PART ONE:
Getting started with Windows ISQL

This section of the tutorial introduces you to some fundamental database concepts and shows you how to apply them using the Windows ISQL (WISQL) tool.

As you apply the lessons you'll build the foundation for a personnel database for employee records, similar to the sample database provided with InterBase.

Basic concepts covered in this section include:

- How to create a database.
- How to create tables.
- How to add data to tables and modify the data.
- How to retrieve data from tables.

next lesson

Starting the Windows ISQL session

The first step is to start Windows ISQL. Normally, you do this by clicking the Windows ISQL icon in the InterBase Windows Client program group. To get you started, however, you can use this shortcut:

Click here to open WISQL.

The main WISQL interface is made up of a menu bar, several buttons and two windows-one for entering SQL statements and one for viewing the output generated by those statements.

Before entering any SQL statements, however, you must first connect to a database. We'll get to that after a quick overview of SQL.

previous lesson
next lesson

An overview of SQL

SQL statements are divided into two major categories:

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

DDL statements are used to define, change, and delete a database. Collectively, the objects defined with DDL statements are known as metadata.

Basic DDL statements that create metadata begin with the keyword CREATE. Statements that modify metadata begin with the keyword ALTER. Statements that delete metadata begin with the keyword DROP. So, for example, you would use CREATE TABLE to define a table, ALTER TABLE to modify the table, and DROP TABLE to delete it.

DML statements are used to manipulate data within the data structures defined with DDL statements. The three basic DDL statements are INSERT, UPDATE, and DELETE.

INSERT adds data to a table, UPDATE modifies data, and DELETE removes data.

DML also includes the SELECT statement, which you'll use often to retrieve or query information from a database.

previous lesson
next lesson

Creating a database

InterBase is a relational database. A relational database is a collection of tables, which are two-dimensional structures composed of rows (also called records) and columns (also called fields). InterBase databases are stored in files, and customarily bear the extension GDB.

To create a database, you must have a valid user name and password in the security database, ISC4.GDB. For instructions on how to access the security database using InterBase's Server Manager, see the description of Server Manager's File | Login menu item.

Initially, the information you need to supply to create a database is the pathname of the database. Create a database now by choosing File | Create Database.... The Create Database dialog box opens. Press Tab to move the cursor from one field to the next. Make sure the User Name field shows your user name. Enter your InterBase password in the Password text field.

In the Database field, type the name of the database to create, including the directory path and file specification. For example:

```
C:\PROGRAM FILES\BORLAND\INTRBASE\TUTORIAL\EXAMPLES\MYDB.GDB
```

Note: Be sure to create the database in a directory where you have the necessary file permissions. If you create the database in your own directory area, then you can give it any name you want. If you create it in a common directory such as IBLOCAL\EXAMPLES, you should give the database a unique name to avoid conflicts with other users. For example, if your name is Fred, you could name the database FRED.GDB. You do not have to use the .GDB file name extension, but it is an InterBase convention.

Leave the Database Options area empty, because you are creating a simple database.

Click OK to create the database.

After a short pause, a message appears in the status area at the bottom of the window:

```
Database: C:\PROGRAM FILES\BORLAND\INTRBASE\TUTORIAL\EXAMPLES\MYDB.GDB
```

This message lets you know you are connected to your new database.

previous lesson
next lesson

Conceptual database design

Before getting into the details of building a database, you should step back and determine exactly what you want to do with the database. In this "conceptual design" phase, you should basically define the objects you want to model with the database, their characteristics, and their relationships. Try to map out the details as much as possible before actually doing any SQL data definition. Sometimes it is useful to create diagrams on paper to help visualize the database.

Because this section of the tutorial is intended to introduce basic database concepts, you are going to build a very simple database, containing only two tables. In the real world, databases will rarely be this simple. But imagine that you only need to keep track of employees and the corporate department to which they belong. The best way to do this is to have a table for the employees, and a table for the departments.

The employee table should contain a row for each employee, and a column for each item of information related to the employee. For this simple example, let's assume that we only need to keep track of the employee's name (first and last), employee number, and department. Each department has a unique department number and a department name.

A conceptual diagram of this database might look like this:

The box on the left represents the table for employees, and the box on the right represents the table for departments. The columns of each table are listed inside each box, and an asterisk indicates that the value of the column uniquely identifies a row (this is known as a primary key, and will be explained later). The arrow indicates that each department number entered in the employee table references a department number in the department table (this is known as a foreign key, and will be explained later).

previous lesson
next lesson

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 relationships among the tables.

For example, in this simple personnel database, the table for employees could be called EMPLOYEE, with columns as defined previously. Each row represents an individual employee. Here is an illustration of what such an EMPLOYEE table might look like this:

EMPNO	LAST_NAME	FIRST_NAME	DEPT_NO
10335	Smith	John	180
21347	Carter	Catherine	620
13314	Jones	Sarah	100
5441	Lewis	Stephen	180

The table containing information on departments could be called DEPARTMENT, and look like this:

DEPTNO	DEPARTMENT
180	Marketing
620	Software Products Div.
100	Sales
600	Engineering

The DEPARTMENT table is simple, so it makes a good starting point. To create a table with Windows ISQL, use the CREATE TABLE statement. The full syntax of this statement, as shown in the Language Reference is quite complex, but the basic form is simple: the keywords CREATE TABLE, followed by the name of the table, and then in parentheses a list of the columns in the table, separated by commas. Each column in the list specifies the name of the column, the data type, and attributes of the column such as NOT NULL and UNIQUE.

To create the DEPARTMENT table, type the following in the SQL Statement area:

```
CREATE TABLE DEPARTMENT
(DEPT_NO CHAR(3) NOT NULL UNIQUE, DEPARTMENT VARCHAR(25) NOT NULL);
```

SQL is not case-sensitive, so you can enter statements in uppercase or lowercase. You could enter the statement all on one line, but it is easier to read if spread across several lines. The semicolon at the end of the statement is optional.

Type the above statement and click on Run. If you did not make any typing mistakes, then the statement will be echoed in the WISQL Output area. If you made a mistake, an error message will be displayed.

Note: From now on in this section of the tutorial, it will be assumed that you know how to type SQL statements in the SQL Statement area, and click on the Run button when you are done.

This statement creates a table called "DEPARTMENT" with two columns: DEPT_NO for the department number and DEPARTMENT for the department name. The department number is defined as a three-character string, and department name is defined as a string with up to 25 characters. The keywords NOT NULL signify that each row must contain data in that column. UNIQUE means that the data in the column must be unique. So each department must have a name and department number and each department number must be unique.

To view your new table definition, choose View | Metadata Information.... The View Information dialog box opens.

Display a drop-down list of types of metadata objects by clicking on the arrow to the right of the top field, then select "Table" and type the name of the table, DEPARTMENT in the Object Name field. Click on OK. The table definition is displayed in the WISQL Output area.

At any point during this tutorial, you can view metadata by choosing View | Metadata Information... and selecting the type of metadata. If you do not enter anything in the Object Name field, then WISQL will display the names of all the metadata objects of the selected type. If you enter a name, then WISQL will display all the details about that object.

Next, you will create the EMPLOYEE table. But first, you have to understand some new concepts.

previous lesson
next lesson

Primary keys and foreign keys

A primary key is a column or set of columns that uniquely identifies a row. In practice, every table should have a primary key. In the employee table, EMP_NO should be a primary key for EMPLOYEE because the employee number uniquely identifies an employee and DEPT_NO should be the primary key for DEPARTMENT because it uniquely identifies a department.

A foreign key is a column in one table that is the primary key column for another table. Primary key and foreign key constraints are defined with the PRIMARY KEY and FOREIGN KEY keywords in a CREATE TABLE statement.

Now define a new table with a primary key and a foreign key. Type the following in the SQL Statement area:

```
CREATE TABLE EMPLOYEE
(EMP_NO SMALLINT NOT NULL,
LAST_NAME VARCHAR(25) NOT NULL,
FIRST_NAME VARCHAR(20) NOT NULL,
DEPT_NO CHAR(3) NOT NULL,
PRIMARY KEY (EMP_NO),
FOREIGN KEY (DEPT_NO)
REFERENCES DEPARTMENT (DEPT_NO));
```

This statement creates a table called "EMPLOYEE" with four columns: EMP_NO for each employee's employee number, FIRST_NAME and LAST_NAME, for each employee's first and last names, and DEPT_NO for the employee's department number. NOT NULL after each column name signifies that data is required in the column when a row is added to the database.

After the list of columns, the keywords PRIMARY KEY define the table's primary key to be the EMP_NO column. The keywords FOREIGN KEY indicate that DEPT_NO references a column in another table, and the data in this column must match the data in the other table.

Make sure the table definition is entered in the database by choosing View | Metadata Information..., selecting Tables, and typing EMPLOYEE as the table name. You should see the table definition in the WISQL Output area.

previous lesson
next lesson

Adding data to tables

Creating a table with CREATE TABLE simply defines the data structure. To create a useful database, you must then add data to the table. The easiest way to add data to a table in SQL is with the INSERT statement. The simplest form of the INSERT statement specifies values to insert into all the columns in a single row of a table, as follows:

```
INSERT INTO table_name VALUES (val1, val2, ...);
```

where table_name is the name of the table, and val1, val2, and so on, are the values to insert.
To use this syntax, you must know the default order of the columns. Because you just created the table with the DEPTNO column first and then DEPARTMENT, you know to give the department number first and then the name. If you try to insert values in a different order, you will get an error.
Now type the following:

```
INSERT INTO DEPARTMENT VALUES (180, "Marketing");
INSERT INTO DEPARTMENT VALUES (100, "Sales");
```

Note: Windows ISQL itself does not require a semicolon at the end of each statement. The semicolon (or another terminator character) is required only in statements in WISQL script files. It is a good idea to get in the habit of ending your SQL statements with semicolons, however, so you will not forget to do so when creating script files.

There is a more general form of the INSERT statement that enables you to enter values for specific columns, even if you do not know the default order:

```
INSERT INTO table_name (col1, col2, ...) VALUES (val1, val2, ...);
```

where col1 is the name of the column into which to insert value val1, col2 is the name of the column into which to insert val2, and so on. To insert the next row of the DEPARTMENT table, type:

```
INSERT INTO DEPARTMENT (DEPARTMENT, DEPT_NO)
VALUES ("Software Products Div.", 620);
```

This form of the INSERT statement is useful if you want to insert values into a subset of the columns, or you do not remember the default column order.
Now, using either form of the INSERT statement, insert one more row into the DEPARTMENT table for a department named "Engineering" with department number 600.
Type the following to insert the values in the EMPLOYEE table. Be sure to enter the statements one at a time and click on Run after each.

```
INSERT INTO EMPLOYEE VALUES (10335, "Smith", "John", 180);
INSERT INTO EMPLOYEE VALUES (21347, "Carter", "Catherine", 620);
INSERT INTO EMPLOYEE VALUES (13314,"Jones", "Sarah", 100);
INSERT INTO EMPLOYEE VALUES (5441, "Lewis", "Stephen", 180);
```

Tip: After you run the first of these statements, you can save some typing by choosing the Previous button to recall it to the SQL Statement area, highlighting the data following the VALUES keyword, and typing just the new values instead of the entire statement.

previous lesson
next lesson

Testing referential integrity

InterBase databases include a feature called referential integrity. Referential integrity in its simplest form are constraints placed upon data by primary and foreign key definitions. When you defined the EMPLOYEE table, you made EMP_NO its primary key and DEPT_NO its foreign key, referencing the DEPARTMENT table. What this means is that each row in the EMPLOYEE table must have a unique value for the EMP_NO column and the value of the DEPT_NO column must match a value in the DEPARTMENT table. These referential integrity constraints are translations of real-world rules: each employee must have a unique employee number and each employee must be assigned to an existing department.

Test out the referential integrity rules for yourself to see how InterBase handles them. First, try to add an employee with the same employee number as another employee. Enter the following:

```
INSERT INTO EMPLOYEE VALUES (21347, "Lesh", "Phil", 620);
```

A small error dialog box will appear stating: "Statement failed, SQLCODE = -803". Choose the Detail button to get more information.

You will see the error message: "Violation of PRIMARY or UNIQUE KEY constraint INTEG_8". The constraint name shown may be something other than INTEG_8, because InterBase automatically gives names to integrity constraints if you do not explicitly name them in your CREATE TABLE statement, and the name it gives them depends on other DDL done previously.

Now try to add an employee with a non-existent department number. Enter:

```
INSERT INTO EMPLOYEE VALUES (7742, "West", "August", 999);
```

The error dialog will appear with the message: "Statement failed, SQLCODE = -530". Choose the Detail button, and you will see "Violation of FOREIGN KEY constraint: INTEG_9". The referential integrity rules will not let you enter an employee with a department number that is not in the DEPARTMENT table.

Committing Work

By default, data definition statements are automatically committed by Windows ISQL. DML statements, such as INSERT, UPDATE, and DELETE, are not committed unless you explicitly do so by choosing File | Commit Work. This means that you can undo any DML statements since the last time you committed. Commit your work now by choosing File | Commit Work to make the changes to the database permanent.

previous lesson
next lesson

Viewing data

Now that you have put data into the tables, you need a way to view it. This requires one of the most important statements in SQL: SELECT. Because SELECT is so powerful, it has a very complex syntax, allowing a great deal of freedom in retrieving data from tables. The simplest form of SELECT is easy, though:

```
SELECT * FROM table_name;
```

where table_name is the name of the table from which to retrieve data, and the asterisk (*) means to select all columns from the table. Enter this statement for the DEPARTMENT table:

```
SELECT * FROM DEPARTMENT;
```

The statement will be echoed to the WISQL Output area, and you should also see the following output:

DEPTNO	DEPARTMENT
180	Marketing
100	Sales
620	Software Products Div.
600	Engineering

Enter the corresponding statement for the EMPLOYEE table to see the values you inserted.

```
SELECT * FROM EMPLOYEE;
```

Instead of selecting all columns from a table, you can specify certain columns, using this form of SELECT:

```
SELECT col1, co2, ... FROM table_name;
```

where col1, col2, and so on, are the names of the columns to select from the table. Experiment with this form to view subsets of the columns of the EMPLOYEE table.

The SELECT statement has a wealth of clauses that make it such a powerful statement. One of the most useful is the WHERE clause, that enables you to specify conditions that rows must meet. For example, enter the following:

```
SELECT * FROM EMPLOYEE WHERE DEPT_NO = 180;
```

This query selects rows with DEPT_NO equal to 180, in other words, only employees in department 180.

previous lesson
next lesson

Modifying data

Now that you have entered data into tables, and learned how to view it, how do you change the data? The UPDATE statement enables you to modify existing rows, using the following syntax:

```
UPDATE table_name SET col1 = val1, col2 = val2,
. . . WHERE condition;
```

where table_name is the name of the table being updated, col1, col2, and so on, are the names of the columns being updated, and val1, val2, and so on, are the new values to assign to the columns. The condition determines which rows are updated. Although condition in its full form allows a great deal of flexibility in determining rows, its basic form is:

```
column [ = | > | < | >= | <= ] value
```

In other words, a simple condition compares the value of a column with some fixed value.

So, for example, say Sarah Jones (employee number 13314) gets married and changes her last name to Zabrinske. To change her record in the employee table, enter the following:

```
UPDATE EMPLOYEE SET LAST_NAME = "Zabrinske" WHERE EMP_NO = 13314;
```

Because EMP_NO is the primary key of the EMPLOYEE table, the condition is guaranteed to identify exactly one row to update. Check that the record has been updated by entering a SELECT statement. The update statement can make sweeping changes to the database, so use caution when entering it against a real database.

The other major DML statement in SQL is DELETE. This statement deletes rows from the table, and should be used with caution to avoid losing valuable data.

The basic form of DELETE is:

```
DELETE FROM table_name WHERE condition;
```

where table_name is the name of the table from which rows are being deleted, and condition is the condition that determines which rows are deleted. As in the UPDATE statement, condition can be quite complex, but in its simplest form it compares the value of a column with a fixed value.

Say Catherine Carter (employee number 21347) is leaving the company, and you want to delete her record from the EMPLOYEE table. Then type:

```
DELETE FROM EMPLOYEE WHERE EMP_NO = 21347;
```

Confirm that the record has been deleted by entering

```
SELECT * from EMPLOYEE WHERE LAST_NAME = "Carter";
```

You won't get an error message, but no output (only the command you entered) is displayed in the WISQL Output area.

previous lesson
next lesson

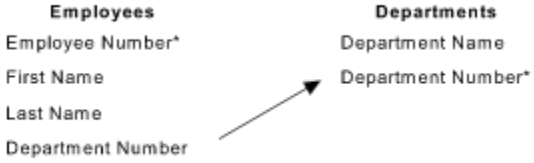
Ending the WISQL session

Whenever you finish your work with WISQL, you should commit it to make it permanent.
To do that, choose File | Commit Work.
If you want to continue the tutorial, do not exit Windows ISQL-continue to the next set of lessons.
If you've had enough for now, you can end your WISQL session by choosing File | Exit to disconnect from the database and exit WISQL. If you want to keep Windows ISQL running, you can choose File | Disconnect from Database to disconnect from the database only.
Now that you have gained some basic experience with SQL, you can move on to the next series of lessons for more detailed tutorial examples.

Sorry. Windows ISQL (WISQL32.EXE) or one of its required library files (DLLs) can't be located. WISQL32.EXE is normally found in the same folder as this Help file, but it may have been relocated since installation. Please try to open WISQL32.EXE yourself by double-clicking the Windows ISQL icon in the InterBase Windows Client program group. If the error message says a DLL is missing, try to locate the missing library file on your hard disk or on the installation disks and restore it to its installed location (DLLs are normally installed into your Windows or Windows/System folder).

PART TWO:
Basic data definition

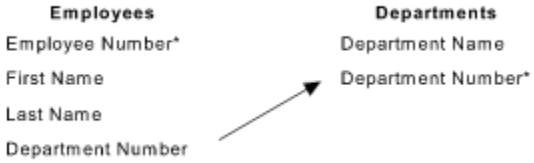
This series of lessons introduce several more basic database concepts and adds new features to the simple database you created in Part One.



More conceptual design

In the previous section of the tutorial, you defined a database consisting of two tables: EMPLOYEE and DEPARTMENT. Now you are going to move from this basic example to a personnel and sales database that might actually be useful in a "real-world" application. Obviously, you will need more than just two tables. You will also have to add more columns and other attributes to the two existing tables.

Start by defining the goals of the database. Let's say that upper management has determined that your company needs a database to keep track of:



Personnel records

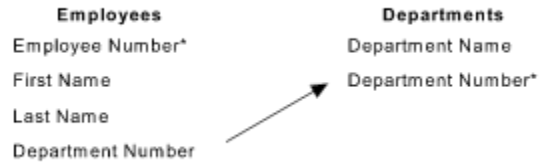
Projects and budgets

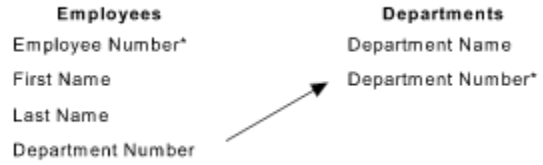
Sales

Customers

Your company does business all over the world, so the database will have to account for many different countries.

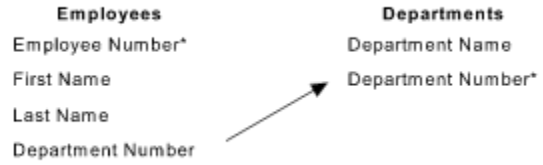
Personnel records include each employee's employee number and name (as before), salary, job code, job grade, and country, and other associated details. The database also needs to maintain information on the manager of each department, the department's budget and location, and how it fits in the departmental hierarchy. Records must be maintained on each job type, including job requirements, maximum and minimum salary, and language requirements. Each employee's salary history must also be maintained.
Project records include the name, project ID, team leader, product type, description of each project, and the project to which each employee is assigned. The department containing the project, the project's budget, and quarterly head count are also important.
Sales records include important information from each purchase order, including PO number, customer, salesperson, date shipped, and so on.
Customer records include a unique customer number, contact names, addresses, and phone number, and other related information.
Designing a database means deciding which tables belong in the database, which columns belong in each table, and the relationship between the tables. A database design in a relational database affords flexibility because the logical structure of the database is independent of the physical storage and structure of the database.
Two concepts, relationship modeling and normalization, are basic to designing a database.



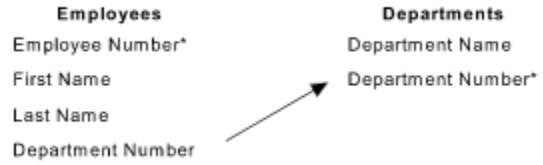


Relationship modeling

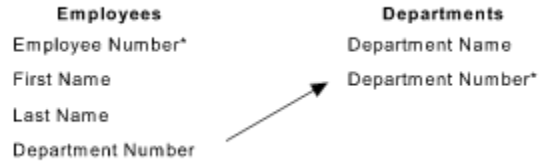
Relationship modeling includes:



Identifying the major groups of information to store in the database.



Analyzing the type of information and its properties.



Identifying relationships among sets of information.

For example, think of the groups of information as tables, with each table describing one thing, such as a company or an employee. The type of information and its properties are columns in the table, describing the employee's salary and the company address. Some questions to ask then are, "Does the information work as a table?" Or "Do the columns need to be moved from one group to another?"

Normalization

Normalization means splitting tables into two or more smaller related tables that can then be joined back together. Initially in a database design, you will probably create tables that contain data that are all related. As your design progresses, however, you will find that you need tables that contain a narrower focus of data.

Normalization also applies to columns within tables. Each column in a row should contain only one value that cannot be broken down into a smaller value. For example, one column should contain an employee first name, another column contains the employee last name, instead of having a single column for first and last name.

After studying the requirements, you determine that you need the following tables:

EMPLOYEE for employees' records, and SALARY_HISTORY for each employee's salary history.

DEPARTMENT for records on each department.

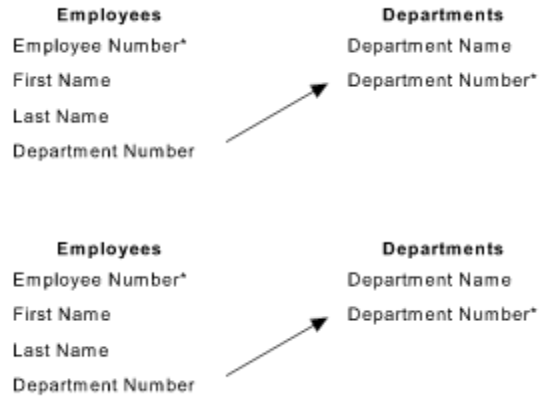
JOB for information about each job type.

PROJECT, EMP_PROJECT, and PROJ_DEPT_BUDGET for project records.

CUSTOMER for records on each customer.

SALES for sales information.

COUNTRY for maintaining the currency for each country.



Defining domains

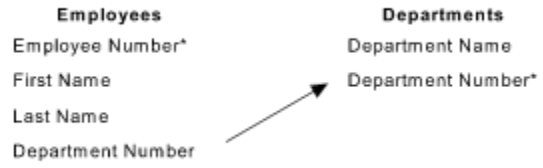
A domain is a customized column definition used in creating tables. A domain allows you to define a column with complex characteristics that you can incorporate in many tables simply by referencing the domain name. This simplifies data definition. Conceptually, a domain is like a user-defined data type.
For example, you could define a domain to use for all employee names in the database. Then every time you need to define a column to contain a name in a table, you can simply refer to the domain. This is a simple example, but you can attach CHECK constraints and other advanced features to a domain definition. Referring to the domain is then much easier than referring to the complex column definition.
Use the SQL statement CREATE DOMAIN to define a domain, including the name of the domain, its data type, and optional characteristics like default value and CHECK constraints. You can use the ALTER DOMAIN statement to change the domain definition, and it changes in every table in which it is used. This simplifies maintenance and updating of the database.
Defining domains is often one of the first steps in data definition, because you can then use the domains in creating tables. The syntax to define domains consists of the keywords CREATE DOMAIN, followed by the name of the domain, then the keyword AS, followed by the data type of the domain, and finally any of the optional characteristics of the domain.
Not every column needs to be defined as a domain, but if it is something that is likely to be used many times in the database, it is a good candidate. For now, you will define domains for employees' first and last names, employee number, and department number.
First, define domains for employees' first and last names and employee numbers. Type the following statements in the SQL Statement area. Be sure to click on the Run button after typing each statement.

```
CREATE DOMAIN LASTNAME AS VARCHAR(20);
CREATE DOMAIN FIRSTNAME AS VARCHAR(15);
CREATE DOMAIN EMPNO AS SMALLINT;
```

If you typed the statements correctly, then each one will be echoed in the ISQL Output area after it is executed.
Next, define a domain for department number. It is defined as a three-character string. In addition to the data type, this domain includes CHECK constraints to ensure that the department number is either "000", alphabetically between "0" and "999", or NULL. Enter the following and then click on Run.

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

You can enter the statement on one line or on several lines to make it easier to read.



Using data definition files

To define the rest of the domains in the database, you can use a data definition file. A data definition file (also referred to as an ISQL script file) contains ISQL statements, and is created with an editor (such as Windows Notepad) and run by Windows ISQL. Data definition files can be very useful, because you can enter multiple SQL statements with all the tools that a text editor provides, including cut, copy, and paste. This makes repetitive tasks much easier.

In practice, most data definition is performed using data definition files, because they enable you to maintain a record of the DDL executed and allow you to work in a text editor instead of command by command.

The data definition files you will need are included in the EXAMPLES\TUTORIAL subdirectory of the InterBase directory. They all have file name extensions of .SQL.

The file, DOMAINS.SQL, contains domain definitions. View this file with Windows Notepad. The first line in the file is a CONNECT statement followed by a dummy database name, user name, and password:

```
CONNECT "server:\dir\mydb.gdb"
USER "USERNAME" PASSWORD "password";
```

Important: Every ISQL script file must begin with a CONNECT statement (or a CREATE DATABASE statement) to connect to a database.

Edit the file and change the database name, user name, and password. Be sure to save the changes before proceeding.

Note: You'll later make the same changes to the CONNECT statement at the beginning of all script files used in this tutorial. To save time, you can cut and paste the information from one file to another.

Now look at the rest of the file, DOMAINS.SQL. You will see that it contains a number of CREATE DOMAIN statements:

```
CREATE DOMAIN ADDRESSLINE AS VARCHAR(30);
CREATE DOMAIN PROJNO
   AS CHAR(5)
   CHECK (VALUE = UPPER (VALUE));
CREATE DOMAIN CUSTNO
   AS INTEGER
   CHECK (VALUE > 1000);
. . .
```

To execute the statements in this file, choose File | Run ISQL Script....

A standard file locator dialog appears. Use the dialog to locate the file DOMAINS.SQL. When you locate it, click Open. A message pops up to ask if you want to save the results to a file. Click No (you want to see the results in the ISQL Output window instead).

As Windows ISQL reads the script file, it echos the statements to the ISQL Output area.

To confirm the domains have been created, choose View | Metadata information..., select Domain from the dropdown list, then click OK. You should see all the domains defined for the database displayed in the SQL Output area.

Starting over

In the previous series of lessons, you created some simple tables and populated them with data. Now its time to delete those tables and the data they contain so you can create and populate a more complex database.

Before you can remove the tables, though, you have to make sure that ISQL will release them. Depending on what you have been doing with ISQL, there may be an active transaction. Choose File | Commit Work (if it is not dimmed) to end any active transactions. If the menu selection is dimmed, then there is no transaction to commit.

Now you can remove these tables from the database. To do this, you will use the DROP statement, which is used to delete metadata. Enter the following:

```
DROP TABLE EMPLOYEE;
DROP TABLE DEPARTMENT;
```

Because DDL statements are automatically committed by default, you do not need to commit these statements to make them permanent. Confirm that the tables are gone by choosing View | Metadata Information... and selecting Tables. Now that you have deleted these two tables and all their data, you can move on and create the EMPLOYEE sample database.

Creating More Tables

Refresh your memory of CREATE TABLE syntax by entering the following statement:

```
CREATE TABLE COUNTRY
(COUNTRY COUNTRYNAME NOT NULL PRIMARY KEY,
CURRENCY VARCHAR(10) NOT NULL);
```

This defines a two-column table to hold the names of countries and their currencies. Notice the declaration of the COUNTRY column uses the COUNTRYNAME domain instead of a standard data type.

Now you will move on to more complex tables using the domains you defined in the previous section. The file, TABLES.SQL, contains statements to create the rest of the tables in the database. Open the file with Notepad to view it.

Note: Don't forget to edit the CONNECT statement at the beginning of TABLES.SQL and put in your database name, user name, and password, and to save the file after changing the connection information.

The first table defined in the file is a more complex version of DEPARTMENT. The definition looks like this:

```
CREATE TABLE DEPARTMENT
   (DEPT_NO DEPTNO NOT NULL,
   DEPARTMENT VARCHAR(25) NOT NULL UNIQUE,
   HEAD_DEPT DEPTNO,
   MNGR_NO EMPNO,
   BUDGET BUDGET,
   LOCATION VARCHAR(15),
   PHONE_NO PHONENUMBER DEFAULT "555-1234",
   PRIMARY KEY (DEPT_NO),
   FOREIGN KEY (HEAD_DEPT) REFERENCES DEPARTMENT (DEPT_NO));
```

The second table defined in the file is named JOB, and defines job descriptions. The next table is the complete EMPLOYEE table. The definition of this table is central to this database:

```
    CREATE TABLE employee
    (
    emp_no          EMPNO NOT NULL PRIMARY KEY,
    first_name      FIRSTNAME NOT NULL,
    last_name       LASTNAME NOT NULL,
    phone_ext       VARCHAR(4),
    hire_date       DATE DEFAULT 'NOW' NOT NULL,
    dept_no         DEPTNO NOT NULL,
    job_code        JOBCODE NOT NULL,
    job_grade       JOBGRADE NOT NULL,
    job_country     COUNTRYNAME NOT NULL,
    salary          SALARY NOT NULL,
    full_name       COMPUTED BY (last_name || ', ' || first_name),
```

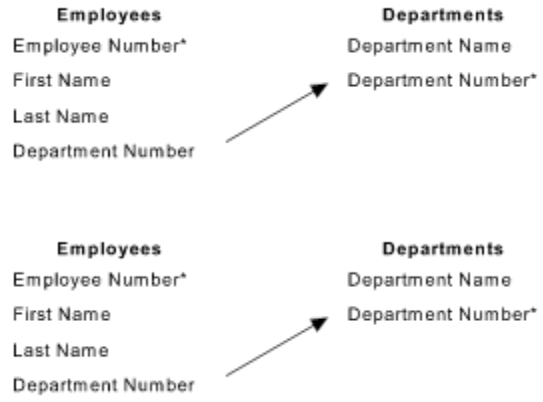
```
    FOREIGN KEY (dept_no)
      REFERENCES department (dept_no),
    FOREIGN KEY (job_code, job_grade, job_country)
      REFERENCES job (job_code, job_grade, job_country),

    CHECK ( salary >= (SELECT min_salary FROM job WHERE
                job.job_code = employee.job_code AND
                job.job_grade = employee.job_grade AND
                job.job_country = employee.job_country) AND
         salary <= (SELECT max_salary FROM job WHERE
                job.job_code = employee.job_code AND
                job.job_grade = employee.job_grade AND
                job.job_country = employee.job_country))
    );
```

Notice the complex check constraint on SALARY. 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.

Skim the file, TABLES.SQL, and look at the rest of the table definitions. Make sure you understand them. Notice that there is a CREATE INDEX statement after each table definition. Indexes will be explained later in this part of the tutorial.

After editing the CONNECT statement at the beginning of the file, choose File | Run an ISQL Script... and select TABLES.SQL.

Then confirm that the tables have been created by choosing View | Metadata Information..., select Table, and choose OK. You will see a list of all the table names in the ISQL Output area.



Creating indexes

Indexes are used to improve the speed of data access for a table. An index identifies columns that can be used to efficiently retrieve and sort rows in the table. Because a primary key uniquely identifies a row, it is often also defined as the index of the table. The CREATE INDEX statement is used to define indexes in SQL.

An index is based on one or more columns in a table. Indexes can also enforce uniqueness and referential integrity constraints. A unique index will prevent duplicate values in the columns in the index.

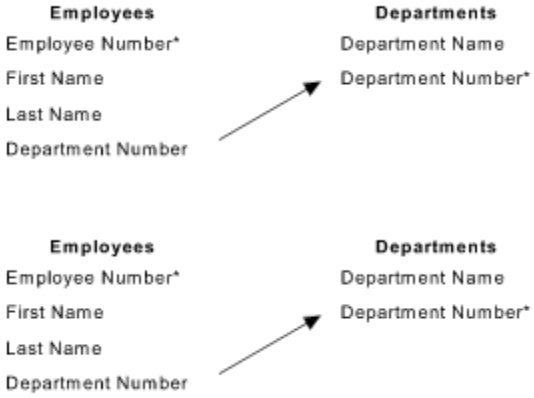
An index is created with the CREATE INDEX statement. Here is the simplified syntax:

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

For example, TABLES.SQL created an index called NAMEX for the EMPLOYEE table, as follows:

```
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.



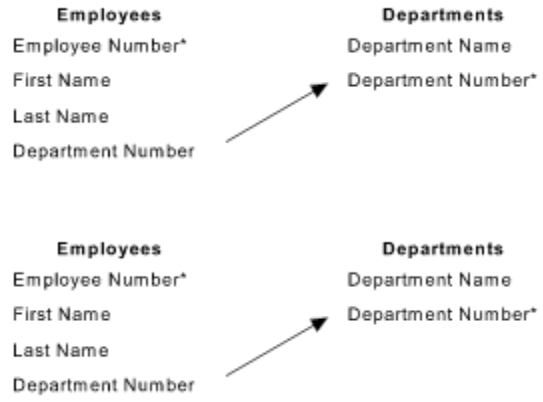
Preventing duplicate row entries

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.

TABLES.SQL defined a unique index named PRODTYPEX, on the PROJECT table as follows:

```
CREATE UNIQUE INDEX PRODTYPEX ON PROJECT (PRODUCT, PROJ_NAME);
```

Note: For unique indexes defined on multiple columns, like PRODTYPEX in the example above, the same value may be entered within individual columns, but the combination of values entered in all columns defined for the index must be unique.



Modifying indexes

You can modify an index definition to change the columns that are indexed, prevent insertion of duplicate entries, or specify a different sort order.

To change the definition of an index, follow these steps:

1. Use ALTER INDEX to make the current index inactive.
2. Drop the current index.
3. Create a new index and give it the same name as the dropped index.

Choose View | Metadata Information... and select Index from the drop-down list of object types. Enter NAMEX in the Object Name field. The ISQL Output area will display the definition of the index:

```
NAMEX INDEX ON EMPLOYEE (LAST_NAME, FIRST_NAME)
```

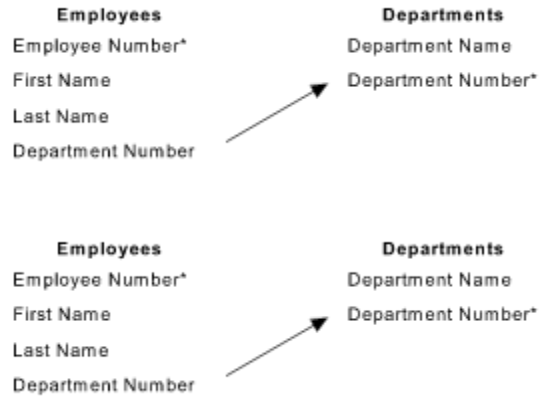
For example, suppose you need to prevent duplicate entries in the NAMEX index you defined for the EMPLOYEE table with a UNIQUE keyword. First, make the current index inactive, then drop it. Enter:

```
ALTER INDEX NAMEX INACTIVE;
DROP INDEX NAMEX;
```

Then redefine NAMEX to include the UNIQUE keyword:

```
CREATE UNIQUE INDEX NAMEX ON EMPLOYEE (LAST_NAME, FIRST_NAME);
```

This index will now prevent entries in the EMPLOYEE table with the same first and last names as an existing row.



Creating views

A view is a virtual table. Views are not physically stored in the database, but appear as "real" tables. A view can contain data from one or more tables or other views and can store an often-used query or set of queries in the database. The CREATE VIEW statement is used to define views in SQL.

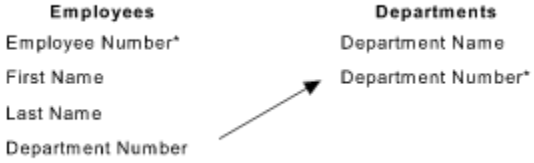
You are now going to create a view called PHONE_LIST that maintains a phone list of employees from the EMPLOYEE and DEPARTMENT tables.

Enter the following statement:

```
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;
```

This statement creates a view called PHONE_LIST from columns in the EMPLOYEE and DEPARTMENT tables. After you populate these tables with data you will be able to query this view just as you would a table.

You have created all the tables for the full EMPLOYEE database. In the next series of lessons, you will populate (add data to) the database.



PART THREE:
Populating the database

This series of lessons lets you populate (add data to) the database created in the introductory parts of the tutorial. It also introduces basic SQL statements, shows you how to manipulate data using those statements, and lets you create a simple view from which you can select data.
SQL statements covered in this section include:

INSERT, for adding data.

UPDATE, for modifying data.

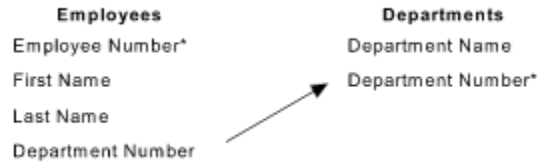
DELETE, for removing data.

Inserting data

You previously learned the basic form of the INSERT statement:

```
INSERT INTO table_name (col1, col2, ...) VALUES (val1, val2, ...);
```

Now, you will use this form again to insert data into the EMPLOYEE database.



Inserting data using column values

Open the file, INSERTS.SQL, with a text editor. It contains a number of INSERT statements to add data to the database. The first group of statements inserts values into the COUNTRY table, for example:

```
INSERT INTO COUNTRY (COUNTRY, CURRENCY) VALUES ("USA", "DOLLAR");
```

The next group inserts values into the DEPARTMENT table. For example:

```
INSERT INTO DEPARTMENT
   (DEPT_NO, DEPARTMENT, HEAD_DEPT, BUDGET, LOCATION, PHONE_NO)
VALUES
   ("000", "CORPORATE HEADQUARTERS", NULL, 1000000, "MONTEREY",
   "(408) 555-1234");
```

There are groups of statements to populate the JOB and EMPLOYEE tables also.

Make sure the CONNECT statement at the beginning of the file contains the correct database name, user name, and password. Choose File | Run an ISQL Script... and then select INSERTS.SQL to execute the script and insert the data into the tables.

To make the changes to the database permanent, you must commit your work by choosing File | Commit Work.

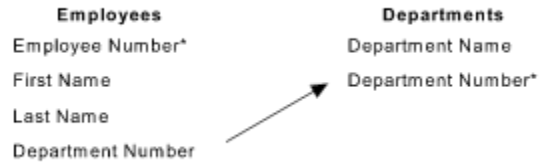
Confirm that the data has been inserted correctly with SELECT, for example:

```
SELECT * FROM DEPARTMENT;
```

Try selecting from the COUNTRY, JOB, and EMPLOYEE tables, too. Notice that all the data in the LANGUAGE_REQ column in the JOB table is NULL. This is because this column is an array, and you cannot insert data into an array using ISQL.

The script file, INSERTS2.SQL, inserts data into the other tables in the database. View this file and then run INSERTS2.SQL by choosing File | Run an ISQL Script.... Commit the changes to the database, by choosing File | Commit Work.

SELECT from the PROJECT, CUSTOMER, and SALES tables to confirm that the data has been successfully inserted. Notice that all the data in the PROJ_DESC column of the PROJECT table is NULL. This is because this column is a BLOB, and you cannot insert data into a BLOB using ISQL.



Inserting data from an external file

Note: This section covers an optional topic and may be skipped without losing any continuity. However, it is an important topic not covered in detail elsewhere in the documentation.

An external table is a special kind of table that stores its data in an ASCII file separate from the database. It may occasionally want to import data from an ASCII file into a database (for example, if the data was originally entered in another application or at a remote location).

You can populate a table with data from a formatted ASCII file by following these steps:

1. Create the table you want to populate. Often this table will already exist in the database.
2. Create an ASCII file on the server containing the data, formatted strictly to conform to the column definitions of the table in step one. Depending on how the file originated (for example, from a desktop application), you may have to edit the file manually with a text editor to ensure that it is formatted correctly.
3. Create a temporary external table that has all the columns that will get data from the external file. It is usually easiest to create all the fields as CHAR(n), even if they will contain numeric data. The table must also have a CHAR(1) column (usually called EOL) to take the end-of-line character.
4. Insert the data into the destination table using INSERT with a SELECT clause. InterBase's automatic type conversion feature will ensure that the data in each column is automatically converted from CHAR to the appropriate data type.

For example, suppose a salesman on the road has been keeping his sales records on his laptop computer in a spreadsheet application. When he gets back to the office, one way he could enter these records into the SALES table would be to export the information to a text file and then import the data into the database through an external table. So, in this example, you do not need to perform step one, because the SALES table already exists in the database.

The next step is to create the data file. The sales data is in the file, SALES.DAT, in the EXAMPLES\TUTORIAL directory. View this file now with the Notepad editor. It looks something like this:

```
V92E0340 1004  11 shipped 15-OCT-1992 16-OCT-1992 17-OCT-1992
V92J1003 1010  61 shipped 26-JUL-1992  4-AUG-1992 15-SEP-1992
V93J2004 1010 118 shipped 30-OCT-1993  2-DEC-1993 15-NOV-1993
```

(To see the entire lines when viewing in Notepad, use the horizontal scroll bar.)

Each line in this file corresponds to a row of data (record) in the SALES table, and each item of text on a line is a value to be inserted into a field in the row. The text is padded with spaces where necessary to make each field have the specified number of characters, even at the end of each line. The first item in each line (for example "V92E0340") is a value for the PONUMBER column, the second (for example "1004") is a value for the CUST_NO column, and so on. It is crucial that the items on each line always are in the same order.

For the server to be able to access this file, you must copy it to the server platform (to a disk to which the server has direct access). Use the standard FTP utility or operating system copy command to copy SALES.DAT to the directory on the server where your database resides. That completes step two of the process.

The next step is to create a temporary external table in the database called SALES_EXT. Look at the file,

SALES_XT.SQL. It contains the following CREATE TABLE statement:

```
CREATE TABLE SALES_EXT EXTERNAL "/PATH/SALES.DAT"
(PO_NUMBER CHAR(10),
   CUST_NO CHAR(12),
   SALES_REP CHAR(10),
   ORDER_STATUS CHAR(13),
   ORDER_DATE CHAR(12),
   SHIP_DATE CHAR(12),
   DATE_NEEDED CHAR(12),
   PAID CHAR(7),
   QTY_ORDERED CHAR(12),
   TOTAL_VALUE CHAR(12),
   DISCOUNT CHAR(16),
   ITEM_TYPE CHAR(8),
   EOL CHAR(1));
```

Notice the keyword EXTERNAL at the top, followed by a file path in quotes. You must edit this path to specify the location on the server to which you copied SALES.DAT in the previous step. All the columns in SALES_EXT are defined as CHAR (character) values. Notice also the EOL column. This is a dummy column to contain the carriage return at the end of each line of data in SALES.DAT.

Input this definition by choosing File | Run an ISQL Script... and selecting SALES_XT.SQL in the EXAMPLES\TUTORIAL directory. At this point, you have an external table which has data stored in a file on the server. You can query data from this table as if it were an ordinary table, but you cannot modify the data, because it does not actually reside in the database, but in the file. Enter the following statement:

```
SELECT * FROM SALES_EXT;
```

You will see the data from the data file in the ISQL Output area. Now you have completed step three of the procedure.

In the final step, you will migrate the data from the external table into the real SALES table. Look at the file, MIGRATE.SQL. It contains the following INSERT statement:

```
INSERT INTO SALES
(PO_NUMBER, CUST_NO, SALES_REP, ORDER_STATUS, ORDER_DATE,       SHIP_DATE,
DATE_NEEDED, PAID, QTY_ORDERED, TOTAL_VALUE, DISCOUNT,    ITEM_TYPE)
SELECT
PO_NUMBER, CUST_NO, SALES_REP, ORDER_STATUS, ORDER_DATE, SHIP_DATE,
   DATE_NEEDED, PAID, QTY_ORDERED, TOTAL_VALUE, DISCOUNT, ITEM_TYPE
   FROM SALES_EXT;
```

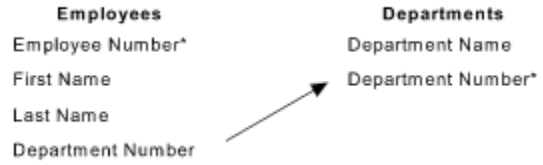
This statement selects values from the SALES_EXT table (excluding the EOL delimiter) and inserts them into rows in the SALES table, migrating the data from the file to the SALES table.

Edit the CONNECT statement at the beginning of this file and specify the server and database you are using. Then input this statement by choosing File | Run ISQL Script... and choosing MIGRATE.SQL from the EXAMPLES\TUTORIAL directory.

Now enter:

```
SELECT * FROM SALES;
```

and you will see the data that was in the SALES.DAT file has been inserted into the SALES table. Notice that the non-character columns have been converted to the appropriate data type automatically.



Updating data

To change values for one or more rows of data, use the UPDATE statement. 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.

For example, increase the salary of salespeople by $2,000, by updating the EMPLOYEE table as follows:

```
UPDATE EMPLOYEE
   SET SALARY = SALARY + 2000
   WHERE JOB_CODE = "Sales";
```

To make a more specific update, make the WHERE clause more restrictive. For example, instead of increasing the salary for all salespeople, you could 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.

Be sure to commit your work to make it permanent by choosing File | Commit Work (if it hasn't already been committed; if it has been, the item is grayed).

Updating with a script file

Open the file, UPDATES.SQL, with Notepad. As you can see, it contains a number of UPDATE statements to update the DEPARTMENT, EMPLOYEE, SALARY_HISTORY, and CUSTOMER tables. Run this file by choosing File | Run ISQL Script.... Confirm that the updates have been made.

Updating using a subquery

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:

```
SELECT DEPT_NO FROM EMPLOYEE
   WHERE FULL_NAME = "Young, Katherine";
```

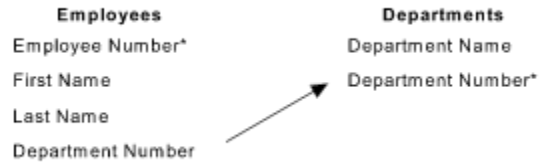
This query returns "623" as the department. Then, using 623 as the search condition in an UPDATE, you could change the manager number of all the employees in the department with the following statement (do not enter this statement):

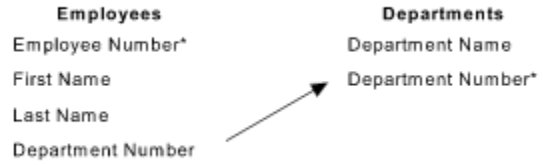
```
UPDATE DEPARTMENT
   SET MNGR_NO = 107
   WHERE DEPT_NO = "623";
```

Instead of doing this, a more efficient way is to combine the two statements together using a subquery as follows. Enter this statement:

```
UPDATE DEPARTMENT
SET MNGR_NO = 107
WHERE DEPT_NO = (SELECT DEPT_NO FROM EMPLOYEE
   WHERE FULL_NAME = "Young, Katherine");
```

Confirm the result by selecting from the department table, and then choose File | Commit Work to make the update permanent.





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.

Caution: A WHERE clause is not required in a DELETE statement. If you fail to include a WHERE clause, you will delete all rows in the table.

Enter the following statement to delete rows from the EMPLOYEE table for which the JOB_CODE column is "MNGR." In other words, managers are removed from the table. Enter:

```
DELETE FROM EMPLOYEE
   WHERE JOB_CODE = "Mngr";
```

You can restrict deletions further by combining search conditions. For example, enter the following statement to delete records of all sales reps hired before 10 July 1993:

```
DELETE FROM EMPLOYEE
   WHERE JOB_CODE = "SRep" AND HIRE_DATE < "10-Jul-1993";
```

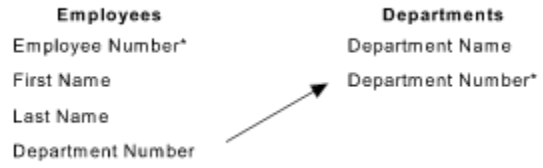
Confirm that these statements deleted the appropriate records by entering the following query

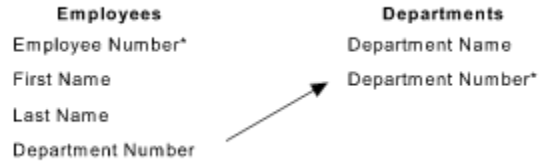
```
SELECT EMP_NO, JOB_CODE, HIRE_DATE FROM EMPLOYEE;
```

You should not see any records with a JOB_CODE of "Mngr" or any records with a JOB_CODE of "SRep" and a hire date before 10 July 1993.

Because you really did not want to delete those records from the table, roll back the changes to the database by choosing File | Rollback Work. Choose Previous to recall the previous SELECT query and then Run to run it. You should now see the deleted records displayed.

Caution: If you do not roll back these deletes, you will not get the correct results when you do the rest of the tutorial.





Deleting data using a subquery

The previous section used a subquery to update data. DELETE statements can also use subqueries. To remove all employees who are in the same department as Katherine Young, including Katherine Young herself, you could first determine Katherine Young's department number:

```
SELECT DEPT_NO FROM EMPLOYEE
WHERE FULL_NAME = "Young, Katherine";
```

This query returns "623" as the department number. Then, using 623 as the search condition in a DELETE, you would enter :

```
DELETE FROM EMPLOYEE
WHERE DEPT_NO = "623";
```

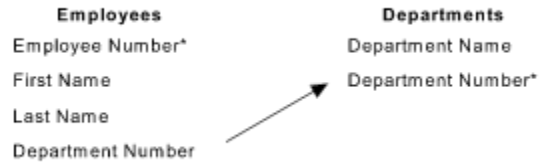
The other way to remove the desired rows is to combine the two previous statements using a subquery. In this case, the DELETE statement becomes:

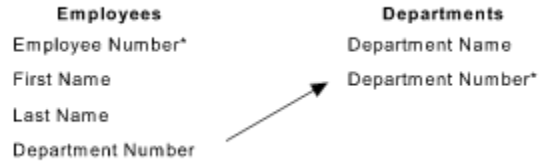
```
DELETE FROM EMPLOYEE
WHERE DEPT_NO = (SELECT DEPT_NO FROM EMPLOYEE
WHERE FULL_NAME = "Young, Katherine");
```

Try this and confirm that it deletes the appropriate rows. Roll back the deletions by choosing File | Rollback Work.

PART FOUR:
Retrieving data

This portion of the tutorial provides further practice with the SQL SELECT statement.





Overview of SELECT

Part One of the tutorial presented the simplest form of the SELECT statement. The full syntax is much more complex, but, as you'll see, its rich syntax is the source of much of its power.
Here's a distilled version of the SELECT syntax:

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

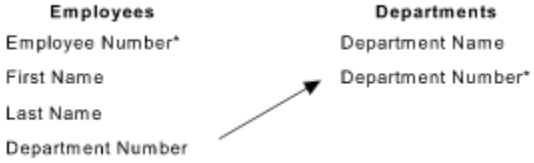
This distilled version has six main keywords. A keyword and its associated information is called a clause. The clauses are:

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 <sort_order>	Specifies the sort order of rows returned by a SELECT.

The order of the clauses in the SELECT statement is important, but SELECT and FROM are the only required clauses.
You have already used some basic SELECT statements to retrieve data from single tables. SELECT can also retrieve data from multiple tables, by listing the table names in the FROM clause, separated by commas. For example, enter the following SQL statement:

```
SELECT DEPARTMENT, DEPT_NO, FULL_NAME, EMP_NO
   FROM DEPARTMENT, EMPLOYEE
   WHERE DEPARTMENT = "Engineering" AND MNGR_NO = EMP_NO;
```

This statement retrieves the specified fields for the employee who is the manager of the Engineering department.
Sometimes a column name occurs in more than one table in the same query. If so, columns must be distinguished from one another by preceding each column name with the table name and a dot (.).



Selecting from a view

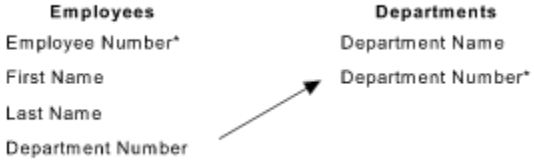
Recall the view named PHONE_LIST you created earlier in the tutorial. You can select from this view just like a table. Try this by entering the statement:

```
SELECT * FROM PHONE_LIST;
```

You will see output like this:

```
EMP_NO FIRST_NAME  LAST_NAME PHONE_EXT LOCATION      PHONE_NO
====== ==========  ========= ========= ============= =============
12     Terri       Lee       256       Monterey      (408) 555-1234
105    Oliver H.   Bender    255       Monterey      (408) 555-1234
85     Mary S.     MacDonald 477       San Francisco (415) 555-1234
. . .
```

As you can see, the output looks just as if there were a table called PHONE_LIST containing the pertinent information.



Removing duplicate rows with DISTINCT

Suppose you want to retrieve a list of all the valid job codes in the EMPLOYEE database. Enter 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.

Revise the previous query by clicking on the Previous button and editing the command as follows:

```
SELECT DISTINCT JOB_CODE FROM JOB;
```

As you can see, each job code is listed once in the results.

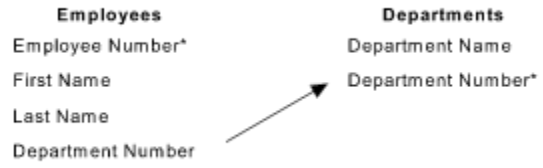
What happens if you specify another column when using DISTINCT? Enter the following SELECT statement:

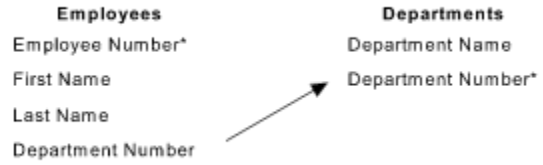
```
SELECT DISTINCT JOB_CODE, JOB_GRADE FROM JOB;
```

This query produces:

```
JOB_CODE JOB_GRADE
======== =========
Accnt    4
Admin    4
Admin    5
CEO      1
CFO      1
Dir      2
Doc      3
Doc      5
Eng      2
Eng      3
En       4
Eng      5
. . .
```

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 of the SELECT statement follows the SELECT and FROM clauses. If an ORDER BY clause is used, the WHERE clause must precede it. The WHERE clause tests data to see whether it meets certain conditions, and the SELECT statement only returns the rows that meet the condition. For example, the statement:

```
SELECT LAST_NAME, FIRST_NAME, PHONE_EXT
   FROM EMPLOYEE
   WHERE LAST_NAME = "Green";
```

returns only rows for which LAST_NAME is "Green". The text following the WHERE keyword, in this case:

```
LAST_NAME = "Green"
```

is called a search condition, because a SELECT statement searches for rows that meet the condition. Search conditions have the following general form:

```
WHERE condition;
```

In this clause:

```
condition = column operator value [log_operator condition]
value = value arith_operator value
```

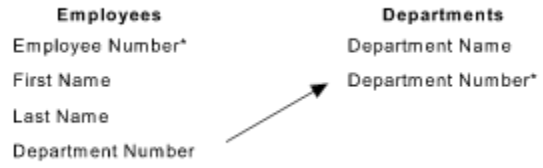
column is the column name in the table being queried, operator is a comparison operator (described in the following table), value is a value or a range of values compared against the column, described in the next table. A condition can be composed of two or more conditions as operands of logical operators. A value can be composed of two or more values as operands of arithmetic operators.
Search conditions use the following types of 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 negate a condition. The keywords NOT, AND, and OR.

Search conditions can use the following types of values:

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.

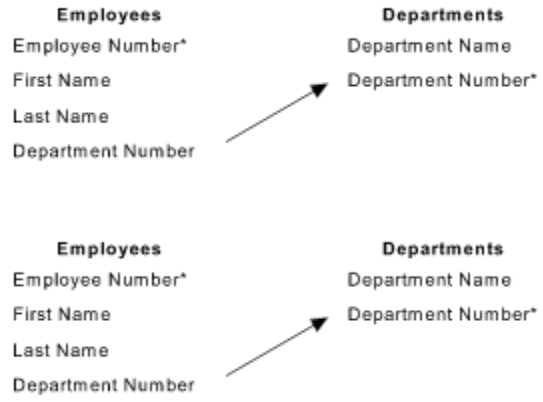
When a row is compared to a search condition, one of three values is returned:



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 unknown value that could not be evaluated because of a NULL value.



Comparison operators

InterBase uses all the standard comparison operators: greater than (>), less than (<), equal to (=), and so on. These operators can be used to compare numeric or alphabetic (text) values. Text literals must be quoted. Numeric literals must not be quoted.

Important: String comparisons are case sensitive.

A previous example had a WHERE clause that compared a column to a literal value:

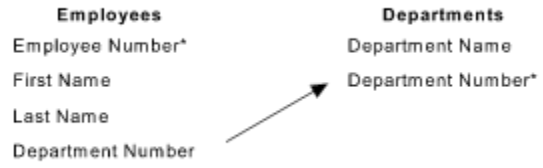
```
SELECT LAST_NAME, FIRST_NAME, PHONE_EXT
   FROM EMPLOYEE
   WHERE LAST_NAME = "Green";
```

This query will retrieve records from the EMPLOYEE table for which the last name is "Green". If you change the equal sign to a greater than (>) sign, it will retrieve rows for which the last name is alphabetically greater than (after) "Green". Likewise, if you change it to less than (<). Try these different queries to see how the results change.

You can negate any expression with the negation operators !, ^, and ~. These operators are all synonyms for NOT. For example, to retrieve all rows except those for which the last name is "Green", change the search condition to:

```
WHERE NOT LAST_NAME = "Green"
```

Try negating some of the previous queries to see how the results change.



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.
There are a great many pattern matching operators. This section will only discuss some of the most commonly used ones: LIKE, STARTING WITH, IS NULL, and BETWEEN.

LIKE Operator

The LIKE operator lets you use wildcard characters in matching text. Wildcard characters are characters that have special meanings when used in a search condition. A percent sign (%) will match zero or more characters. An underscore (_) will match any single character.
For example, enter this statement in the SQL Statement area:

```
SELECT LAST_NAME, FIRST_NAME, EMP_NO FROM EMPLOYEE
   WHERE LAST_NAME LIKE "%an";
```

You should see the following results:

```
LAST_NAME            FIRST_NAME      EMP_NO
==================== =============== ======
Ramanathan           Ashok           45
Steadman             Walter          46
```

As you can see from the results, this statement retrieves rows for employees whose last names end with "an", because the percent sign will match any characters. LIKE distinguishes between uppercase and lowercase.
Now enter the following statement:

```
SELECT LAST_NAME, FIRST_NAME, EMP_NO FROM EMPLOYEE
   WHERE LAST_NAME LIKE "_e%";
```

This statement retrieves rows for employees whose last name has "e" as the second letter. The underscore will match any one character in the last name.

STARTING WITH Operator

The STARTING WITH operator tests whether a value starts with a particular character or sequence of characters. As with the LIKE operator, STARTING WITH distinguishes between uppercase and lowercase. STARTING WITH does not support wildcard characters.
The following statement retrieves employee last names that start with "Ke":

```
SELECT LAST_NAME, FIRST_NAME
   FROM EMPLOYEE
   WHERE FIRST_NAME STARTING WITH "Ke";
```

The CONTAINING operator is similar to STARTING WITH, except it matches strings containing the specified string, anywhere, within the string.

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 (that is, absent). To test for the presence of any value, use IS NOT NULL.

For example, to retrieve the names of employees who do not have phone extensions, enter:

```
SELECT LAST_NAME, FIRST_NAME, PHONE_EXT
   FROM EMPLOYEE
   WHERE PHONE_EXT IS NULL;
```

You should see the following results:

```
LAST_NAME            FIRST_NAME           PHONE_EXT
==================== ===========          =========
Sutherland           Claudia              <null>
Glon                 Jacques              <null>
Osborne              Pierre               <null>
```

To retrieve the names of employees who do have phone extensions, enter:

```
SELECT LAST_NAME, FIRST_NAME, PHONE_EXT
   FROM EMPLOYEE
   WHERE PHONE_EXT IS NOT NULL;
```

The results should be all the employee records except the three retrieved by the previous query.

Comparing Against a Range or List of Values

The previous sections described operators to compare columns to a single value. The BETWEEN and IN operators enable comparison against multiple values.

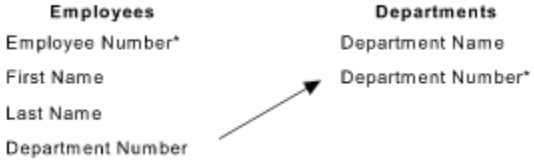
BETWEEN tests whether a value falls within a range. For example, to retrieve names of employees whose salaries are between $100,000 and $250,000, inclusive, enter:

```
SELECT LAST_NAME, FIRST_NAME, SALARY
   FROM EMPLOYEE
   WHERE SALARY BETWEEN 100000 AND 250000;
```

The IN operator searches for values matching one of the values in a list. For example, to retrieve the names of all employees in departments 120, 600, and 623, enter:

```
SELECT DEPT_NO, LAST_NAME, FIRST_NAME, SALARY FROM EMPLOYEE
   WHERE DEPT_NO IN (120, 600, 623);
```

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.



Logical operators

Up until now, the examples presented have included only one search condition. However, you can include any number of search conditions in a WHERE clause by combining them with the logical operators AND or OR.

When AND appears between search conditions, both conditions must be true for a row to be retrieved. For example, enter this query:

```
SELECT DEPT_NO, LAST_NAME, FIRST_NAME, HIRE_DATE
   FROM EMPLOYEE
   WHERE DEPT_NO = 623 AND HIRE_DATE > "01-Jan-1992";
```

The query returns information on employees in department 623 who were hired after 1 January 1992.

When OR appears between search conditions, either search condition can be true for a row to be retrieved. Choose Previous to recall the previous query and change AND to OR. As you can see, the results are quite different, because the query retrieves rows for employees who are in department 623 or who were hired before 1 January 1992.

As another example of using OR in a search condition, enter this query:

```
SELECT CUSTOMER, CUST_NO, COUNTRY
FROM CUSTOMER
   WHERE COUNTRY = "USA" OR COUNTRY = "Canada";
```

This query retrieves customer records for customers in the US or Canada.

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 1 January 1992. Try entering 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
```

The WHERE clause was not interpreted the way you meant it 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 parentheses to override normal precedence. In the example, place parentheses around the two departments so they are tested against the AND operator as a unit:

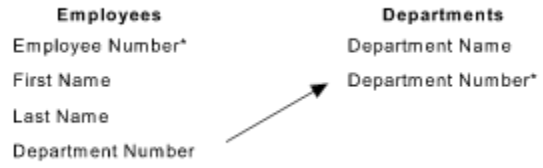
```
SELECT LAST_NAME, FIRST_NAME, HIRE_DATE, DEPT_NO
   FROM EMPLOYEE
   WHERE (DEPT_NO = 623 OR DEPT_NO = 600)
   AND HIRE_DATE > "01-JAN-1992";
```

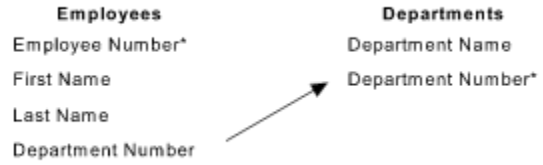
This displays the results you want:

```
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.

Tip: To avoid confusion with operator precedence, always use parentheses to group operations in complex search conditions.





Using subqueries

Suppose you want to retrieve a list of employees who work in the same country as a particular employee whose ID is 144. You would first need to find out what country this employee works in. Enter this query:

```
SELECT JOB_COUNTRY FROM EMPLOYEE
   WHERE EMP_NO = 144;
```

This query returns "USA." With this information, you can form your next query:

```
SELECT EMP_NO, LAST_NAME FROM EMPLOYEE
   WHERE JOB_COUNTRY = "USA";
```

This query returns a list of employees in the USA, the same country as employee number 144. You can obtain the same result by combining the two queries:

```
SELECT EMP_NO, LAST_NAME, JOB_COUNTRY FROM EMPLOYEE
   WHERE JOB_COUNTRY =
      (SELECT JOB_COUNTRY FROM EMPLOYEE
      WHERE EMP_NO = 144);
```

This statement uses a subquery, a SELECT statement inside the WHERE clause of another SELECT statement. A subquery works like a search condition to restrict the number of rows returned by the outer, or parent, query.

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. Because the WHERE clause is testing for a single value, the subquery must return a single value; otherwise, the statement produces an error. As long as a subquery retrieves a single value, you can use it in any search condition that tests for a single value.

If a subquery returns more than one value, you must use an operator that tests against more than one value. IN is such an operator. The following example retrieves all management-level employees. It uses a subquery that returns any job grade lower than or equal to 2:

```
SELECT FIRST_NAME, LAST_NAME, JOB_GRADE
   FROM EMPLOYEE
   WHERE JOB_GRADE IN
      (SELECT JOB_GRADE FROM JOB WHERE JOB_GRADE <= 2);
```

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.

Suppose you want to see how salaries compare to the salaries of employees in department 623. First you would need an expression that returns employee salaries for department 623. The following query returns

that information:

```
SELECT SALARY FROM EMPLOYEE
   WHERE DEPT_NO = 623;
```

and produces this output:

```
                SALARY
======================
              60000.00
              62000.00
              50000.00
              35000.00
              60000.00
```

The previous query can now be used as a subquery in the next several examples. To see which employees have the same salary as those in department 623, enter:

```
SELECT LAST_NAME, DEPT_NO FROM EMPLOYEE
   WHERE SALARY IN
   (SELECT SALARY FROM EMPLOYEE WHERE DEPT_NO = 623);
```

The IN operator tests whether a value equals one of the values in a list. In this case, the value being tested is SALARY, and the list comes from a subquery. The statement yields this output:

```
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
```

The output shows that two employees, Hall and Montgomery, earn the same as someone in department 623.

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);
```

to yield this output:

```
LAST_NAME                       SALARY
==================== ======================

Nelson                        98000.00
Young                         90000.00
Lambert                       95000.00
Forest                        72000.00
Weston                        77000.00
Papadopoulos                  80000.00
. . .
```

This example uses the ALL operator. The statement tests against all values in the subquery. If the salary is greater, the row is retrieved. 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

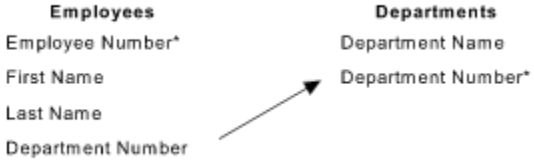
Instead of testing against all values returned by a subquery, you can rewrite the example to test for at

least one value. Enter this query:

```
SELECT LAST_NAME, SALARY FROM EMPLOYEE
   WHERE SALARY > ANY
   (SELECT SALARY FROM EMPLOYEE WHERE DEPT_NO = 623);
```

This statement retrieves rows for which SALARY is greater than any of the values from the subquery, so this statement retrieves records of employees whose salary is greater than any salary in department 623. 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.

For example, 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 is:

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

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

Enter the following query to retrieve the average salary of employees from the EMPLOYEE table:

```
SELECT AVG(SALARY) FROM EMPLOYEE;
```

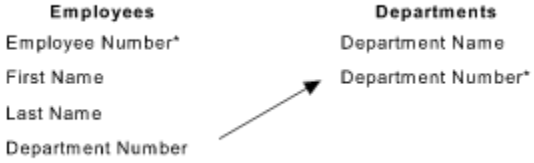
A single SELECT can retrieve multiple aggregate functions. Enter this statement to retrieve the number of employees, the earliest hire date, and the total salary paid to all employees:

```
SELECT COUNT(EMP_NO), MIN(HIRE_DATE), SUM(SALARY)
   FROM EMPLOYEE;
```

The result is:

```
        COUNT          MIN                       SUM
=========== =========== ======================
           42 28-DEC-1988           115390775.00
```

Sometimes, a value involved in an aggregate calculation is NULL or unknown. In this case, 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.



Grouping query results

You can use the optional GROUP BY clause to organize data retrieved from aggregate functions. Each column name that appears in a GROUP BY clause must also appear in the SELECT clause. And each SELECT clause in a query can have only one GROUP BY clause.

Suppose you want to display the maximum allowable salary for each job code and job grade in the United States. Enter this query:

```
SELECT JOB_CODE, JOB_GRADE, MAX_SALARY, JOB_COUNTRY
   FROM JOB WHERE JOB_COUNTRY = "USA";
```

You should see these results (shown in part):

```
JOB_CODE JOB_GRADE              MAX_SALARY JOB_COUNTRY
======== ========= ====================== ===============
CEO              1              250000.00 USA
CFO              1              140000.00 USA
VP               2              130000.00 USA
Dir              2              120000.00 USA
Mngr             3              100000.00 USA
Mngr             4               60000.00 USA
Admin            4               55000.00 USA
Admin            5               40000.00 USA
. . .
```

Now suppose you want to total the salaries for each group of job codes. In other words, find the maximum total possible salary for all job codes, regardless of job grade. To do so, use the SUM() function and group the results by job code. Enter the following query:

```
SELECT JOB_CODE, SUM(MAX_SALARY)
   FROM JOB WHERE JOB_COUNTRY = "USA"
   GROUP BY JOB_CODE;
```

to produce the desired output (shown in part):

```
JOB_CODE                     SUM
======== ======================
Accnt                  55000.00
Admin                  95000.00
CEO                   250000.00
CFO                   140000.00
Dir                   120000.00
Doc                   100000.00
Eng                   300000.00
. . .
```

Note the difference in the results. The first query produces four entries for engineers (Eng). The second query totals the salaries for these four entries and displays a single row as the result.

As another example, the DEPARTMENT table lists budgets for each department in the company. Each department also has a head department to which it reports. Suppose you want to find out the total budget

for each head department. To do so, you would need to add the budgets for individual departments and group the results by each head department. Enter the following query:

```
SELECT HEAD_DEPT, SUM(BUDGET)
   FROM DEPARTMENT
   GROUP BY HEAD_DEPT;
```

to produce these results:

```
HEAD_DEPT                    SUM
========= ======================
000                   3500000.00
100                   3800000.00
110                    800000.00
120                   1300000.00
600                   2350000.00
620                   1350000.00
670                   1310000.00
<null>                1000000.00
```

Using the HAVING clause

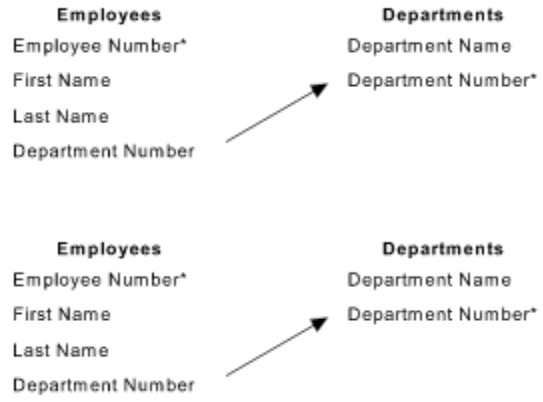
Just as a WHERE clause reduces the number of rows returned by a SELECT clause, the HAVING clause can be used to reduce the number of rows returned by a GROUP BY clause. Like the WHERE clause, a HAVING clause has a search condition. In a HAVING clause, the search condition typically corresponds to an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

For example, you can modify the previous query to display only the head departments whose total budgets are greater than 2,000,000. Change the query as follows:

```
SELECT HEAD_DEPT, SUM(BUDGET)
   FROM DEPARTMENT
   GROUP BY HEAD_DEPT
   HAVING SUM(BUDGET) > 2000000;
```

This query produces the following results:

```
HEAD_DEPT          SUM
========= ===========
000         3500000.00
100         3800000.00
600         2350000.00
```



Using the ORDER BY clause

By default, a query retrieves rows in "natural order," the order it finds them in a table. Because internal table storage is typically unordered, retrieval is unordered as well. The ORDER BY clause sorts results according to a column you specify. Every column in the ORDER BY clause must also appear in the SELECT clause of the statement.

For example, enter the statement:

```
SELECT LAST_NAME, FIRST_NAME, PHONE_EXT
   FROM EMPLOYEE
   ORDER BY LAST_NAME;
```

As you can see, this query sorts results by employee's last name.

By default, ORDER BY sorts in ascending order, in this case from A to Z. To sort in descending order instead, use the DESC keyword. Enter:

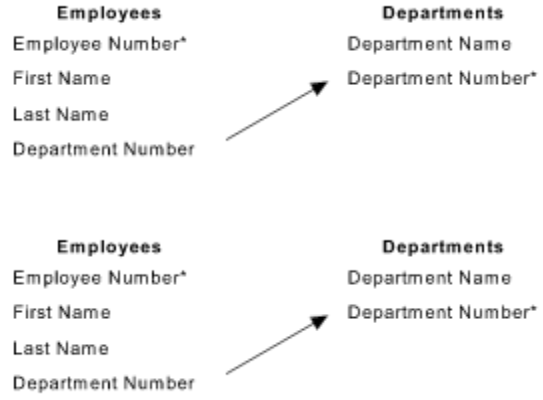
```
SELECT LAST_NAME, FIRST_NAME, PHONE_EXT
   FROM EMPLOYEE
   ORDER BY LAST_NAME DESC;
```

In the previous two examples, the sort column contains characters, so ORDER BY performs an alphanumeric sort. If a sort column contains numbers, results are sorted numerically.

ORDER BY can also sort results by more than one column. For example, if several employees have the same last name, you can sort by both first name and last name using the following SELECT statement:

```
SELECT LAST_NAME, FIRST_NAME, PHONE_EXT
   FROM EMPLOYEE
   ORDER BY LAST_NAME DESC, FIRST_NAME;
```

In this case, the results are initially sorted by last name, in descending order. For employees with the same last name, data is further sorted by first name. The first name is also sorted in descending order because once you specify a column's sort order, it applies to all subsequent columns until you specify another sort order. To explicitly sort in ascending order, use the ASC keyword.



Joining tables

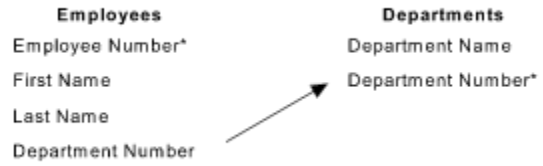
Joins enable a SELECT statement to retrieve data from two or more tables in a database. The tables are listed in the FROM clause. The optional ON clause can reduce the number of rows returned, and the WHERE clause can further reduce the number of rows returned.

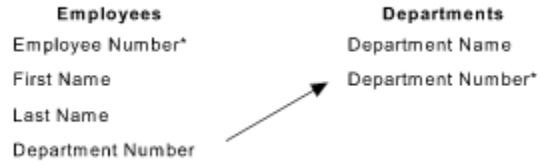
From the information in a SELECT that describes a join, InterBase builds a table that contains the results of the join operation, the result table, sometimes also called a dynamic or virtual table.

InterBase supports two basic 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 result table. Inner joins are the more common type because they restrict the data returned and show a clear relationship between two or more tables.

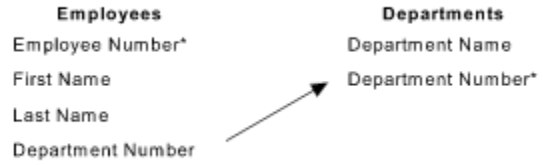
Outer joins link rows in tables based on specified join conditions but return rows whether they match the join conditions or not. Outer joins are useful for viewing joined rows against a background of rows that do not meet the join conditions.



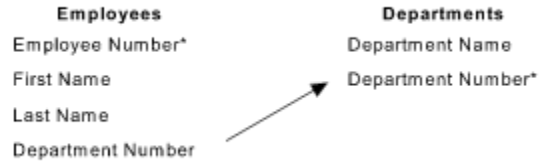


Inner joins

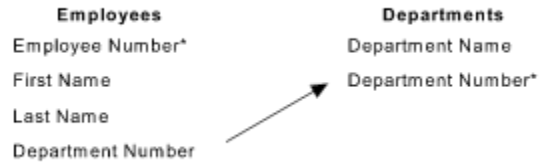
There are three types of inner joins:



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 types of joins, but for simplicity's sake they may 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 WHERE clause. The simplified syntax is:

```
SELECT <columns>
   FROM <left_table> [INNER] JOIN <right_table>
     [ON <searchcondition>]
   [WHERE <searchcondition>];
```

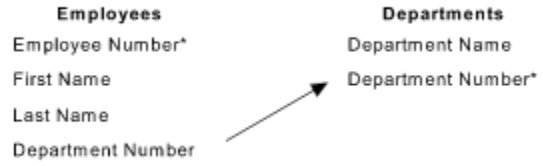
Search conditions based on a column in the right table can be specified in an optional ON clause following the right table reference.

For example, consider the following query including an inner join:

```
SELECT D.DEPARTMENT, D.MNGR_NO, E.SALARY
  FROM DEPARTMENT D JOIN EMPLOYEE E
```

```
ON D.MNGR_NO = E.EMP_NO
   AND E.SALARY*2 > (SELECT SUM(S.SALARY) FROM EMPLOYEE S
   WHERE D.DEPT_NO = S.DEPT_NO)
ORDER BY D.DEPARTMENT;
```

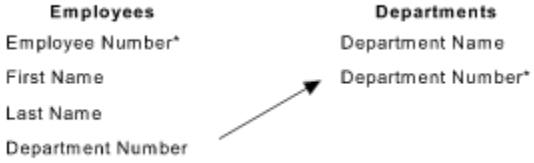
Examine this statement in detail. The SELECT clause uses correlation names D for DEPARTMENT and E for EMPLOYEE (as specified in the FROM clause) to select the department name and manager number from DEPARTMENT and the manager's salary from the EMPLOYEE table.
The ON clause states a compound join condition:



The MNGR_NO column in the DEPARTMENT table must match the EMP_NO column in EMPLOYEE.

The manager's salary times two (E.SALARY*2) must be greater than the sum of all employees' salaries in the department. In other words, the manager's salary must be greater than half the sum of all salaries in the department.
Enter the above statement. You should see the following results:

```
DEPARTMENT                MNGR_NO                  SALARY
========================= ======= ======================
Consumer Electronics Div.     107              111262.50
Corporate Headquarters        105              212850.00
Customer Services              94               54000.00
Engineering                     2               98000.00
Field Office: Canada           72               96800.00
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
Sales and Marketing            85              111262.50
```

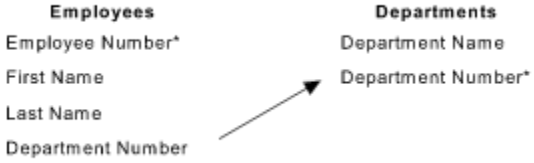


Outer joins

Outer joins produce a result 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 <searchcondition>]
   [WHERE <searchcondition>];
```

However, with outer joins, you need to specify the type of join to perform. There are three possibilities:



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 to the background of all data from which it is retrieved. For example, when listing the employees that are assigned to projects, it may be interesting to see the employees that are not assigned to projects, too.

The following outer join retrieves employee names from the EMPLOYEE table and project IDs from the

EMPLOYEE_PROJECT table, for employees that are assigned to projects.

```
SELECT PROJ_ID, FULL_NAME
   FROM EMPLOYEE LEFT OUTER JOIN EMPLOYEE_PROJECT
   ON EMPLOYEE.EMP_NO = EMPLOYEE_PROJECT.EMP_NO;
```

All employee names in the EMPLOYEE table are retrieved, regardless of whether they are assigned to a project, because EMPLOYEE is the left table in the join. Enter it to see what the results look like.
Notice that some employees are not assigned to a project; the PROJ_ID column is empty for them.
Reverse the outer join, by changing the FROM clause to:

```
   FROM EMPLOYEE_PROJECT LEFT OUTER JOIN EMPLOYEE
```

The results look different. Why?

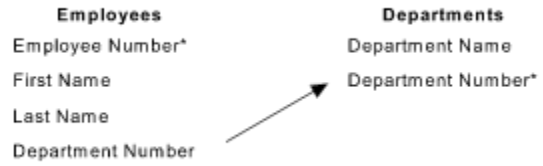
Formatting data

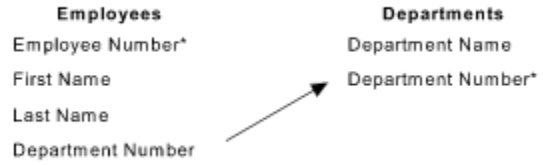
This section describes three ways to change data formats:

Converting data types

Concatenating strings

Converting characters to uppercase





Using CAST( ) to convert data types

Normally, only similar data types can be compared in search conditions, but you can work around this by using CAST(). Use the CAST function in search conditions to translate one data type into another. The syntax for CAST() is:

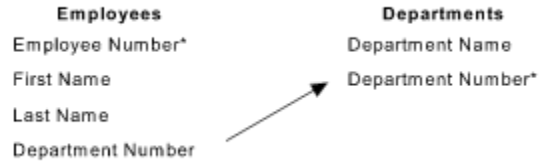
```
CAST (<value> | NULL AS datatype)
```

For example, the following WHERE clause uses CAST() to translate a CHAR data type, INTERVIEW_DATE, to a DATE data type. 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 Data Type	To Data Type
NUMERIC	CHARACTER, DATE
CHARACTER	NUMERIC, DATE
DATE	CHARACTER, NUMERIC



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. Character strings can be constants or values retrieved from a column. For example, enter the following:

```
SELECT DEPARTMENT, LAST_NAME || " is the manager"
   FROM DEPARTMENT, EMPLOYEE
   WHERE MNGR_NO = EMP_NO;
```

to produce this result:

```
DEPARTMENT
```

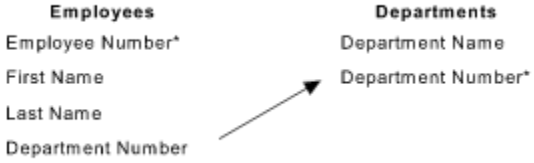
Corporate Headquarters	Bender is the manager
Sales and Marketing	MacDonald is the manager
Engineering	Nelson is the manager
Finance	Steadman is the manager
Quality Assurance	Forest is the manager
Customer Support	Young is the manager
Consumer Electronics Div.	Cook is the manager
Research and Development	Papadopoulos is the manager
Customer Services	Williams is the manager
Field Office: East Coast	Weston is the manager

. . .

Converting to uppercase

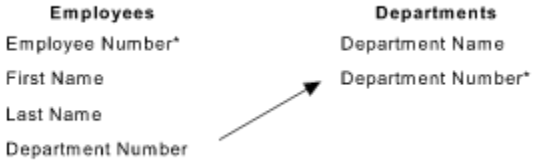
The UPPER() function converts character values to uppercase. For example, when defining a column in a table, you can use a CHECK constraint that ensures that all column values are entered in uppercase. The following CREATE DOMAIN statement uses the UPPER() function in defining the PROJNO domain:

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



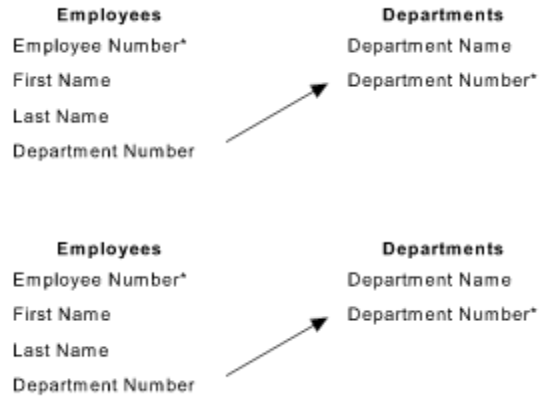
PART FIVE:
Advanced data definition

This final series of lessons introduces some advanced DDL features, including:



Creating and using triggers.

Creating and using stored procedures.



Triggers and stored procedures

A trigger is a self-contained routine associated with a table. Triggers automatically perform an action when a row in a table is inserted, updated, or deleted.

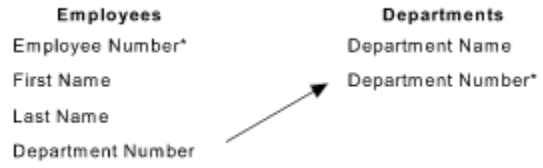
A stored procedure is a program that can be called by applications or from WISQL.

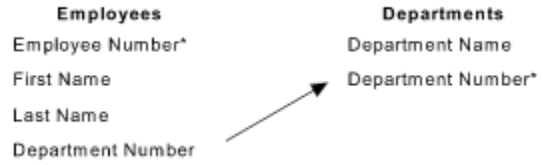
Both stored procedures and triggers are part of a database's metadata and are written in procedure and trigger language, an InterBase extension of SQL.

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.

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 never called directly. Instead, when an application or user attempts to INSERT, UPDATE, or DELETE a row in a table, any triggers associated with that table and operation are automatically executed, or fired.



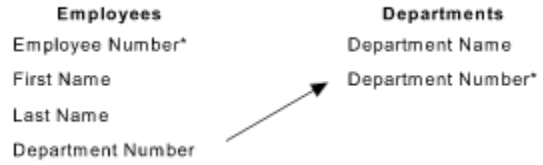


Triggers

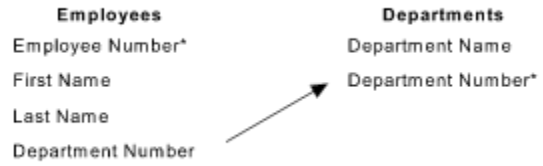
Triggers have a wide variety of uses, but in general, they enable you to automate tasks that would otherwise be done manually. They enable you to define actions that occur automatically whenever data is inserted, updated or deleted in a particular table. Triggers are a versatile tool, and their uses are virtually unlimited.

The triggers defined in the EMPLOYEE database:

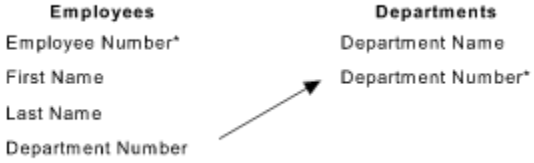
Generate and insert unique employee numbers in the EMPLOYEE table and customer numbers in the CUSTOMER table.



Maintain a record of employees' salary changes.



Post an event when a new sale is made.



Generating unique column values with triggers

Recall the EMPLOYEE table in the example database. This table has a primary key column named EMP_NO for each employee's employee number. Because it is a primary key, each employee number must be unique. And, generally, employee numbers are sequential. So, each time you insert a new employee record in this table, you would have to remember what the last employee number issued was, and then give the new employee the next number. This would be cumbersome and error-prone.

Triggers provide a simple 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. Each time GEN_ID() is called, it generates the next incremental value of the generator. The value of the generator is initialized with SET GENERATOR.

Look at the SQL file, TRIGGERS.SQL. The beginning of the file has the following statements:

```
CREATE GENERATOR EMP_NO_GEN;
SET GENERATOR EMP_NO_GEN TO 145;
```

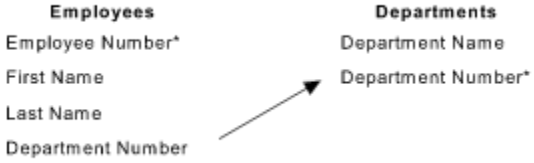
The first statement creates a generator named EMP_NO_GEN. The second statement initializes the generator to 145 (recall that in the script file, INSERTS.SQL, records were inserted into EMPLOYEE for employee numbers up to 145).

The next statements define a trigger named SET_EMP_NO that uses EMP_NO_GEN to generate unique sequential employee numbers, and inserts them into the EMPLOYEE table.

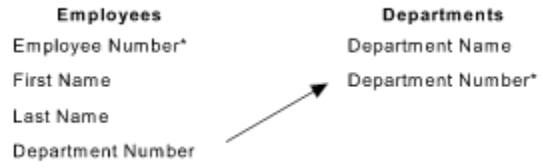
```
/* Create trigger to add unique customer number */
SET TERM !! ;
CREATE TRIGGER SET_EMP_NO FOR EMPLOYEE
BEFORE INSERT
AS
BEGIN
   NEW.EMP_NO = GEN_ID(EMP_NO_GEN, 1);
END !!
SET TERM ; !!
```

The statements above define the trigger. Because each statement in a trigger body must be terminated by a semicolon, SET TERM is first used to define a different symbol to terminate the CREATE TRIGGER statement as a whole.

The CREATE TRIGGER statement above specifies:



The name of the trigger, SET_EMP_NO



The table with which the trigger is associated, EMPLOYEE

When and how the trigger is fired, in this case before every INSERT operation

Following the AS keyword, the body of the trigger-what the trigger does when it fires, bracketed by BEGIN and END. In this case, it uses a context variable, NEW.EMP_NO to insert the next employee number into the EMP_NO column.
Context variables are unique to triggers. They allow you to specify NEW and OLD to reference the values of columns being updated.
There are several other triggers defined in TRIGGERS.SQL, which you will examine later. But first, you are going to see how the SET_EMP_NO trigger works. Read the file into ISQL by choosing File | Run an ISQL Script.... Now, refresh your memory of the EMPLOYEE table by typing the statement:

```
SELECT * from EMPLOYEE;
```

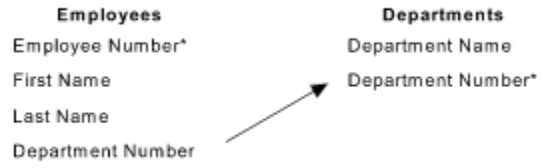
Notice that the last employee listed has employee number 145. Now enter a new employee record, for instance:

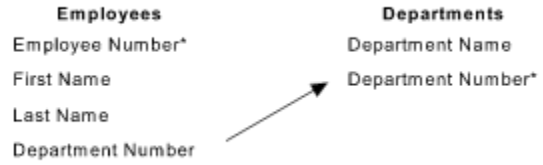
```
INSERT INTO EMPLOYEE (FIRST_NAME, LAST_NAME, DEPT_NO, JOB_CODE, JOB_GRADE,
JOB_COUNTRY, HIRE_DATE, SALARY, PHONE_EXT) VALUES ("Reed", "Richards",
"671", "Eng", 5, "USA", "07/27/95", 34000, "444");
```

Retrieve the new record by entering

```
SELECT * from EMPLOYEE WHERE LAST_NAME = "Richards";
```

Notice that the employee number is 146. The trigger has automatically assigned the new employee the next employee number.
TRIGGERS.SQL defines a similar trigger named SET_CUST_NO to assign unique customer numbers. It also defines two other triggers--SAVE_SALARY_CHANGE and POST_NEW_ORDER.





Maintaining change records with a trigger

SAVE_SALARY_CHANGE maintains a record of changes to employees' salaries in the SALARY_HISTORY table. Choose View | Metadata Information..., select Trigger, and then type "SAVE_SALARY_CHANGE" to view the trigger. This will be displayed in the Output area:

```
SHOW TRIGGER SAVE_SALARY_CHANGE
Triggers on Table EMPLOYEE:
SAVE_SALARY_CHANGE, Sequence: 0, Type: AFTER UPDATE, Active
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 then compares the value of the SALARY column before the update to SALARY after the update, and if they are different, it enters a record in SALARY_HISTORY consisting of the employee number, date, previous salary, and percentage change in the salary. Update an employee record and change the salary to see how this trigger works.

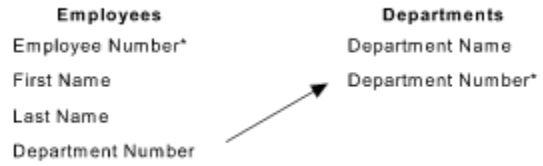
Posting an event with a trigger

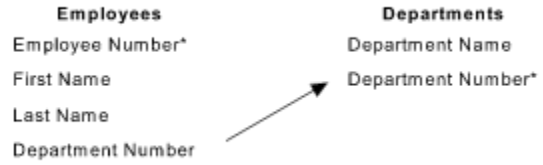
The trigger, POST_NEW_ORDER, posts an event named "new_order" whenever a record is inserted into the SALES table.

```
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 which have registered interest in an event can pause execution and wait for the specified event to occur.

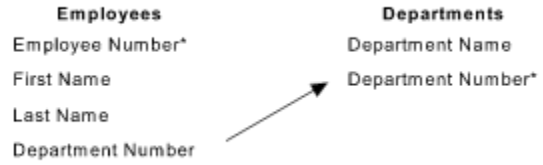
The POST_NEW_ORDER trigger is fired after a new record is inserted into the SALES table, in other words when a new sale is made. When this event occurs, interested applications may take appropriate action, such as printing an invoice or notifying the shipping department.



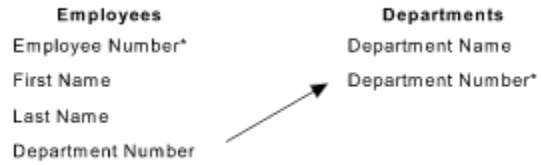


Stored procedures

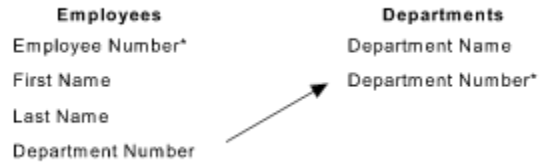
Stored procedures are programs stored with a database's metadata. Applications can call stored procedures and you can also use stored procedures in ISQL.
There are two types of stored procedures:



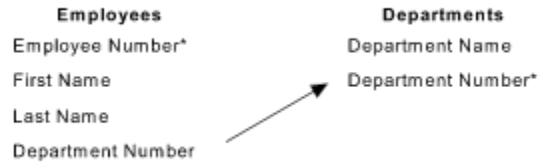
Select procedures that an application can use in place of a table or view in a SELECT statement. A select procedure must be defined to return one or more values (output parameters), or an error results.



Executable procedures that an application can call directly with the EXECUTE PROCEDURE statement. An executable procedure may or may 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. Select procedures can return more than one row, so that to the calling program they appear as a table or view. Executable procedures are simply routines invoked by the calling program which may or may not return values.
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.

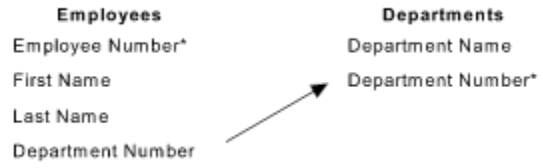


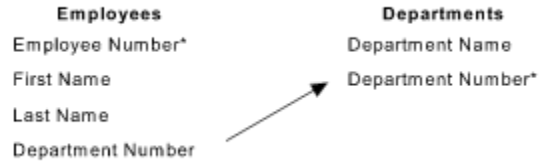
An optional list of input parameters and their data types that a procedure receives from the calling program.

If the procedure returns values to the calling program, the RETURNS keyword followed by a list of output parameters and their data types.
The procedure body contains:

An optional list of local variables and their data types.

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 may be many levels of nesting.
The stored procedures for the EMPLOYEE database are defined in the script file named PROCS.SQL. Open up this file with a text editor to view them. You are going to experiment with these procedures one at a time to learn about them.
Before that, though, you'll need to run PROCS.SQL. First, edit the file using Notepad (or any other text editor) and change the database name, user name, and password to match the current database. Be sure to save the changes. You can keep the file open in Notepad to review the procedures as you move through the rest of this lesson.
To execute the statements in this file, choose File | Run ISQL Script....
A standard file locator dialog appears. Use the dialog to locate the file PROCS.SQL (it should be in your local /Interbas/Examples/Tutorial directory). When you locate it, click Open.
Before running the script, Windows ISQL asks if you want to save the results to a file. Click No (you want to see the results in the ISQL Output window instead).
As Windows ISQL reads the script file, it echos the statements to the ISQL Output area.





A simple select procedure

The first procedure defined in PROCS.SQL is named GET_EMP_PROJ:

```
CREATE PROCEDURE GET_EMP_PROJ (EMP_NO SMALLINT)
RETURNS (PROJ_ID CHAR(5))
AS
BEGIN
   FOR SELECT PROJ_ID
   FROM EMPLOYEE_PROJECT
   WHERE EMP_NO = :EMP_NO
   INTO :PROJ_ID
   DO
      SUSPEND;
END ^
```

This is a select procedure that takes an employee number as its input parameter (EMP_NO, specified in parentheses after the procedure name) and returns all the projects to which the employee is assigned (PROJ_ID, specified after RETURNS).

It uses a FOR SELECT . . . DO statement to retrieve multiple rows from the EMPLOYEE_PROJECT table. This statement retrieves values just like a normal select statement, 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, ISQL).

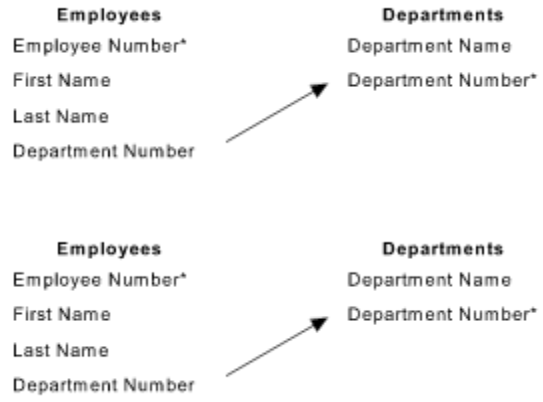
See how it works by entering the following query:

```
SELECT * FROM GET_EMP_PROJ(71);
```

As you can see, this query looks as if there is a table named GET_EMP_PROJ, except that you provide the input parameter in parentheses following the procedure name. The results are:

```
PROJ_ID
=======
VBASE
MAPDB
```

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



A simple executable procedure

The next procedure defined in PROCS.SQL is an executable procedure named ADD_EMP_PROJ. It is a simple example of an executable procedure and makes use of an exception, a named error message, defined with CREATE 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 statement EXCEPTION UNKNOWN_EMP_ID. The associated error message is then returned to the calling application.

The stored procedure, ADD_EMP_PROJ, is shown below:

```
CREATE PROCEDURE ADD_EMP_PROJ (EMP_NO SMALLINT, PROJ_ID CHAR(5))
AS
BEGIN
   BEGIN
      INSERT INTO EMPLOYEE_PROJECT (EMP_NO, PROJ_ID)
      VALUES (:emp_no, :proj_id);
      WHEN SQLCODE -530 DO
         EXCEPTION UNKNOWN_EMP_ID;
   END
   SUSPEND;
END ^
```

This procedure takes an employee number and project ID as input parameters and adds the employee to the specified project using a simple 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.

Use this procedure through the EXECUTE PROCEDURE statement, for example:

```
EXECUTE PROCEDURE ADD_EMP_PROJ(20, "DGPII");
```

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 is displayed on the screen.

A recursive procedure

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. Look at the stored procedure, DEPT_BUDGET:

```
SHOW PROCEDURE DEPT_BUDGET;
Procedure text:
============================================================
DECLARE VARIABLE sumb DECIMAL(12, 2);
DECLARE VARIABLE rdno CHAR(3);
DECLARE VARIABLE cnt INTEGER;
BEGIN
   tot = 0;

SELECT BUDGET FROM DEPARTMENT WHERE DEPT_NO = :dno INTO :tot;

SELECT COUNT(BUDGET) FROM DEPARTMENT WHERE HEAD_DEPT = :dno INTO :cnt;

IF (cnt = 0) THEN
SUSPEND;

FOR SELECT DEPT_NO
FROM DEPARTMENT
WHERE HEAD_DEPT = :dno
INTO :rdno
DO
BEGIN
EXECUTE PROCEDURE DEPT_BUDGET :rdno RETURNING_VALUES :sumb;
   tot = tot + sumb;
END

SUSPEND;
END

============================================================
Parameters:
DNO INPUT CHAR(3)
TOT OUTPUT NUMERIC(15, 2)
```

This procedure takes as its input parameter a department number and returns the budget of the department and all departments under it in the corporate hierarchy. It uses local variables declared with DECLARE VARIABLE statements. These variables are only used within the context of the procedure. First, the procedure retrieves from the DEPARTMENT table the budget of the department given as the input parameter and stores it in the variable, tot. Then it retrieves the number of departments reporting to that department using the COUNT() aggregate function. If there are no reporting departments, then it returns the value of tot 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 parentheses, 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 tot, 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. Try it:

```
EXECUTE PROCEDURE DEPT_BUDGET(620);
```

The result is:

```
                     TOT
======================
             2550000.00
```

Notice that the 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 automatic type conversion, a handy feature that will convert data types, where possible to the required data type. So the integer 620 is automatically converted to the character string "620".

The automatic type conversion will not work for department number 000 because it will convert it to the string "0", which is not a department number. Execute the procedure again with:

```
EXECUTE PROCEDURE DEPT_BUDGET("000");
```

This should give the same answer as the query:

```
SELECT SUM(BUDGET) FROM DEPARTMENT;
```

Can you figure out why?

There are a number of other procedures, some quite complex, defined in PROCS.SQL for the EMPLOYEE database. Now that you have a basic understanding of procedures, see if you can understand and use them.