values

The previous sections present comparison operators that test against a single value. The BETWEEN and IN operators test against multiple values.

BETWEEN tests whether a value falls within a range. The BETWEEN operator is case-sensitive and does not require wildcards.

▶ **Find something BETWEEN values**

1. Execute the following query to find all the last names that start with letters between C and H. Notice that the query does not include names that begin with the final value ("H"). This is because BETWEEN finds values that are less than or equal to the terminating value. A name that begins with the letter but includes other letters is greater than H. If there were someone whose last name was just "H", the query would return it.

   ```
   SELECT last_name, first_name FROM Employee
       WHERE last_name BETWEEN 'C' AND 'H'
   ```

   The result set is:

   ```
   LAST_NAME            FIRST_NAME
   ==================== ===============
   Forest               Phil
   Fisher               Pete
   De Souza             Roger
   Cook                 Kevin
   Ferrari              Roberto
   Glon                 Jacques
   Green                T.J.
   Guckenheimer         Mark
   ```

2. To demonstrate that BETWEEN is case sensitive, repeat the previous query using lower-case letters. There are no names returned.

```
SELECT last_name, first_name FROM Employee
    WHERE last_name BETWEEN 'a' AND 'p'
```

3. Execute the following query to retrieve names of employees whose salaries are between $62,000 and $98,000, inclusive:

```
SELECT last_name, first_name, salary FROM Employee
    WHERE salary BETWEEN 60000 AND 80000
    ORDER BY salary
```

The result set should return 12 rows, with salaries that include both the low figure and the high figure in the range. (See page 52 for a discussion of the ORDER BY clause.)

LAST_NAME	FIRST_NAME	SALARY
Bishop	Dana	60000.00
Johnson	Scott	60000.00
Young	Katherine	60000.00
Hall	Stewart	62000.00
De Souza	Roger	62000.00
Johnson	Leslie	62000.00
Leung	Luke	66000.00
Ramanathan	Ashok	72000.00
Forest	Phil	72000.00
Fisher	Pete	73000.00
Weston	K. J.	77000.00
Papadopoulos	Chris	80000.00

▶ **Find what's IN**

The IN operator searches for values matching one of the values in a list. The values in the list must be separated by commas, and the list must be enclosed in parentheses. Use NOT IN to search for values that do not occur in a set

Execute the following query to retrieve the names of all employees in departments 120, 600, and 623:

```
SELECT dept_no, last_name, first_name FROM Employee
    WHERE dept_no IN (120, 600, 623)
    ORDER BY dept_no, last_name
```

The returns the following result set:

DEPT_NO	LAST_NAME	FIRST_NAME
120	Bennet	Ann
120	Stansbury	Willie
600	Brown	Kelly
600	Nelson	Robert
623	De Souza	Roger
623	Johnson	Scott
623	Parker	Bill
623	Phong	Leslie
623	Young	Katherine

Logical operators

You can specify multiple search conditions in a WHERE clause by combining them with the logical operators AND or OR.

▶ **Find rows that match multiple conditions**

When AND appears between search conditions, both conditions must be true for a row to be retrieved. For example, execute this query to find employees in a particular department who were hired before January 1, 1992:

```
SELECT dept_no, last_name, first_name, hire_date
   FROM Employee
   WHERE dept_no = 623 AND hire_date > '01-Jan-1992'
```

It should return two rows, one each for employees Parker and Johnson.

▶ **Find rows that match at least one condition**

Use OR between search conditions where you want to retrieve rows that match at least one of the conditions.

1. Click the Previous Query ⬕ button to display your last query. Change AND to OR and execute the new query. Notice that the results are dramatically different; the query returns 25 rows.

2. As a more likely example of the OR operator, execute the following query to find customers who are in either Japan or Hong Kong:

```
SELECT customer, cust_no, country FROM Customer
    WHERE country = 'Japan' OR country = 'Hong Kong'
```

The result set should look like this:

```
CUSTOMER                     CUST_NO COUNTRY
========================= =========== ================
DT Systems, LTD.               1005 Hong Kong
MPM Corporation                1010 Japan
```

Controlling the order of evaluation

When entering compound search conditions, you must be aware of the order of evaluation of the conditions. Suppose you want to retrieve employees in department 623 or department 600 who have a hire date later than January 1, 1992.

▶ **Try a compound condition**

Try executing this query:

```
SELECT last_name, first_name, hire_date, dept_no
   FROM Employee
   WHERE dept_no = 623 OR dept_no = 600 AND hire_date > '01-JAN-1992'
```

As you can see, the results include employees hired earlier than you want:

```
last_name           first_name       hire_date   dept_no
=================== ================ =========== =======
Young               Katherine        14-JUN-1990 623
De Souza            Roger            18-FEB-1991 623
Phong               Leslie            3-JUN-1991 623
Brown               Kelly             4-FEB-1993 600
Parker              Bill              1-JUN-1993 623
Johnson             Scott            13-SEP-1993 623
```

This query produces unexpected results because AND has *higher precedence* than OR. This means that the expressions on either side of AND are tested before those associated with OR. In the example as written, the search conditions are interpreted as follows:

```
( WHERE dept_no = 623 )
   OR
( WHERE dept_no = 600 AND hire_date > '01-JAN-1992' )
```

The restriction on the hire date applies only to the second department. Employees in department 623 are listed regardless of hire date.

▶ **Use a compound condition successfully**

Use parentheses to override normal precedence. In the exercise below, place parentheses around the two departments so they are tested against the AND operator as a unit. Redisplay your last query and add parentheses, so that your query is interpreted correctly:

```
SELECT last_name, first_name, hire_date, dept_no
   FROM Employee
   WHERE (dept_no = 623 OR dept_no = 600)
   AND hire_date > '01-JAN-1992'
```

This displays the results you want:

```
last_name           first_name       hire_date   dept_no
=================== ================ =========== =======
Brown               Kelly             4-FEB-1993 600
Parker              Bill              1-JUN-1993 623
Johnson             Scott            13-SEP-1993 623
```

Order of precedence is not just an issue for AND and OR. All operators are defined with a precedence level that determines their order of interpretation. You can study precedence levels in detail by reading any number of books about SQL, but in general, the following rule of thumb is all you need to remember.

TIP   Always use parentheses to group operations in complex search conditions.

Using Subqueries

Subqueries are a special case of the WHERE clause, but they are an important tool and deserve a discussion of their own.

Recall that in a WHERE clause, you provide a column name, a comparative operator, and a value. WHERE tests the column contents against the value using the operator. You can use a SELECT statement in place of the value portion of a WHERE clause. This internal SELECT clause is the *subquery*. InterBase executes the SELECT subquery and uses its result set as the value for the WHERE clause.

Suppose, for example, that you want to retrieve a list of employees who work in the same country as a particular employee whose ID is 144. If you don't use a subquery, you would first need to find out what country this employee works in:

```
SELECT job_country FROM Employee
   WHERE emp_no = 144
```

This query returns "USA." With this information, you would issue a second query to find a list of employees in the USA, the same country as employee number 144:

```
SELECT emp_no, last_name FROM Employee
   WHERE job_country = 'USA'
```

Using a subquery permits you to perform both queries in a single statement.

▶ **Use a subquery to find a single item**

You can obtain the same result by combining the two queries:

```
SELECT emp_no, last_name FROM Employee
   WHERE job_country = (SELECT job_country FROM Employee WHERE emp_no = 144)
```

In this case, the subquery retrieves a single value, "USA." The main query interprets "USA" as a value to be tested by the WHERE clause. The subquery must return a single value because the WHERE clause is testing for a single value ("="); otherwise the statement produces an error.

The result set for this query is a list of 33 employee numbers and last names. These are the employees who work in the same country as employee number 144.

Multiple-result subqueries

If a subquery returns more than one value, the WHERE clause that contains it must use an operator that tests against more than one value. IN is such an operator.

▶ **Use a subquery to find a collection of items**

Execute the following example to retrieve the last name and department number of all employees whose salary is equal to that of someone in department 623. It uses a subquery that returns all the salaries of employees in department 623. The main query selects each employee in turn and checks to see if the associated salary is in the result set of the subquery.

```
SELECT last_name, dept_no FROM Employee
   WHERE salary IN (SELECT salary FROM Employee WHERE dept_no = 623)
```

The result set should look like this:

```
LAST_NAME            DEPT_NO
==================== =======
Johnson               180
Hall                  900
Young                 623
De Souza              623
Stansbury             120
Phong                 623
Bishop                621
Parker                623
Johnson               623
Montgomery            672
```

Conditions for subqueries

The following table summarizes the operators that compare a value on the left of the operator to the results of a subquery to the right of the operator:

Operator	Purpose
ALL	Returns true if a comparison is true for all values returned by a subquery.
ANY or SOME	Returns true if a comparison is true for at least one value returned by a subquery.
EXISTS	Determines if a value exists in *at least one* value returned by a subquery.
SINGULAR	Determines if a value exists in *exactly one* value returned by a subquery.

TABLE 6   InterBase comparison operators that require subqueries

Using ALL

The IN operator tests only against the *equality* of a list of values. What if you want to test some relationship other than equality? For example, suppose you want to find out who earns more than the people in department 623. Enter the following query:

```
SELECT last_name, salary FROM Employee
   WHERE salary > ALL
   (SELECT salary FROM Employee WHERE dept_no = 623)
```

The result set should look like this:

```
LAST_NAME                           SALARY
===================      ==================
Nelson                             98000.00
Young                              90000.00
Lambert                            95000.00
Forest                             72000.00
Weston                             77000.00
Papadopoulos                       82000.00
Fisher                             75000.00
Ramanathan                         72000.00
Steadman                          116100.00
Leung                              66000.00
Sutherland                         96800.00
MacDonald                         111262.50
Bender                            212850.00
Cook                              111262.50
Ichida                           6000000.00
Yamamoto                         7480000.00
Ferrari                         99000000.00
Glon                              390500.00
Osborne                           110000.00
```

This example uses the ALL operator. The statement tests against *all* values in the subquery and retrieves the row if the salary is greater. The manager of department 623 can use this output to see which company employees earn more than his or her employees.

Using ANY, EXISTS, *and* SINGULAR

Instead of testing against all values returned by a subquery, you can rewrite the example to test for at least one value:

```
SELECT last_name, salary FROM Employee
   WHERE salary > ANY(SELECT salary FROM Employee WHERE dept_no = 623)
```

This statement retrieves 34 rows for which *salary* is greater than any of the values from the subquery. The ANY keyword has a synonym, SOME. The two are interchangeable.

Two other subquery operators are EXISTS and SINGULAR.

- For a given value, EXISTS tests whether *at least one* qualifying row meets the search condition specified in a subquery. EXISTS returns either true or false, even when handling NULL values.

- For a given value, SINGULAR tests whether *exactly one* qualifying row meets the search condition specified in a subquery.

Using aggregate functions

SQL provides aggregate functions that calculate a single value from a group of values. A group of values is all data in a particular column for a given set of rows, such as the job code listed in all rows of the JOB table. Aggregate functions may be used in a SELECT clause, or anywhere a value is used in a SELECT statement.

The following table lists the aggregate functions supported by InterBase:

Function	What It Does
AVG(*value*)	Returns the average value for a group of rows
COUNT(*value*)	Counts the number of rows that satisfy the WHERE clause
MIN(*value*)	Returns the minimum value in a group of rows
MAX(*value*)	Returns the maximum value in a group of rows
SUM(*value*)	Adds numeric values in a group of rows

TABLE 7   Aggregate functions supported by InterBase

▶ **Practice with aggregate functions**

1. Suppose you want to know how many different job codes are in the *Job* table. Enter the following statement:

```
SELECT COUNT(job_code) FROM Job
```

The result set should look like this:

```
      COUNT
===========
         31
```

However, this is not what you want, because the query included duplicate job codes in the count.

2. To count only the unique job codes, use the DISTINCT keyword as follows:

```
SELECT COUNT(DISTINCT job_code) FROM Job
```

This produces the correct result:

```
      COUNT
===========
         14
```

3. Enter the following to retrieve the average budget of departments from the *Department* table:

```
SELECT AVG(budget) FROM Department
```

The result set should look like this:

```
AVG
=====================
733809.52
```

4. A single SELECT can retrieve multiple aggregate functions. Enter the following statement to retrieve the number of employees, the earliest hire date, and the total salary paid to all employees:

```
SELECT COUNT(emp_no), MIN(hire_date), SUM(salary)
    FROM Employee
```

The result set should look like this:

```
      COUNT          MIN                    SUM
=========== ===========   =====================
         42  28-DEC-1988           115398775.00
```

(The value in the *sum* column may vary, depending on which exercises you have done and whether you have done some of them more than once.)

IMPORTANT   If a value involved in an aggregate calculation is NULL or unknown, the function ignores the entire row to prevent wrong results. For example, when calculating an average over fifty rows, if ten rows contain a NULL value, then the average is taken over forty values, not fifty.

5. To see for yourself that aggregate functions ignore NULL rows, perform the following test: first, look at all the rows in the *Department* table:

```
SELECT dept_no, mngr_no FROM Department
```

Notice that there are 21 rows, but four of them have NULLs in the *mngr_no* column.

6. Now count the rows in *mngr_no*:

```
SELECT COUNT(mngr_no) FROM Department
```

The result is 17, not 21. COUNT did not count the NULL rows.

Grouping and ordering query results

Rows are not stored in any particular order in a database. So when you execute a query, you may find that the results are not organized in any useful way. The ORDER BY clause lets you specify how the returned rows should be ordered. You can use the GROUP BY clause to group the results of aggregate functions.

Using ORDER BY to arrange rows

You can use the ORDER BY clause to organize the data that is returned from your queries. You can specify one or more columns by name or by ordinal number. The syntax of the ORDER BY clause is:

```
ORDER BY [col_name | int] [ASC[ENDING] | DESC[ENDING]] [, …]
```

Notice that you can specify more than one column. As an alternative to naming the columns, you can provide an integer that references the order in which you named the columns in the query. The second column that you named in the SELECT can be referenced as 2.

By default, InterBase uses ASCENDING order, so you only need to specify the order if you want it to be DESCENDING.

▶ **Practice with ORDER BY**

1. Execute the following query (you did this one earlier when you worked with conditions for subqueries).

```
SELECT cust_no, total_value FROM Sales
    WHERE total_value > 10000
```

There's no particular order to the returned rows. The result set should look like this:

```
 CUST_NO       TOTAL_VALUE
===========   ===========
    1010         18000.40
    1012        450000.49
    1001         60000.00
    1006        399960.50
    1008         16000.00
```

2. Execute the same query, but order the results by the *cust_no* column:

```
SELECT cust_no, total_value FROM Sales
    WHERE total_value > 10000
    ORDER BY cust_no
```

Notice that the result set now has the *cust_no* column in ascending order. Ascending order is the default.

3. Order the result set by the total value of the sales:

```
SELECT cust_no, total_value FROM Sales
    WHERE total_value > 10000
    ORDER BY total_value
```

4. Execute the query above, but order the result set by the descending order of the *total_value* column:

```
SELECT cust_no, total_value FROM Sales
    WHERE total_value > 10000
    ORDER BY total_value DESC
```

5. To see the effect of ordering by more than one column, execute the following query:

```
SELECT last_name, first_name, phone_ext FROM Employee
    ORDER BY last_name DESC, first_name
```

Notice that there are 42 rows with the last names are in descending order, as requested, and the first names in ascending order, the default.

Using the GROUP BY clause

You use the optional GROUP BY clause to organize data retrieved from aggregate functions. When you issue a query (a SELECT statement) that has both aggregate (AVG, COUNT, MIN, MAX, or SUM) and non-aggregate columns, you *must* use GROUP BY to group the result set by each of the nonaggregate columns. The three following rules apply:

- Each column from which you are doing a nonaggregate SELECT must appear in the GROUP BY clause
- The GROUP BY clause can reference only columns that appear in the SELECT clause
- Each SELECT clause in a query can have only one GROUP BY clause

A group is defined as the subset of rows that match a distinct value in the columns of the GROUP BY clause.

▶ **Group the result set of aggregate functions**

1. Execute the following query to find out how many employees there are in each country:

```
SELECT COUNT(emp_no), job_country FROM Employee
    GROUP BY job_country
```

The result set should look like this:

```
       COUNT JOB_COUNTRY
========== ===============

           1 Canada
           3 England
           1 France
           1 Italy
           2 Japan
           1 Switzerland
          33 USA
```

Using the HAVING clause

Just as a WHERE clause reduces the number of rows returned by a SELECT clause, the HAVING clause can be used to reduce the number of rows returned by a GROUP BY clause. Like the WHERE clause, a HAVING clause has a search condition. However, in a HAVING clause, the search condition typically corresponds to an aggregate function used in the SELECT clause.

▶ **Control your query with GROUP BY and HAVING**

Issue the following query to list the departments that have an average salary of over $60,000 and order the result set by department.

```
SELECT Department, AVG(budget) FROM department
   GROUP BY Department
   HAVING AVG(budget) >60000
   ORDER BY Department
```

The result set should look like this:

```
DEPARTMENT                             AVG
======================== ======================
Consumer Electronics Div.         1150000.00
Corporate Headquarters            1000000.00
Customer Services                  850000.00
Customer Support                   650000.00
Engineering                       1100000.00
European Headquarters              700000.00
Field Office: Canada               500000.00
Field Office: East Coast           500000.00
Field Office: France               400000.00
Field Office: Italy                400000.00
Field Office: Japan                500000.00
Field Office: Singapore            300000.00
Field Office: Switzerland          500000.00
Finance                            400000.00
Marketing                         1500000.00
Pacific Rim Headquarters           600000.00
Quality Assurance                  300000.00
Research and Development           460000.00
Sales and Marketing               2000000.00
Software Development               400000.00
Software Products Div.            1200000.00
```

Ordering by an aggregate column

But what if you want to list the result set by the average salary in the previous query? In that query, it wouldn't work to say "ORDER BY salary" because the query generates a two-column result set in which the first column is named "department" but the second column, which is generated by the aggregate function, doesn't have a name. The ORDER BY clause is actually referencing the columns of the result set. In order to request that the result set be ordered by the results of an aggregate function, you must reference the ordinal column number. (Look back at the ORDER BY syntax on page 52 and notice that it begins "ORDER BY [*col_name* | int]". You must reference the column by its integer number.

▶ **Order result set by the results of an aggregate function**

Display your last query, but change the ORDER BY clause as follows, to order by the second column:

```
SELECT department, AVG(budget) FROM department
   GROUP BY Department
   HAVING AVG(budget) >60000
   ORDER BY 2
```

Now the result set should have the second column in ascending order.

```
DEPARTMENT                                       AVG
======================== ======================
Field Office: Singapore           300000.00
Quality Assurance                 300000.00
Finance                           400000.00
Field Office: Italy               400000.00
Field Office: France              400000.00
Software Development              400000.00
Research and Development          460000.00
Field Office: Japan               500000.00
Field Office: Canada              500000.00
Field Office: East Coast          500000.00
Field Office: Switzerland         500000.00
Pacific Rim Headquarters          600000.00
Customer Support                  650000.00
European Headquarters             700000.00
Customer Services                 850000.00
Corporate Headquarters           1000000.00
Engineering                      1100000.00
Consumer Electronics Div.        1150000.00
Software Products Div.           1200000.00
Marketing                        1500000.00
Sales and Marketing              2000000.00
```

Joining tables

Joins enable a SELECT statement to retrieve data from two or more tables in a database. The tables are listed in the FROM clause. The optional ON clause can reduce the number of rows returned, and the WHERE clause can further reduce the number of rows returned.

From the information in a SELECT that describes a join, InterBase builds a table that contains the results of the join operation, the *results table*, sometimes also called a *dynamic* or *virtual table*.

InterBase supports two types of joins: inner joins and outer joins.

Inner joins link rows in tables based on specified join conditions and return only those rows that match the join conditions. If a joined column contains a NULL value for a given row, that row is not included in the results table. Inner joins are the more common type because they restrict the data returned and show a clear relationship between two or more tables.

Outer joins link rows in tables based on specified join conditions but return rows whether they match the join conditions or not. Outer joins are useful for viewing joined rows in the context of rows that do not meet the join conditions.

Correlation names

Once you begin to query multiple tables, it becomes important to identify unambiguously what table each column is in. The standard syntax for doing this is to state the table name followed by a period and the column name:

```
table_name.col_name
```

In complex queries, this can get very tedious, so InterBase permits you to state a shorter version of the table name in the FROM clause of a join. This short name is called a *correlation name* or an *alias*. You will see many examples of correlation names in the next few pages. The form is as follows:

```
SELECT a.col, b.col FROM table_1 a, table_2 b
   ON a.some_col = b.some_col
   WHERE a.conditional_col <condition>
   …
```

Notice the FROM clause, where *table_1* is given the correlation name of *a* and *table_2* is named *b*. These abbreviated names are used even in the initial select list.

IMPORTANT   If you include a subquery in a join, you must assign new correlation names to any tables that appeared in the main query.

Inner joins

There are three types of inner joins:

- *Equi-joins* link rows based on common values or equality relationships in the join columns.

- Joins that link rows based on comparisons other than equality in the join columns. There is not an officially recognized name for these joins, but for simplicity's sake they can be categorized as *comparative* joins, or *non-equi-joins*.

- *Reflexive* or *self-joins* compare values within a column of a single table.

  To specify a SELECT statement as an inner join, list the tables to join in the FROM clause, and list the columns to compare in the ON clause. Use the WHERE clause to restrict which rows are retrieved. The simplified syntax is:

```
SELECT <columns> | *
   FROM left_table [INNER] JOIN right_table
       ON left_table.col = right_table.col
   [WHERE <searchcondition>]
```

There are several things to note about this syntax:

- The INNER keyword is optional because INNER is the default join type. If you want to perform an outer join, you must state the OUTER keyword explicitly.

- The FROM statement often specifies the correlation names

  ```
  FROM table1 t1 JOIN table2 t2
  ```

- The operator in the ON clause doesn't have to be equality. It can be any of the comparison operators, such as !=, >, >=, or <>.

▶ **Perform inner joins**

1. Enter the following query to list all department managers and their departments where the manager earns more than 80,000. (This isn't stated as dollars because some of the employee salaries are in other currencies.)

```
SELECT D.department, D.mngr_no
    FROM Department D INNER JOIN Employee E
    ON D.mngr_no = E.emp_no
        WHERE E.salary > 80000
    ORDER BY D.department
```

The result set should look like this:

```
DEPARTMENT                 MNGR_NO
========================= =======
Consumer Electronics Div.    107
Corporate Headquarters       105
Customer Support             107
Engineering                    2
Field Office: Canada          72
Field Office: France         134
Field Office: Italy          121
Field Office: Japan          118
Field Office: Switzerland    141
Finance                       46
Sales and Marketing           85
```

2. The next query uses a subquery to display all departments and department managers where the manager's salary is at least 20% of a department's total salary:

```
SELECT D.department, D.mngr_no, E.salary
    FROM Department D JOIN Employee E
    ON D.mngr_no = E.emp_no
    WHERE E.salary*5 >= (SELECT SUM(S.salary) FROM Employee S
            WHERE D.dept_no = S.dept_no)
    ORDER BY D.department
```

The subquery sums the salaries for one department at a time and hands out the result to be compared to the manager's salary (multiplied by 5).

The result set should look like this:

```
DEPARTMENT                 MNGR_NO              SALARY
========================= ======= =====================
Consumer Electronics Div.    107           111262.50
Corporate Headquarters       105           212850.00
Customer Services             94            54000.00
Customer Support             107           111262.50
Engineering                    2            98000.00
European Headquarters         36            34000.00
Field Office: Canada          72            96800.00
Field Office: East Coast      11            77000.00
Field Office: France         134           390500.00
Field Office: Italy          121         99000000.00
Field Office: Japan          118          7480000.00
Field Office: Switzerland    141           110000.00
Finance                       46           116100.00
Pacific Rim Headquarters      34            59000.00
Quality Assurance              9            72000.00
Research and Development       20            80000.00
Sales and Marketing           85           111262.50
```

Note Joins are not limited to two tables. There is theoretically no limit to how many tables can be joined in one statement, although on the practical level of time and resources, 16 is usually considered the workable maximum.

Outer joins

Outer joins produce a results table containing columns from every row in one table and a subset of rows from another table. Outer join syntax is very similar to that of inner joins.

```
SELECT col [, col …] | *
   FROM left_table {LEFT | RIGHT | FULL} OUTER JOIN
       right_table ON joincondition
   [WHERE <searchcondition>]
```

The *joincondition* is of the form `left_table.col = right_table.col` where the equality operator can be replaced by any of the comparison operators.

With outer joins, you must specify the type of join to perform. There are three types:

- A *left outer join* retrieves all rows from the left table in a join, and retrieves any rows from the right table that match the search condition specified in the ON clause.

- A *right outer join* retrieves all rows from the right table in a join, and retrieves any rows from the left table that match the search condition specified in the ON clause.

- A *full outer join* retrieves all rows from both the left and right tables in a join regardless of the search condition specified in the ON clause.

Outer joins are useful for comparing a subset of data in the context of all data from which it is retrieved. For example, when listing the employees that are assigned to projects, it might be interesting to see the employees that are not assigned to projects, too.

▶ **Practice with joins**

1. The following outer join retrieves employee names from the *Employee* table and project IDs from the *Employee_project* table. It retrieves all the employee names, because this is a *left* outer join, and *Employee* is the left table.

```
SELECT e.full_name, p.proj_id
    FROM Employee e LEFT OUTER JOIN Employee_project p
    ON e.emp_no = p.emp_no
    ORDER BY p.proj_id
```

This should produce a list of 48 names. Notice that some employees are not assigned to a project; the *proj_id* column displays <null> for them.

2. Now reverse the order of the tables and execute the query again.

```
SELECT e.full_name, p.proj_id
    FROM Employee_project p LEFT OUTER JOIN Employee e
    ON e.emp_no = p.emp_no
    ORDER BY p.proj_id
```

This produces a list of 28 names. Notice that you get different results because this time the left table is *Employee_project*. The left outer join is only required to produce all the rows of the *Employee_project* table, not all of the *Employee* table.

3. As a last experiment with joins, repeat the query in Exercise 2 (the one you just did), but this time do a *right* outer join. Before you execute the query, think about it for a moment. What do you think this query will return?

```
SELECT e.full_name, p.proj_id
    FROM Employee_project p RIGHT OUTER JOIN Employee e
    ON e.emp_no = p.emp_no
    ORDER BY p.proj_id
```

You should get the same result set as in Exercise 1. Did you realize that performing a right outer join on tables B JOIN A is the same as a left outer join on tables A JOIN B?

Formatting data

This section describes three ways to change data formats:

- Using CAST to convert datatypes
- Using the string operator to concatenate strings
- You can convert characters to uppercase

Using CAST to convert datatypes

Normally, only similar datatypes can be compared in search conditions, but you can work around this by using CAST. Use the CAST clause in search conditions to translate one datatype into another. The syntax for the CAST clause is:

```
CAST (<value> | NULL AS datatype)
```

For example, the following WHERE clause uses CAST to translate a CHAR datatype, INTERVIEW_DATE, to a DATE datatype. This conversion lets you compare INTERVIEW_DATE to another DATE column, *hire_date*:

```
. . . WHERE hire_date = CAST(interview_date AS DATE)
```

You can use CAST to compare columns in the same table or across tables. CAST allows the conversions listed in the following table:

From datatype	To datatype
NUMERIC	CHARACTER, DATE
CHARACTER	NUMERIC, DATE
DATE	CHARACTER, NUMERIC

TABLE 8   Compatible datatypes for CAST

Using the string operator in search conditions

The string operator, also referred to as a *concatenation operator*, ||, joins two or more character strings into a single string. The strings to be joined can be the result set of a query or can be quoted strings that you supply. The operator is the pipe character, typed twice.

▶ **Use the string operator to join strings**

1. Execute the following SELECT statement to concatenate the result of the query with the additional text " is the manager." Remember to have a space as the first character of the string. The query returns the manager names for all department that are not field offices.

```
SELECT D.dept_no, D.department, E.last_name || ' is the manager'
    FROM Department D, Employee E
    WHERE D.mngr_no = E.emp_no AND D.department NOT CONTAINING 'Field'
    ORDER BY D.dept_no
```

You should see the following result:

```
DEPT_NO DEPARTMENT
======= ======================= ===================================
000     Corporate Headquarters  Bender is the manager
100     Sales and Marketing     MacDonald is the manager
110     Pacific Rim Headquarters Baldwin is the manager
120     European Headquarters   Reeves is the manager
600     Engineering             Nelson is the manager
622     Quality Assurance       Forest is the manager
623     Customer Support        Young is the manager
670     Consumer Electronics Div. Cook is the manager
671     Research and Development Papadopoulos is the manager
672     Customer Services       Williams is the manager
900     Finance                 Steadman is the manager
```

2. You can concatenate as many strings as you like. The following query is a slight variation on the previous one: it concatenates the first name to the other output strings:

```
SELECT D.dept_no, D.department, E.first_name || ' ' || E.last_name ||
       ' is the manager'
    FROM Department D, Employee E
    WHERE D.mngr_no = E.emp_no AND D.department NOT CONTAINING 'Field'
    ORDER BY D.dept_no
```

Notice that in order to get a space between the first and last names, you have to concatenate a string that consists solely of a space. The result set should look like this:

```
DEPT_NO DEPARTMENT
======= ======================= =======================================
000     Corporate Headquarters  Oliver H. Bender is the manager
100     Sales and Marketing     Mary S. MacDonald is the manager
110     Pacific Rim Headquarters Janet Baldwin is the manager
120     European Headquarters   Roger Reeves is the manager
600     Engineering             Robert Nelson is the manager
622     Quality Assurance       Phil Forest is the manager
623     Customer Support        Kevin Cook is the manager
670     Consumer Electronics Div. Kevin Cook is the manager
671     Research and Development Chris Papadopoulos is the manager
672     Customer Services       Randy Williams is the manager
900     Finance                  Walter Steadman is the manager
```

Converting to uppercase

The UPPER function converts character values to uppercase. For example, you could include a CHECK constraint that ensures that all column values are entered in uppercase when defining a table column or domain. The following CREATE DOMAIN statement uses the UPPER function to guarantee that column entries are all upper case:

```
CREATE DOMAIN PROJNO
   AS CHAR(5)
   CHECK (VALUE = UPPER (VALUE));
```

Part V **Advanced topics**

This chapter provides examples of some advanced DDL features, including:

- Granting and revoking access privileges
- Creating and using triggers
- Creating and using stored procedures

Access privileges

Initially, only a table's creator, its *owner*, and the SYSDBA user have access to a table. On UNIX servers that have a superuser, or a user with root privileges, those users also have access to all database objects.

You can grant other users the right to look at or change your tables by assigning *access privileges* using the GRANT statement. Table 9 lists the available access privileges:

Privilege	Access
ALL	SELECT, DELETE, INSERT, UPDATE, and REFERENCES; note that ALL does not include the EXECUTE privilege
SELECT	Read data
DELETE	Delete data
INSERT	Write new data
UPDATE	Modify existing data
EXECUTE	Execute or call a stored procedure
REFERENCES	Reference a primary key with a foreign key
ROLE	All privileges assigned to the role

TABLE 9   SQL access privileges

The GRANT statement assigns access privileges for a table or view to specified users, roles, or procedures. The REVOKE statement removes previously granted access privileges.

Assigning privileges with GRANT

The GRANT statement can grant one or more privileges to one or more users. The privileges can be to one or more complete tables or can be restricted to certain columns of the tables. Only UPDATE and REFERENCES privileges can be assigned at the column level.

Granting access to whole tables

The following statement grants one privilege on the *Department* table to one user:

```
GRANT SELECT ON Department TO EMIL
```

The following statement assigns two privileges (INSERT and UPDATE) on the *Department* table to three users:

```
GRANT INSERT, UPDATE ON Department TO EMIL, RAVI, HELGA
```

To grant privileges to everyone, use the PUBLIC keyword. The following statement grants all privileges except EXECUTE on the *Department* table to anyone who connects to the database:

```
GRANT ALL ON Department to PUBLIC
```

Granting access to columns

In the previous examples, users were granted access to entire tables. Often, however, you may want to grant access only to certain columns of a table. The following statement assigns UPDATE privilege to all users for the *contact* and *phone* columns in the *Customers* table:

```
GRANT UPDATE (contact, phone) ON Customers TO PUBLIC
```

This is a very brief introduction to an important topic: security. For more information about granting access, see the *Data Definition Guide*.

Revoking privileges

The REVOKE statement removes access privileges that were granted with GRANT. The following statement removes the insert and update privileges on the *Department* table that were granted to Emil, Ravi, and Helga in an earlier example.

```
REVOKE INSERT, UPDATE ON Department FROM EMIL, RAVI, HELGA
```

Using roles to control security

InterBase 5 is now compliant with entry- and mid-level SQL92 standards regarding roles. A *role* is a named group of privileges.

In practice, a company might want to grant a particular collection of privileges to its sales people and a different collection of privileges to its accounting staff. The privileges list in each case might be quite complex. Without roles, you would have to use a lengthy and detailed GRANT statement each time a new sales or accounting person joined the company. The role feature avoids that.

Implementing roles is a four-step process:

1. Define the role.

   ```
   CREATE ROLE role_name
   ```

2. Grant privileges to the role

   ```
   GRANT {one or more of INSERT, UPDATE, DELETE, SELECT, REFERENCES, EXECUTE}
       ON table_name to role_name
   ```

   ```
   GRANT UPDATE (col_name1, col_name 2) ON table_name TO role_name
   ```

   When the access is restricted to certain columns, as in the second line, only UPDATE and REFERENCES can be granted. EXECUTE must always be granted in a separate statement.

3. Grant the role to users

```
GRANT role_name TO user_name1, user_name2
```

The users have now have all the privileges that were granted to the `role_name` role. But there's an additional step they must take before those privileges are available to them. They must specify the role when the connect to a database.

4. Specify the role when connecting to a database.
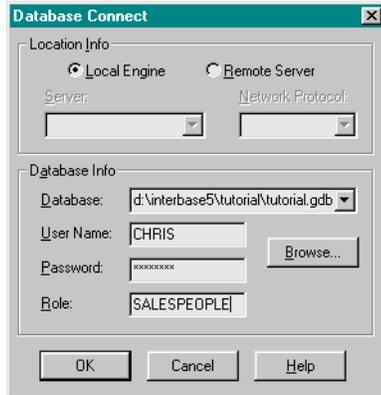
▶ **Commit your work**

If you have not committed your work in InterBase Windows ISQL lately, do so now (**File | Commit Work**). That way, if anything goes astray, you can roll back your work to this point (**File | Rollback Work**).

▶ **Use a role to control access**

This exercise takes you through all four steps of implementing a role. You begin by creating another user and trying to access one of your tables when you are connected as this new user, in order to experience InterBase's security at work. Then you create the Salesperson role, assign some privileges to it, assign the role to your new user, and finally, repeat the access that failed earlier to experience that your new user now has the necessary permissions.

1. Run Server Manager, log in as SYSDBA, and create a new user called CHRIS with a password of chris4ib. Refer to "Creating a new user" on page 6 if you've forgotten how to create a user.

2. Return to InterBase Windows ISQL. Click the Connect 🔲 button or choose **File | Connect to Database** and connect to the TUTORIAL database as CHRIS. Leave the Role field blank.

3. Execute the following SELECT statement:

```
SELECT * FROM SALES
```

InterBase issues an error statement, because all those tables you've created in this tutorial belong to user TUTOR. User CHRIS doesn't have permission to do anything at all with them.

You can click the Details button to get more information about the problem. In this case, it says that the current user has no read/select permissions on the table.

4. Now connect to the TUTORIAL database again, this time as TUTOR. Leave the Role field empty. Create a role called SALESPEOPLE.

```
CREATE ROLE SALESPEOPLE
```

5. Execute the following GRANT statements to assign privileges to the SALESPERSON role. Remember that you must execute each GRANT statement before entering the next one.
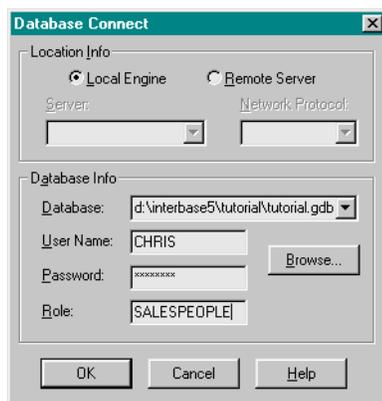
```
GRANT SELECT, UPDATE, INSERT ON Sales to SALESPEOPLE
GRANT update (contact_first, contact_last, phone_no) ON Customer to SALESPEOPLE
```

6. Grant the SALESPEOPLE role to user CHRIS.

```
GRANT SALESPEOPLE TO CHRIS
```

7. Connect to the TUTORIAL database as user CHRIS. Choose Yes when InterBase asks you if you want to commit your work. In the Role field, enter SALESPEOPLE.

8. Now enter the same query that failed in Step 3. This time, InterBase retrieves all the rows in the *Sales* table, because CHRIS now has the required permissions, thanks to the role.

9. Now reconnect to the database as user TUTOR.

Triggers and stored procedures

Stored procedures and triggers are part of a database's metadata and are written in *stored procedure and trigger language*, an InterBase extension to SQL. Procedure and trigger language includes SQL data manipulation statements and some powerful extensions, including IF … THEN … ELSE, WHILE … DO, FOR SELECT … DO, exceptions, and error handling.

▪ A *stored procedure* is a program that can be called by applications or from InterBase Windows ISQL.

Stored procedures can be invoked directly from applications, or can be substituted for a table or view in a SELECT statement. They can receive input parameters from and return values to the calling application.

▪ A *trigger* is a self-contained routine that is associated with a table. A trigger definition specifies an action to perform when a specified event, such as an update, insert, or delete occurs on the table. A trigger is never called directly by an application or user. Instead, when an application or user attempts to perform the action stated in the trigger definition, the trigger automatically executes, or *fires*.

See the *Data Definition Guide* for a full explanation of stored procedures and triggers.

Triggers

Triggers have a great variety of uses, but in general, they permit you to automate tasks that would otherwise have to be done manually. You can use them to define actions that should occur automatically when data is inserted, updated, or deleted in a particular table. Triggers are a versatile tool, with a wide range of uses.

The triggers defined in the TUTORIAL database perform the following actions:

▪ The *set_emp_no* trigger generates and inserts unique employee numbers when a row is inserted into the *Employee* table and the *set_cust_no* trigger does the same for customer numbers in the *Customer* table.

▪ The *save_salary_change* trigger maintains a record of employees' salary changes.

▪ The *new_order* trigger posts an event when a new row is inserted into the *Sales* table.

Using SET TERM

In command-line **isql** and SQL scripts, the trigger statement must be preceded by a SET TERM statement that defines what characters will terminate the trigger statement, since the SQL statements in the body of a trigger must each end with a semicolon (;). The double exclamation mark (!!) is a common choice for this terminator. *The* SET TERM *statement is not necessary or permitted in the InterBase Windows ISQL interface.*

In environments where the terminator was changed using the SET TERM statement, the trigger code should be followed by another SET TERM statement to change the terminator back to a semicolon.

The following example changes the terminator to !!, but ends with the current terminator (semicolon), because that is the one in effect for this statement:

```
SET TERM !!;
```

The structure of triggers

▪ The CREATE TRIGGER keywords are followed by the trigger *name* and the *table* for which the trigger is defined.

▪ The next line determines when the trigger fires. Choices are before or after an insert, deletion, or update. If more than one trigger is defined for a particular point—such as AFTER INSERT—you can add a *position number* that specifies the sequence in which the trigger should fire.

▪ If the trigger uses local variables, they are declared next.

▪ The variables, if any, are followed by SQL statements that determine the behavior of the trigger. These statements are bracketed between the keywords BEGIN and END. Each of these SQL statements ends with a semicolon.

▪ If you are working in **isql** or writing a script, the END keyword is followed by the terminator that was defined by the SET TERM statement.

The syntax for a trigger looks like this.

```
CREATE TRIGGER trigger_name FOR Table_name
{BEFORE | AFTER} {INSERT | DELETE | UPDATE} [POSITION NUMBER]
AS
[DECLARE VARIABLE variable_name datatype;]
BEGIN
    statements in InterBase procedure and trigger language
END
```

See *Triggers.sql* for an example. The *Language Reference*, *Data Definition Guide*, and *Programmer's Guide* all contain more discussion of triggers.

Generators: Generating unique column values

There are many cases in which table columns require unique, sequential values. The *emp_no* column of the *Employee* table is a good example. Without a trigger, you would have to know what the last employee number is each time you add a row for a new employee, so that you could increment it by one to create the new employee number. This is cumbersome and error prone.

Triggers provide a way to automate this process, by using a handy database object called a *generator*. A generator is a named variable that is called and incremented through the **gen_id( )** function. The value of the generator is initialized with SET GENERATOR. After that, it generates the next incremental value each time **gen_id( )** is called. The **gen_id()** function takes a generator name and an increment as inputs.

Context variables

Context variables are unique to triggers. Triggers are often used to change a value, and in the process of doing so, they must temporarily store both the old and new values. The context variables, *Old* and *New*, are the mechanisms by which they do this. As you perform the exercises in this section, look for them in contexts such as `New.emp_no = gen_id(emp_no_gen, 1).` For more information about context variables, see the *Data Definition Guide*

▶ **Create a generator**

1. Begin by checking the employee numbers in the *Employee* table, to confirm that the highest employee number currently in use is 145:

```
SELECT emp_no from Employee
    ORDER BY emp_no
```

Note The statement above returns all the employee numbers so that you can confirm that 145 is the highest. The following statement produces the same information more efficiently:

```
SELECT max(emp_no) from Employee
```

2. Triggers often use generators, and the trigger you create in the next exercise is an example of one. Execute the following statement to create a generator called *emp_no_gen*.

```
CREATE GENERATOR emp_no_gen
```

3. Now initialize the generator to 145, the highest value currently in use.

```
SET GENERATOR emp_no_gen TO 145
```

▶ **Create a trigger that generates a value**

1. The next statements define a trigger named *set_emp_no* that makes use of the *emp_no_gen* generator to generate unique sequential employee numbers and insert them into the *Employee* table.

```
CREATE TRIGGER set_emp_no FOR Employee
BEFORE INSERT AS
    BEGIN
    New.emp_no = gen_id(emp_no_gen, 1);
    END
```

This statement says that the *set_emp_no* trigger will fire before an insert operation, and that it will create a new value for *emp_no* by calling the **gen_id()** function on the *emp_no_gen* generator with an increment of 1.

2. To test the generator, execute the following INSERT statement:

```
INSERT INTO Employee (first_name, last_name, dept_no, job_code, job_grade,
    job_country, hire_date, salary, phone_ext)
    VALUES ('Reed', 'Richards', '671', 'Eng', 5, 'USA', '07/27/95',
        '34000', '444')
```

Notice that you did not include a value for the *emp_no* column in the INSERT statement. Look at the new record by entering

```
SELECT * from Employee WHERE last_name = 'Richards'
```

The employee number is 146. Remember that the highest employee number before you created the generator and inserted a new row was 145. The trigger has automatically assigned the new employee the next employee number.

3. If your INSERT ran without errors and your SELECT returns the correct result set, commit your work.

Finishing the trigger exercises

The remainder of this section on triggers takes you through the process of creating another generator and three more triggers. The text instructs you to enter them by hand in order to get more experience with them.

TIP   If you want to save time, you can use the **File | Run an ISQL Script** command to read in the *Triggers.sql* script in place of entering the remaining trigger and generator statements yourself. *Triggers.sql* defines another generator and a trigger named *set_cust_no* that assigns unique customer numbers. It defines two other triggers: *save_salary_change* and *post_new_order*.

Whether you choose to enter the remaining trigger statements yourself or to run the script, do take time to open *Triggers.sql* in a text editor and see that you understand the code in it. Notice that triggers in a script require the use of the SET TERM statement. (See "The structure of triggers" on page 68 for more about SET TERM.)

If you choose to work through this section manually instead of running the script, commit your work after creating and testing each trigger or stored procedure.

▶ **More practice with generators and triggers**

1. The next trigger that you will create uses the *cust_no_gen* generator. Execute each statement in turn to create and initialize this generator:

```
CREATE GENERATOR cust_no_gen

SET GENERATOR cust_no_gen to 1015
```

Remember, these are two separate statements, and you must execute each one before entering the next.

2. Now execute the following CREATE TRIGGER statement to create the *set_cust_no* trigger.

```
CREATE TRIGGER set_cust_no FOR Customer
    BEFORE INSERT AS
    BEGIN
        new.cust_no = gen_id(cust_no_gen, 1);
    END
```

3. To test this trigger, first select max(cust_no) from *Customer* to confirm that the highest customer number is 1015. Then insert the following row:

```
INSERT INTO Customer (customer, contact_first, contact_last,
    phone_no, address_line1, address_line2, city, state_province,
        country, postal_code, on_hold)
    VALUES ('Big Rig', 'Henry', 'Erlig', '(701) 555-1212', '100 Big Rig Way',
        NULL, 'Atlanta', 'GA', 'USA', '70008', NULL)
```

Now perform the following SELECT to confirm that the new customer number is, as you expect, 1016:

```
SELECT cust_no FROM Customer WHERE customer = 'Big Rig'
```

▶ **Create a trigger to maintain change records**

Enter the following CREATE TRIGGER statement to create the *save_salary_change* trigger, which maintains a record of changes to employees' salaries in the *Salary_history* table:

```
CREATE TRIGGER save_salary_change FOR Employee
    AFTER UPDATE AS
    BEGIN
        IF (Old.salary <> New.salary) THEN
        INSERT INTO Salary_history
        (emp_no, change_date, UPDATER_ID, old_salary, percent_change)
        VALUES (
            OLD.emp_no,
            'NOW',
            USER,
            OLD.salary, (NEW.salary - OLD.salary) * 100 / OLD.salary
        );
    END
```

This trigger fires AFTER UPDATE of the *Employee* table. It compares the value of the *salary* column before the update to the value after the update and if they are different, it enters a record in *Salary_history* that consists of the employee number, date, previous salary, and percentage change in the salary.

Notice that when the values to be entered in the *Salary_history* table are to be taken from the *Employee* table, they are always preceded by the *Old* or *New* context variable. That is because InterBase creates two versions of a record during the update process, and you must specify which version the value is to come from.

In addition, note that this example makes use of two other InterBase features: it inserts the current date into a column of DATE datatype by supplying the string 'NOW' in single quotes, and it inserts the name of the user who is currently connected to the database by supplying the keyword USER.

Update an employee record and change the salary to see how this trigger works.

▶ **Create a trigger that posts an event**

Execute the following CREATE TRIGGER statement to create a trigger, *post_new_order*, that posts an event named "new_order" whenever a record is inserted into the *Sales* table.

```
CREATE TRIGGER post_new_order FOR Sales
AFTER INSERT AS
BEGIN
    POST_EVENT 'new_order';
END
```

An *event* is a message passed by a trigger or stored procedure to the InterBase event manager to notify interested applications of the occurrence of a particular condition. Applications that have registered interest in an event can pause execution and wait for the specified event to occur. For more information on events, see the *Programmer's Guide*.

The *post_new_order* trigger fires after a new record is inserted into the *Sales* table—in other words when a new sale is made. When this event occurs, interested applications can take action, such as printing an invoice or notifying the shipping department.

Stored procedures

Stored procedures are programs stored with a database's metadata that run on the server. Applications can call stored procedures to perform tasks, and you can also use stored procedures in InterBase Windows ISQL. See the *Programmer's Guide* for more information on calling stored procedures from applications.

There are two types of stored procedures:

■ *Select procedures* that an application can use in place of a table or view in a SELECT statement. A select procedure must be defined to return one or more values (output parameters), or an error results. Since select procedures can return more than one row, they appear as a table or view to a calling program.

■ *Executable procedures* that an application can call directly with the EXECUTE PROCEDURE statement. Executable procedures can perform a variety of tasks; they might or might not return values to the calling program.

Both kinds of procedures are defined with CREATE PROCEDURE and have essentially the same syntax. The difference is in how the procedure is written and how it is intended to be used.

Stored procedure syntax

A CREATE PROCEDURE statement is composed of a *header* and a *body*. The header contains:

· The *name* of the stored procedure, which must be unique among procedure, view, and table names in the database.

· An optional list of *input parameters* and their datatypes that a procedure receives from the calling program.

· If the procedure returns values to the calling program, the next item is the RETURNS keyword, followed by a list of *output parameters* and their datatypes.

The procedure body contains:

· An optional list of *local variables* and their datatypes.

· A *block* of statements in InterBase procedure and trigger language, bracketed by BEGIN and END. A block can itself include other blocks, so that there might be many levels of nesting.

The simplified syntax of a procedure looks like this:

```
CREATE PROCEDURE procedure_name
   [(input_var1 datatype[, input_var2 datatype …])]
   [RETURNS (output_var1 datatype[, output_var2 datatype …])]
   AS
      BEGIN
          statements in InterBase procedure and trigger language
      END
```

Like trigger definitions, procedure definitions in SQL scripts, embedded SQL, and command-line **isql** must be preceded by a SET TERM statement that sets the terminator to something other than a semicolon. When all procedure statements have been entered, the SET TERM statement must be used again to set the terminator back to a semicolon. See *Procs.sql* for an example. The *Language Reference*, *Data Definition Guide*, and *Programmer's Guide* all contain more information on stored procedures.

▶ **Create a simple SELECT procedure**

1. Execute the following code to create the *get_emp_proj* procedure. There is a useful convention of starting variable names with "v_" to help make the code readable, but it is not required. You can name variables anything you wish. The following code is a single SQL statement. Enter the whole thing and then execute it:

```
CREATE PROCEDURE get_emp_proj (v_empno SMALLINT)
RETURNS (project_id CHAR(5))
AS
BEGIN
    FOR SELECT proj_id
        FROM Employee_project
        WHERE emp_no = :v_empno
        INTO :project_id
    DO
        SUSPEND;
END
```

This is a select procedure that takes an employee number as its input parameter (*v_empno*, specified in parentheses after the procedure name) and returns all the projects to which the employee is assigned (*project_id*, specified after RETURNS). The variables are named in the header as *varname* and then referenced in the body as *:varname*.

It uses a FOR SELECT … DO statement to retrieve multiple rows from the *Employee_project* table. This statement retrieves values just as a normal SELECT statement does, but retrieves them one at a time into the variable listed after INTO, and then performs the statements following DO. In this case, the only statement is SUSPEND, which suspends execution of the procedure and sends values back to the calling application (in this case, InterBase Windows ISQL).

2. See how the procedure works by entering the following query:

```
SELECT * FROM get_emp_proj(71)
```

This query looks at first as though there were a table named *get_emp_proj*, but you can tell that it's a procedure rather than a table because of the input parameter in parentheses following the procedure name. The results are:

```
project_id
==========
VBASE
MAPDB
```

These are the projects to which employee number 71 is assigned. Try it with some other employee numbers.

▶ **Create a more complex select procedure**

1. The next exercise starts with the code for the previous procedure and adds an output column that counts the line numbers. Execute the following code. (You can display the previous query if you wish and add the new code.)

```
CREATE PROCEDURE get_emp_proj2 (v_empno SMALLINT)
    RETURNS (line_no integer, project_id CHAR(5))
    AS
    BEGIN
    line_no = 0;
        FOR SELECT proj_id
            FROM Employee_project
            WHERE emp_no = :v_empno
            INTO :project_id
        DO
            BEGIN
                line_no = line_no+1;
                SUSPEND;
            END
    END
```

2. To test this new procedure, execute the following query:

```
SELECT * FROM get_emp_proj2(71)
```

You should see the following output:

```
    LINE_NO PROJECT_ID
=========== ==========
          1 VBASE
          2 MAPDB
```

3. If your procedure returns the correct result set, commit your work.

▶ **Create a simple executable procedure**

1. The executable procedure that you create in the next step of this exercise, *add_emp_proj*, makes use of an *exception*, a named error message, that you define with CREATE EXCEPTION. Execute the following SQL statement to create the UNKNOWN_EMP_ID exception:

```
CREATE EXCEPTION unknown_emp_id
    'Invalid employee number or project ID.'
```

Once defined, this exception can be *raised* in a trigger or stored procedure with the EXCEPTION clause. The associated error message is then returned to the calling application.

2. Execute the following statement to create the *add_emp_proj* stored procedure:

```
CREATE PROCEDURE add_emp_proj (v_empno SMALLINT, v_projid CHAR(5))
AS
BEGIN
    INSERT INTO Employee_project (emp_no, proj_id)
    VALUES (:v_empno, :v_projid);
    WHEN SQLCODE -530 DO
        EXCEPTION UNKNOWN_EMP_ID;
END
```

This procedure takes an employee number and project ID as input parameters and adds the employee to the specified project using an INSERT statement. The error-handling WHEN statement checks for SQLCODE -530, violation of foreign key constraint, and then raises the previously-defined exception when this occurs.

3. Practice using this procedure by executing the following SQL statement:

```
EXECUTE PROCEDURE add_emp_proj(20, 'DGPII')
```

To confirm that this worked, execute the following SELECT statement:

```
SELECT * FROM Employee_project where emp_no = 20
```

You should see that employee 20 is now assigned to both the DGPII project and the GUIDE project.

4. Now try adding a non-existent employee to a project, for example:

```
EXECUTE PROCEDURE add_emp_proj(999, 'DGPII')
```

The statement fails and the exception message displays on the screen.
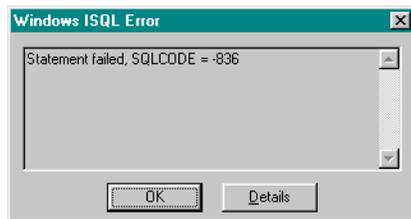
5. Click the Details button to view the text that you specified when you created the exception ("Invalid employee number or project ID.").

Recursive procedures

Stored procedures support *recursion*, that is, they can call themselves. This is a powerful programming technique that is useful in performing repetitive tasks across hierarchical structures such as corporate organizations or mechanical parts.

▶ **Create a recursive procedure**

In this exercise, you create a stored procedure called *dept_budget* that takes a department number as its input parameter and returns the budget of the department and all departments that are under it in the corporate hierarchy. It uses *local variables* declared with DECLARE VARIABLE statements. These variables are used only within the context of the procedure.

First, the procedure retrieves the budget of the department given as the input parameter from the *Department* table and stores it in the *total_budget* variable. Then it retrieves the number of departments reporting to that department using the COUNT aggregate function. If there are no reporting departments, it returns the value of *total_budget* with SUSPEND.

Using a FOR SELECT … DO loop, the procedure then retrieves the department number of each reporting department into the local variable *rdno*, and then recursively calls itself with

```
EXECUTE PROCEDURE dept_budget :rdno RETURNING_VALUES :sumb
```

This statement executes *dept_budget* with input parameter *rdno*, and puts the output value in *sumb*. Notice that when using EXECUTE PROCEDURE within a procedure, the input parameters are not put in parenthesis, and the variable into which to put the resultant output value is specified after the RETURNING_VALUES keyword. The value of *sumb* is then added to *total_budget*, to keep a running total of the budget. The result is that the procedure returns the total of the budgets of all the reporting departments given as the input parameter plus the budget of the department itself.

1. Execute the following SQL statement:

```
CREATE PROCEDURE dept_budget (v_dno CHAR(3))
RETURNS (total_budget NUMERIC(15, 2))
AS
    DECLARE VARIABLE sumb DECIMAL(12, 2);
    DECLARE VARIABLE rdno CHAR(3);
    DECLARE VARIABLE cnt INTEGER;
BEGIN
    total_budget = 0;
    SELECT budget FROM Department WHERE dept_no = :v_dno INTO :total_budget;
    SELECT COUNT(budget)
        FROM Department
        WHERE head_dept = :v_dno
        INTO :cnt;
    IF (cnt = 0) THEN
        SUSPEND;
    FOR SELECT dept_no
        FROM Department
        WHERE head_dept = :v_dno
        INTO :rdno
    DO
    BEGIN
        EXECUTE PROCEDURE Dept_budget :rdno RETURNING_VALUES :sumb;
            total_budget = total_budget + sumb;
    END
END
```

2. To find the total budget for department 620, including all its subdepartments, execute the following SQL statement:

```
EXECUTE PROCEDURE dept_budget(620)
```

The result is:

```
        TOTAL_BUDGET
=====================
        2550000.00
```

A little about datatype conversion

Notice that the *dept_budget* procedure is defined to take a CHAR(3) as its input parameter, but that you can get away with giving it an integer (without quotes). This is because of InterBase's automatic type conversion, which converts datatypes, where possible, to the required datatype. It automatically converts the integer 620 to the character string "620". The automatic type conversion won't work for department number 000, however, because it would convert to the string "0", which is not a department number.

More procedures

There are a number of other procedures, some quite complex, defined in *Procs.sql* for the TUTORIAL database. Now that you have a basic understanding of procedures, it will be worth your while to read them over so that you understand them. Then try using them. Notice that comments are often included within the text of a statement to make it easier for people to understand what the code is doing.