



Bonus Reference

Transact-SQL

IN CHAPTER 20, YOU BUILT SQL statements to retrieve and update rows in a database. You also learned all the variations of the SELECT statement. Some restrictions, however, can't be expressed with a WHERE clause, no matter how complicated. To perform complicated searches, many programmers would rather write an application to retrieve the desired rows, process them, and then display some results. The processing could include running totals, formatting of the query's output, calculations involving multiple rows of the same table, and so on.

Most database management systems include extensions, which enhance SQL and make it behave more like a programming language. SQL Server provides a set of statements known as *Transact-SQL (T-SQL)*. T-SQL recognizes statements that fetch rows from one or more tables, flow-control statements like IF...ELSE and WHILE, and numerous functions to manipulate strings, numeric values, and dates, similar to Visual Basic's functions. With T-SQL you can do everything that can be done with SQL, as well as program these operations. At the very least, you can attach lengthy SQL statements to a database as stored procedures, so that you can call them by name. In addition, you can use parameters, so that the same procedure can act on different data.

This chapter is an overview of T-SQL and demonstrates the basic topic with stored procedures that perform complicated tasks on the server. We'll start with the COMPUTE BY statement, which allows you to calculate totals on groups of the rows retrieved from the database. This statement looks and feels very much like the straight SQL statements, yet it's not part of standard SQL.

Then we'll look at T-SQL in depth. If you need to look up a function or how to declare a stored procedure's arguments, you can use this chapter as reference material. If you're new to T-SQL, you should read this material, because T-SQL is a powerful language, and if you're working with or plan to switch to SQL Server, you'll need it sooner or later. In addition, you'll improve your Visual Basic programming. T-SQL is the native language of SQL Server. By seeing how the basic operations can be implemented in T-SQL and VB, you'll gain a deeper understanding of database programming.

The COMPUTE BY Clause

As you recall from Chapter 20, SQL SELECT statements return a set of rows, all of which have the same structure. In other words, a SQL query can return either details or totals, but not both. For example, you can calculate the order totals for all customers with a GROUP BY clause, but this clause displays totals only. Let’s say you want a list of all customers in the Northwind database, their orders, and the total for each customer. Listing 1 provides a SQL SELECT statement that retrieves the desired information from the database.

LISTING 1: THE ORDERTOTALS.SQL QUERY

```
USE NORTHWIND
SELECT CompanyName, Orders.OrderID,
       SUM([Order Details].UnitPrice * Quantity * (1 - Discount))
FROM Products, [Order Details], Customers, Orders
WHERE [Order Details].ProductID = Products.ProductID AND
      [Order Details].OrderID = Orders.OrderID AND
      Orders.CustomerID=Customers.CustomerID
GROUP BY CompanyName, Orders.OrderID
ORDER BY CompanyName, Orders.OrderID
```

This statement calculates the totals for each order. The Orders.OrderID field is included in the GROUP BY clause because it’s part of the SELECT list, but doesn’t appear in an aggregate function’s arguments. This statement will display groups of customers and the totals for all orders placed by each customer:

Alfreds Futterkiste	10643	814.5
Alfreds Futterkiste	10692	878.0
Alfreds Futterkiste	10702	330.0
Alfreds Futterkiste	10835	845.80
Alfreds Futterkiste	10952	471.20
Alfreds Futterkiste	11011	933.5
Ana Trujillo Emparedados y helados	10308	88.80
Ana Trujillo Emparedados y helados	10625	479.75
. . .		

If you want to see the totals per customer, you must modify Listing 1 as follows:

```
USE NORTHWIND
SELECT CompanyName,
       SUM([Order Details].UnitPrice * Quantity * (1 - Discount))
FROM Products, [Order Details], Customers, Orders
WHERE [Order Details].ProductID = Products.ProductID AND
      [Order Details].OrderID = Orders.OrderID AND
      Orders.CustomerID=Customers.CustomerID
GROUP BY CompanyName
ORDER BY CompanyName
```

This time I've omitted the `Orders.OrderID` field from the `SELECT` list and the `GROUP BY` clause. This statement will display the total for each customer, since we are not grouping by `OrderID`:

```

Alfreds Futterkiste          4272.9999980926514
Ana Trujillo Emparedados y helados 1402.9500007629395
Antonio Moreno Taquería      7023.9775543212891
Around the Horn              13390.650009155273
Berglunds snabbköp          24927.57746887207
. . .

```

What we need is a statement that can produce a report of the details with total breaks after each order and each customer, as shown here (I have shortened the product names to fit the lines on the printed page without breaks):

```

Alfreds Futterkiste    10643  Rössle      45.60  15  25  513.00
Alfreds Futterkiste    10643  Chartreuse  18.0   21  25  283.50
Alfreds Futterkiste    10643  Spegesild   12.0    2  25   18.00
                                     814.50
Alfreds Futterkiste    10692  Vegie-spread 43.90  20   0  878.00
                                     878.00
Alfreds Futterkiste    10702  Aniseed Syrup 10.0    6   0   60.00
Alfreds Futterkiste    10702  Lakkaalikööri 18.0   15   0  270.00
                                     845.80
Alfreds Futterkiste    10952  Grandma's    25.00  16   5  380.00
Alfreds Futterkiste    10952  Rössle      45.6    2   0   91.20
                                     471.20
Alfreds Futterkiste    11011  Escargots    13.25  40   5  503.50
Alfreds Futterkiste    11011  Flotemysost  21.50  20   0  430.00
                                     933.50
                                     4273.00
Ana Trujillo           10308  Gudbrand    28.80   1   0   28.80
Ana Trujillo           10308  Outback     12.00   5   0   60.00
                                     88.80
Ana Trujillo           10625  Tofu        23.25   3   0   69.75
Ana Trujillo           10625  Singaporean 14.00   5   0   70.00
Ana Trujillo           10625  Camembert   34.00  10   0  340.00
                                     479.75

```

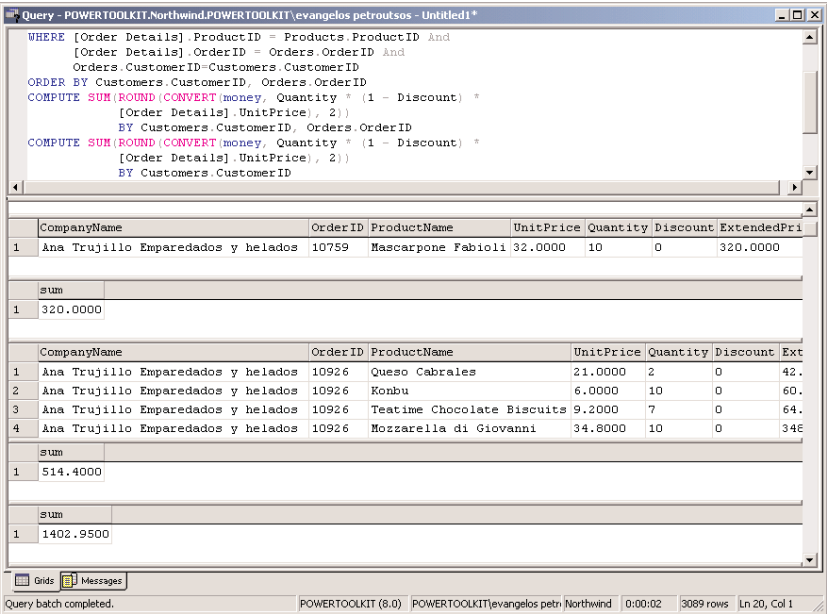
T-SQL provides an elegant solution to this problem with the `COMPUTE BY` clause. The `COMPUTE BY` clause calculates aggregate functions (sums, counts, and so on) while a field doesn't change value. This field is specified with the `BY` keyword. When the field changes value, the total calculated so far is displayed and the aggregate function is reset. To produce the list shown here, you must calculate the sum of line totals (quantity × price – discount) and group the calculations according to `OrderID` and `CustomerID`. Listing 2 presents the complete statement that produced the preceding subtotaled list.

LISTING 2: THE ORDERSGROUPED.SQL QUERY

```
USE NORTHWIND
SELECT CompanyName, Orders.OrderID, ProductName,
       UnitPrice=ROUND([Order Details].UnitPrice, 2),
       Quantity,
       Discount=CONVERT(int, Discount * 100),
       ExtendedPrice=ROUND(CONVERT(money,
       Quantity * (1 - Discount) * [Order Details].UnitPrice), 2)
FROM Products, [Order Details], Customers, Orders
WHERE [Order Details].ProductID = Products.ProductID And
      [Order Details].OrderID = Orders.OrderID And
      Orders.CustomerID = Customers.CustomerID
ORDER BY Customers.CustomerID, Orders.OrderID
COMPUTE SUM(ROUND(CONVERT(money, Quantity * (1 - Discount) *
      [Order Details].UnitPrice), 2))
      BY Customers.CustomerID, Orders.OrderID
COMPUTE SUM(ROUND(CONVERT(money, Quantity * (1 - Discount) *
      [Order Details].UnitPrice), 2))
      BY Customers.CustomerID
```

The first COMPUTE BY clause groups the invoice line totals by order ID within each customer. The second COMPUTE BY clause groups the same totals by customer, as shown in Figure 1. The CONVERT() function converts data types similar to the Format() function of VB, and the ROUND() function rounds a floating-point number. Both functions are discussed later in this chapter.

FIGURE 1
Using the COMPUTE BY clause to calculate totals on groups



The COMPUTE BY clause can be used with any of the aggregate functions you have seen so far. Listing 3 displays the order IDs by customer and calculates the total number of invoices issued to each customer:

LISTING 3: THE COUNTINVOICES.SQL QUERY

```
USE NORTHWIND
SELECT Customers.CompanyName, Orders.OrderID
FROM Customers, Orders
WHERE Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerID
COMPUTE COUNT(Orders.OrderID) BY Customers.CustomerID
```

The SQL engine will count the number of orders while the CustomerID field doesn't change. When it runs into a new customer, the current total is displayed and the counter is reset to zero in anticipation of the next customer. Here's the output produced by Listing 3:

CompanyName	OrderID
-----	-----
Alfreds Futterkiste	10643
Alfreds Futterkiste	10692
Alfreds Futterkiste	10702
Alfreds Futterkiste	10835
Alfreds Futterkiste	10952
Alfreds Futterkiste	11011
	=====
	6
Ana Trujillo Emparedados y helados	10308
Ana Trujillo Emparedados y helados	10625
Ana Trujillo Emparedados y helados	10759
Ana Trujillo Emparedados y helados	10926
	=====
	4

In addition to combining multiple COMPUTE BY clauses in the same statement (as we did in Listing 2), you can add another COMPUTE statement without the BY clause to display a grand total:

```
USE NORTHWIND
SELECT Customers.CompanyName, Orders.OrderID
FROM Customers, Orders
WHERE Customers.CustomerID=Orders.CustomerID
ORDER BY Customers.CustomerID
COMPUTE COUNT(Orders.OrderID) BY Customers.CustomerID
COMPUTE COUNT(Orders.OrderID)
```

The COMPUTE BY clause requires that the rows are furnished in the proper order, so all the fields following the BY keyword must also appear in an ORDER BY clause. The COMPUTE BY

clause will not change the order of the rows to facilitate its calculations. Actually, the SQL engine will refuse to execute a statement that contains a COMPUTE BY clause but not the equivalent ORDER clause; it will abort the statement's execution and display the following error message:

```
A COMPUTE BY item was not found in the order by list.
All expressions in the compute by list must also be
present in the order by list.
```

Stored Procedures

A *stored procedure* is a routine written in T-SQL that acts on the rows of one or more tables. All SQL statements you have seen so far act on selected rows (they select, update, or delete rows), but SQL doesn't provide the means to alter the course of action depending on the values of the fields. There's no support for IF statements, no functions to manipulate strings, no formatting functions, and so on. Every DBMS manufacturer extends standard SQL with statements that add the functionality of a programming language. Access queries, for example, recognize the Mid() function, which is identical to the VB function by the same name. It extracts part of a string field and uses it as another field. The equivalent T-SQL function is called SUBSTRING(). In the rest of this chapter, we'll look at the statements and functions of T-SQL.

Stored procedures are attached to SQL Server databases and become objects of the database, like tables and views. The simplest application of stored procedures is to attach complicated queries to the database and call them by name, so that users won't have to enter them more than once. As you will see, stored procedures have many more applications, and they can even be used to build business rules into the database (but more on this later in this chapter).

A stored procedure performs the same calculations as your VB application, only it's executed on the server and uses T-SQL statements. T-SQL is practically a programming language. It doesn't have a user interface, so you can't use it to develop full-blown applications, but when it comes to querying or updating the database and data processing, it can do everything a VB application can do.

You may wonder now, why bother with stored procedures if they don't do more than VB? The answer is that T-SQL is SQL Server's native language, and stored procedures are executed on the server. A stored procedure can scan thousands of records, perform calculations, and return a single number to a VB application. If you perform calculations that involve a large number of rows, you can avoid downloading too much information to the client by writing a stored procedure to do the work on the server instead. Stored procedures are executed faster than the equivalent VB code because they're compiled and they don't move data from the server to the client.

Another good reason for using stored procedures is that once they're defined, they become part of the database and appear to applications as database objects like tables and views. Consider a stored procedure that adds new orders to a database. This stored procedure is part of the database, and you can set up the database so that users and applications can't modify the Orders table directly. By forcing them to go through a stored procedure, you can be sure that all orders are recorded properly. If you provide a procedure for editing the Orders table, no one can tamper with the integrity of your data. Moreover, if you change the structure of the underlying tables, you can modify the stored procedure, and all VB applications that use the stored procedure will continue as before. You can also implement business rules in the stored procedure (decrease the stock, update a list of best-sellers, and

so on). By incorporating all this functionality in to the stored procedure, you simplify the coding of the client application.

Creating and Executing Stored Procedures

To write, debug, and execute stored procedures against a SQL Server database, you must use the Query Analyzer. You can also right-click the Stored Procedures item under a database in the Server Explorer and select New Stored Procedure, as explained in Chapter 20. In this chapter, I will use the Query Analyzer.

To create a new stored procedure, enter the definition of the procedure in the Query pane and then press Ctrl+E to execute that definition. This action will attach the procedure to the database, but it will not actually execute it. To execute a procedure that's already been stored to the database, you must use the EXECUTE statement, which is discussed shortly.

To create a new stored procedure and attach it to the current database, use the CREATE PROCEDURE statement. The syntax of the statement is:

```
CREATE PROCEDURE procedure_name
AS
{ procedure definition }
```

where *procedure_name* is the name of the new stored procedure and the statement block following the AS keyword is the body of the procedure. In its simplest form, a stored procedure is a SQL statement, like the ones we have discussed so far. If you think you'll be frequently executing the AllInvoices query (shown in Listing 4), you can create a stored procedure containing the SQL statement that retrieves customers, orders, and order details. Every time you need this report, you can call this procedure by name. To create the AllInvoices stored procedure, enter the lines from Listing 4 in the Query pane of the Query Analyzer.

LISTING 4: THE ALLINVOICES QUERY STORED PROCEDURE

```
USE NORTHWIND
IF EXISTS (SELECT name FROM sysobjects
           WHERE name = 'AllInvoices')
    DROP PROCEDURE AllInvoices
GO
CREATE PROCEDURE AllInvoices
AS
SELECT CompanyName, Orders.OrderID, ProductName,
       UnitPrice=ROUND([Order Details].UnitPrice, 2),
       Quantity,
       Discount=CONVERT(int, Discount * 100),
       ExtendedPrice=ROUND(CONVERT(money, Quantity * (1 - Discount) *
                                   [Order Details].UnitPrice), 2)
FROM Products, [Order Details], Customers, Orders
WHERE [Order Details].ProductID = Products.ProductID And
      [Order Details].OrderID = Orders.OrderID And
      Orders.CustomerID=Customers.CustomerID
ORDER BY Customers.CustomerID, Orders.OrderID
```

```

COMPUTE SUM(ROUND(CONVERT(money, Quantity * (1 - Discount) *
                        [Order Details].UnitPrice), 2))
        BY Customers.CustomerID, Orders.OrderID
COMPUTE SUM(ROUND(CONVERT(money, Quantity * (1 - Discount) *
                        [Order Details].UnitPrice), 2))
        BY Customers.CustomerID

```

Because this is not actually a SQL statement, the first time you execute it, it will not return the list of invoices. Instead, it will add the *AllInvoices* procedure to the current database—so be sure to select the Northwind database in the DB drop-down list, or use the *USE* keyword to make Northwind the active database.

If the procedure exists already, you can't create it again. You must either drop it from the database with the *DROP PROCEDURE* statement, or modify it with the *ALTER PROCEDURE* statement. The syntax of the *ALTER PROCEDURE* statement is identical to that of the *CREATE PROCEDURE* statement. By replacing the *CREATE* keyword with the *ALTER* keyword, you can replace the definition of an existing procedure.

A common approach is to test for the existence of a stored procedure and drop it if it exists. Then, you can add a new procedure with the *CREATE PROCEDURE* statement. For example, if you are not sure the *myProcedure* procedure exists, use the following statements to find and modify it:

```

USE DataBase
IF EXISTS (SELECT name FROM sysobjects
           WHERE name = 'myProcedure')
    DROP PROCEDURE myProcedure
GO
CREATE PROCEDURE myProcedure
AS
. . .

```

The *SELECT* statement retrieves the name of the desired procedure from the database objects (again, be sure to execute it against the desired database). If a procedure by the name *myProcedure* exists already, *EXISTS* returns *True* and drops the procedure definition from the database. Then it proceeds to add the revised definition.

Once you've entered Listing 4 into the Query Analyzer, press *Ctrl+E* to execute the procedure's declaration. If you haven't misspelled any keywords, the message "The command(s) completed successfully." will appear in the lower pane of the Query Analyzer's window. When you execute a stored procedure's definition, you add it to the database, but the procedure's statements are not executed.

To execute a stored procedure, you must use the *EXECUTE* statement (or its abbreviation, *EXEC*) followed by the name of the procedure. Assuming that you have created the *AllInvoices* procedure, here's how to execute it.

1. First, clear the Query pane of Query Analyzer, or open a new window in the Query Analyzer.
2. In the fresh Query pane, type:


```

USE Northwind
EXECUTE AllInvoices

```

and press *Ctrl+E*. The result of the query will appear in the Results pane of the Query Analyzer.

The first time you execute the procedure, SQL Server will put together an execution plan, so it will take a few seconds. After that, the procedure's execution will start immediately, and the rows will start appearing on the Results pane as soon as they become available.

TIP *If a procedure takes too long to execute, or it returns too many rows, you can interrupt it by clicking the Stop button (a red rectangular button on SQL Server's toolbar). If you execute an unconditional join by mistake, for example, you can stop the execution of the query and not have to wait until all rows arrive.*

The USE statement isn't necessary, as long as you remember to select the proper database in the Analyzer's window. Since the stored procedure is part of the database, it's very unlikely that you will call a stored procedure that doesn't belong to the current database.

Executing Command Strings

In addition to executing stored procedures, you can use the EXECUTE statement to execute strings with valid T-SQL statements. If the variable *@TSQLcmd* contains a valid SQL statement, you can execute it by passing it as an argument to the EXECUTE procedure:

```
EXECUTE (@TSQLcmd)
```

The parentheses are required. If you omit them, SQL Server will attempt to locate the *@TSQLcmd* stored procedure. Here's a simple example of storing SQL statements into variables and executing them with the EXECUTE method:

```
DECLARE @Country varchar(20)
DECLARE @TSQLcmd varchar(100)
SET @Country = 'Germany'
SET @TSQLcmd = 'SELECT City FROM Customers WHERE Country="' + @Country + '"'
EXECUTE (@TSQLcmd)
```

TIP *All T-SQL variables must begin with the @ symbol.*

All T-SQL variables must be declared with the DECLARE statement, have a valid data type, and be set with the SET statement. You will find more information on the use of variables in the following sections of this chapter.

The EXECUTE statement with a command string is commonly used to build SQL statements on the fly. You'll see a more practical example of this technique in the section "Building SQL Statements on the Fly," later in this chapter.

TIP *Statements that are built dynamically and executed with the help of a command string as explained in this section do not take advantage of the execution plan. Therefore, you should not use this technique frequently. Use it only when you can't write the stored procedure at design time.*

Why Use Stored Procedures?

Stored procedures are far more than a programming convenience. When a SQL statement, especially a complicated one, is stored in the database, the database management system (DBMS) can execute it efficiently. To execute a SQL statement, the query engine must analyze it and put together an execution plan. The execution plan is analogous to the compilation of a traditional application. The

DBMS translates the statements in the procedure to statements it can execute directly against the database. When the SQL statement is stored in the database as a procedure, its execution plan is designed once and is ready to be used. Moreover, stored procedures can be designed once, tested, and used by many users and/or applications. If the same stored procedure is used by more than one user, the DBMS keeps only one copy of the procedure in memory, and all users share the same instance of the procedure. This means more efficient memory utilization. Finally, you can limit user access to the database's tables and force users to access the database through stored procedures. This is a simple method of enforcing business rules.

Let's say you have designed a database like Northwind, and you want to update each product's stock, and perhaps customer balances, every time a new invoice is issued. You could write the applications yourself and hope you won't leave out any of these operations, then explain the structure of the database to the programmers and hope they'll follow your instructions. Or you could implement a stored procedure that accepts the customer's ID and the IDs and quantities of the items ordered, then updates all the tables involved in the transaction. Application programmers can call this stored procedure and never have to worry about remembering to update some of the tables. At a later point, you may add a table to the database for storing the best-selling products. You can change the stored procedure, and the client applications that record new orders through this stored procedure need not be changed. Later in this chapter, you'll see examples of stored procedures that implement business rules.

T-SQL: The Language

The basic elements of T-SQL are the same as those of any other programming language: variables, flow-control statements, and functions. In the following few sections, we'll go quickly through the elements of T-SQL. Since this book is addressed to VB programmers, I will not waste any time explaining what variables are and why they must have a type. I will discuss T-SQL by comparing its elements to the equivalent VB elements and stress the differences in the statements and functions of the two languages.

T-SQL Variables

T-SQL is a typed language. Every variable must be declared before it is used. T-SQL supports two types of variables: local and global. Local variables are declared in the stored procedure's code, and their scope is limited to the procedure in which they were declared. The global variables are exposed by SQL Server, and you can use them without declaring them and from within any procedure.

LOCAL VARIABLES AND DATA TYPES

Local variables are declared with the DECLARE statement, and their names must begin with the @ character. The following are valid variable names: *@CustomerTotal*, *@Avg_Discount*, *@i*, *@counter*. To use them, declare them with the DECLARE statement, whose syntax is:

```
DECLARE var_name var_type
```

where *var_name* is the variable's name and *var_type* is the name of a data type supported by SQL Server. Unlike older versions of Visual Basic, T-SQL doesn't create variables on the fly. All T-SQL variables must be declared before they can be used. The available data types are listed here.

char, varchar

These variables store characters, and their length can't exceed 8,000 characters. The following statements declare and assign values to char variables:

```
DECLARE @string1 char(20)
DECLARE @string2 varchar(20)
SET @string1 = 'STRING1'
SET @string2 = 'STRING2'
PRINT '[' + @string1 + ']'
PRINT '[' + @string2 + ']'
```

The last two statements print the following:

```
[STRING1      ]
[STRING2]
```

The char variable is padded with spaces to the specified length, while the varchar variable is not. If the actual data exceeds the specified length of the string, both types truncate the string to its declared maximum length (to 20 characters, in this example).

nchar, nvarchar

The nchar and nvarchar types are equivalent to the char and varchar types, but they are used for storing Unicode strings.

bit, bigint, int, smallint, tinyint

Use these types to store whole numbers (integers). Table 1 details their storage and memory characteristics.

TABLE 1: T-SQL INTEGER DATA TYPES

T-SQL TYPE	EQUIVALENT VB TYPE	MEMORY	RANGE
bit		1 bit	0 or 1
tinyint	Byte	1 byte	0 to 255
smallint	Short	2 bytes	−32,768 to 32,767
int	Integer	4 bytes	−2,147,483,648 to 2,147,483,647
bigint	Long	8 bytes	−9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

decimal, numeric

These two types store an integer to the left of the decimal point and a fractional value to the right of the decimal point. The digits allocated to the integer and fractional part of the number are specified in the variable's declaration:

```
DECLARE @DecimalVar decimal(4, 3)
```

The first argument specifies the maximum total number of digits that can be stored to the left and right of the decimal point. Valid values are in the range 1 through 28. The second argument specifies the maximum number of digits that can be stored to the right of the decimal point (the integer part), and its value must be in the range 0 through the specified precision (in other words, the second argument can't be larger than the first argument).

If you store the value 30.25 / 4 to the *@DecimalVar* variable and then print it, it will be displayed as 7.563. If the variable was declared as:

```
DECLARE @DecimalVar decimal(12, 6)
```

the same value will be printed as 7.562500.

datetime, smalldatetime

Use these two types to store date and time values. The *datetime* type uses eight bytes: four bytes for the date and four bytes for the time. The time portion of the *datetime* type is the number of milliseconds after midnight and is updated every 3.33 milliseconds. The *smalldatetime* type uses four bytes: two bytes to store the number of days after January 1, 1900, and two bytes to store the number of minutes since midnight.

The following statements demonstrate how to add days, hours, and minutes to variables of the *datetime* and *smalldatetime* types:

```
DECLARE @DateVar datetime
DECLARE @smallDateVar smalldatetime
PRINT 'datetime type'
SET @DateVar = '1/1/2000 03:02.10'
PRINT @DateVar
/* Add a day to a datetime variable */
SET @DateVar = @DateVar + 1
PRINT @DateVar
/* Add an hour to a datetime variable */
SET @DateVar = @DateVar + 1.0/24.0
PRINT @DateVar
/* Add a minute to a datetime variable */
SET @DateVar = @DateVar + 1.0/(24.0 * 60.0)
PRINT @DateVar
PRINT 'smalldatetime type'
SET @smallDateVar = '1/1/2000 03:02.10'
PRINT @smallDateVar
/* Add a day to a smalldatetime variable */
SET @smallDateVar = @smallDateVar + 1
PRINT @smallDateVar
/* Add an hour to a smalldatetime variable */
SET @smallDateVar = @smallDateVar + 1.0/24.0
PRINT @smallDateVar
```

If you enter these lines in the Query pane of the Query Analyzer and execute them, the following output will be produced:

```
datetime type
Jan  1 2000  3:02AM
Jan  2 2000  3:02AM
Jan  2 2000  4:02AM
Jan  2 2000  4:03AM
smalldatetime type
Jan  1 2000  3:02AM
Jan  2 2000  3:02AM
Jan  2 2000  4:02AM
```

float, real

The float and real types are known as approximate data types and are used for storing floating-point numbers. These types are known as approximate numeric types, because they store an approximation of the exact value (which is usually the result of a calculation). The approximation is unavoidable because binary and decimal fractional values don't match exactly. The most obvious manifestation of this approximation is that you can't represent the value 0 with a float or real variable. If you perform complicated calculations with floating-point numbers and you want to know whether the result is zero, do not compare it directly to zero, as it may be a very small number but not exactly zero. Use a comparison like the following one:

```
SET @zValue = 1E-20
IF Abs(@result) < @zValue
    . . .
ELSE
    . . .
```

The LOG() function returns the logarithm of a numeric value. The following two statements return the same value:

```
SET @n1 = log(1 / 0.444009)
SET @n2 = -log(0.444009)
```

(The variables @n1 and @n2 must be declared as float or real.) If you print the two values with the PRINT statement, the values will be identical. However, the two values are stored differently internally, and if you subtract them, their difference will not be zero.

```
PRINT @n1 - @n2
```

The result will be the value $-4.41415\text{e}-008$.

money, smallmoney

Use these two types to store currency amounts. Both types use four fractional digits. The money type uses eight bytes and can represent amounts in the range $-922,337,203,685,477.5807$ to $922,337,203,685,377.5807$. The smallmoney type uses four bytes and can represent amounts in the range $-214,748.3648$ to $214,748.3647$.

text

The text data type can store non-Unicode data with a maximum length of 2,147,483,647 characters.

image

This data type can store binary data up to 2,147,483,647 bytes in size.

binary, varbinary

These variables store binary values. The two following statements declare two binary variables, one with fixed length and another with variable length:

```
DECLARE @bVar1 binary(4), @bVar2 varbinary
```

The binary data types can store up to 8,000 bytes. Use binary variables to store the values of binary columns (small images or other unusual bits of information). To assign a value to a binary variable, use the hexadecimal notation:

```
SET @bVar1 = 0x000000F0
```

This statement assigns the decimal value 255 to the *bVar1* variable.

timestamp

This data type holds a counter value that's unique to the database—an ever-increasing value that represents the current date and time. Timestamp columns are commonly used to find out whether a row has been changed since it was read. You can't extract date and time information from a timestamp value, but you can compare two timestamp values to find out whether a row was updated since it was read, or the order in which rows were read.

Each table can have only one timestamp column, and this column's values are set by SQL Server each time a row is inserted or updated. However, you can read this value and compare it to the other timestamp values.

uniqueidentifier

This is a globally unique identifier (GUID), which is usually assigned to a column with the NEWID() function. These values are extracted from network card identifiers or other machine-related information, and they are used in replication schemes to identify rows.

GLOBAL VARIABLES

In addition to the local variables you can declare in your procedures, T-SQL supports some global variables, whose names begin with the symbols @@. These values are maintained by the system, and you can read them to retrieve system information.

@@FETCH_STATUS

The @@FETCH_STATUS variable is zero if the FETCH statement successfully retrieved a row, nonzero otherwise. This variable is set after the execution of a FETCH statement and is commonly used to terminate a WHILE loop that scans a cursor. The global variable @@FETCH_STATUS

may also have the value -2 , which indicates that the row you attempted to retrieve has been deleted since the cursor was created. This value applies to keyset-driven cursors only.

@@CURSOR_ROWS, @@ROWCOUNT

The @@CURSOR_ROWS global variable returns the number of rows in the most recently opened cursor, and the @@ROWCOUNT variable returns the number of rows affected by the action query. The @@ROWCOUNT variable is commonly used with UPDATE and DELETE statements to find out how many rows were affected by the SQL statement. To find out how many rows were affected by an update statement, print the @@ROWCOUNT global variable after executing the SQL statement:

```
UPDATE Customers
SET PhoneNumber = '031' + PhoneNumber
WHERE Country = 'Germany'
PRINT @@ROWCOUNT
```

@@ERROR

The @@ERROR variable returns an error number for the last T-SQL statement that was executed. If this variable is zero, then the statement was executed successfully.

@@IDENTITY

The @@IDENTITY global variable returns the most recently used value for an Identity column. Identity columns can't be set; they are assigned a value automatically by the system, each time a new row is added to the table. Applications usually need this information because Identity fields are used as foreign keys into other tables. Let's say you're adding a new order to the Northwind database. First you must add a row to the Orders table. You can specify any field's value, but not the OrderID field's value. When the new row is added to the Orders table, you must add rows with the invoice details to the Order Details table. To do so, you need the value of the OrderID field, which can be retrieved by the @@IDENTITY global variable. In the section "Implementing Business Rules with Stored Procedures," later in this chapter, you'll find examples on how to use the @@IDENTITY variable.

Other Global Variables

Many global variables relate to administrative tasks, and they are listed in Table 2. T-SQL exposes more global variables, but the ones listed here are the most common.

TABLE 2: COMMONLY USED T-SQL GLOBAL VARIABLES

VARIABLE NAME	DESCRIPTION
@@CONNECTIONS	The number of login attempts since SQL Server was last started
@@CPU_BUSY	The number of ticks the CPU spent for SQL Server since it was last started
@@IDENTITY	The most recently created IDENTITY value

Continued on nextpage

TABLE 2: COMMONLY USED T-SQL GLOBAL VARIABLES (continued)

VARIABLE NAME	DESCRIPTION
@@IDLE	The number of ticks SQL Server has been idle since it was last started
@@IO_BUSY	The number of ticks SQL Server spent for input/output operations since it was last started
@@LANGID	The current language ID
@@LANGUAGE	The current language
@@LOCK_TIMEOUT	The current lock-out setting in milliseconds
@@MAX_CONNECTIONS	The maximum number of simultaneous connections that can be made to SQL Server
@@MAX_PRECISION	The current precision setting for decimal and numeric data types
@@NESTLEVEL	The number of nested transactions for the current execution (transactions can be nested up to 16 levels)
@@SERVERNAME	The name of the local SQL Server
@@TOTAL_ERRORS	The number of total errors since SQL was started for the last time
@@TOTAL_READ	The number of reads from the disk since SQL Server was last started
@@TOTAL_WRITE	The number of writes from the disk since SQL Server was last started
@@TRANCOUNT	The number of active transactions for the current user
@@SPID	The process ID of the current user process on the server (this number identifies a process, not a user)

You should consult SQL Server’s online documentation for more information on the global variables. These variables are used mainly by database administrators, not programmers.

Flow-Control Statements

T-SQL supports the basic flow-control statements that enable you to selectively execute blocks of statements based on the outcome of a comparison. They are similar to the equivalent VB statements and, even though there aren’t as many, they are adequate for the purposes of processing rows.

IF...ELSE

This statement executes a block of statements conditionally, and its syntax is:

```
IF condition
  { statement }
ELSE
  { statement }
```

Notice that there's no THEN keyword and that a T-SQL IF block is not delimited with an END IF keyword. To execute more than a single statement in the IF or ELSE clauses, you must use the BEGIN and END keywords to enclose the blocks of statements:

```
IF condition
  BEGIN
    { multiple statements }
  END
ELSE
  BEGIN
    { multiple statements }
  END
```

Depending on the *condition*, one of the two blocks of statements are executed. Here's an example of the IF...THEN statement with statement blocks:

```
IF (SELECT COUNT(*) FROM Customers WHERE Country = 'Germany') > 0
  BEGIN
    { statements to process German customers }
  END
ELSE
  BEGIN
    PRINT "The database contains no customers from Germany"
  END
```

Notice the second pair of BEGIN/END keywords are optional because the ELSE clause consists of a single statement.

CASE

The CASE statement is equivalent to Visual Basic's `Select Case` statement. SELECT is a reserved SQL keyword and shouldn't be used with the CASE statement, except as explained shortly. The CASE statement compares a variable (or field) value against several values and executes a block of statement, depending on which comparison returns a True result.

A car rental company may need to calculate insurance premiums based on a car's category. Instead of multiple IF statements, you can use a CASE structure like the following:

```
CASE @CarCategory
  WHEN 'COMPACT' THEN 25.5
  WHEN 'ECONOMY' THEN 37.5
  WHEN 'LUXURY' THEN 45.0
END
```

The CASE statement will return a single value: the one that corresponds to the first WHEN clause that's true. Notice this statement is similar to Visual Basic's `Select Case` statement, but in T-SQL, it's called CASE.

To include the value returned by the CASE statement to the result set, you must combine the SELECT and CASE statements as shown here:

```
SELECT @premium=
  CASE @CarCategory
    WHEN 'COMPACT' THEN 25.5
```

```

        WHEN 'ECONOMY' THEN 37.5
        WHEN 'LUXURY'   THEN 45.0
    END

```

The SELECT keyword in the previous line simply tells SQL Server to display the outcome of the CASE statement. If the variable *@CarCategory* is “ECONOMY,” then the value 37.5 is printed in the Results pane of the Query Analyzer’s window.

T-SQL CASE STATEMENT VS. VB SELECT CASE STATEMENT

As a VB programmer, sooner or later you’ll code a SQL CASE statement as SELECT CASE. The result will not be an error message. The statement will simply select the result of the CASE statement (SELECT is a T-SQL keyword that assigns a value to a variable). Let’s clarify this with an example. The following statements will return the value 7.5. This value will be printed in the Results pane of the Query Analyzer, but you won’t be able to use it in the statements following the CASE statement.

```

DECLARE @state char(2)
SET @state = 'CA'
SELECT CASE @state
    WHEN 'AZ' THEN 5.5
    WHEN 'CA' THEN 7.5
    WHEN 'NY' THEN 8.5
END

```

If you want to store the result to a variable, use the following syntax instead:

```

DECLARE @state char(2)
DECLARE @stateTAX real
SET @state = 'CA'
SET @stateTAX =
    CASE @state
        WHEN 'AZ' THEN 5.5
        WHEN 'CA' THEN 7.5
        WHEN 'NY' THEN 8.5
    END
PRINT @stateTAX

```

This syntax has no counterpart in Visual Basic. Note that the entire CASE statement is, in effect, embedded into the assignment. The @stateTAX variable is set to the value selected by the CASE statement.

WHILE

The WHILE statement repeatedly executes a single statement or a block of T-SQL statements. If you want to repeat multiple statements, enclose them in a pair of BEGIN/END keywords, as explained in the description of the IF statement. The most common use of the WHILE statement is to scan the rows of a cursor, as shown in the following example:

```

FETCH NEXT INTO variable_list
WHILE @@FETCH_STATUS = 0

```

```

BEGIN
    { statements to process the fields of the current row }
    FETCH NEXT INTO variable_list
END

```

The `FETCH NEXT` statement reads the next row of a cursor set and stores its fields' values into the variables specified in the *variable_list*, which is a comma-separated list of variables. Finally, `@@FETCH_STATUS` is a global variable that returns 0 while there are more records to be fetched. When we reach the end of the cursor, `@@FETCH_STATUS` returns `-1`.

CONTINUE AND BREAK

These two keywords are used in conjunction with the `WHILE` statement to alter the flow of execution. The `CONTINUE` keyword ends the current iteration and forces another one. In other words, the `WHILE` statement's condition is evaluated and the loop is re-entered. If the condition is False, then the `WHILE` loop is skipped and execution continues with the line following the `END` keyword that delimits the loop's body of statements.

The `BREAK` keyword terminates the loop immediately and branches to the line following the loop's `END` keyword. The following code segment shows how the two keywords are used in a `WHILE` loop:

```

WHILE <condition>
BEGIN
    { read column values into variables }
    IF @balance < 0
        CONTINUE
    IF @balance > 999999
        BREAK
    { process @balance variable and/or other variables }
END

```

This loop reads the rows of a table or cursor and processes only the ones with a positive balance, less than 1,000,000. If a row with a negative balance is found, the code doesn't process it and continues with the next row. If a row with a balance of 1,000,000 or more is found, the code stops processing the rows by breaking out of the loop.

GOTO AND RETURN

These are the last two flow-control statements, and they enable you to alter the flow of execution by branching to another location in the procedure. The `GOTO` statement branches to a line identified by a label. Here's a simple example of the `GOTO` statement (in effect, it's a less elegant method of implementing a `WHILE` loop):

```

RepeatLoop:
    FETCH NEXT INTO variable_list
    IF @@FETCH_STATUS = 0
        BEGIN
            { process variables }
            GOTO RepeatLoop
        END
END

```

While more records are in the result set, the GOTO statement branches to the FETCH NEXT statement. The identifier *RepeatLoop* is a label (a name identifying the line to which you want to branch), and it must be followed by a colon. If there are no more records to be fetched and processed, the procedure continues with the statement following the END keyword.

The RETURN statement ends a procedure unconditionally and, optionally, returns a result. To return a value from within a stored procedure, use a statement like

```
RETURN @error_value
```

@error_value is a local variable, which can be set by the procedure’s code. The calling application, which could be another stored procedure, should be aware of the possible values returned by the procedure.

If you don’t specify your own error code, SQL Server returns its error code, which is one of the values shown in Table 3.

TABLE 3: RETURN STATEMENT ERROR CODES

ERROR CODE	DESCRIPTION
-1	Missing object
-2	Data type error
-3	Process involved in deadlock
-4	Permission error
-5	Syntax error
-6	User error
-7	Resource error
-8	Internal problem
-9	System limit reached
-10	Internal inconsistency
-11	Internal inconsistency
-12	Corrupt table or index
-13	Corrupt database
-14	Hardware error

Miscellaneous Statements

In addition to the flow-control statements, T-SQL provides other statements as well, of which the PRINT and RAISERROR statements are frequently used in stored procedures. The PRINT statement shouldn’t normally appear in a stored procedure (it doesn’t produce output that’s returned to the calling application), but it’s used frequently in quickly debugging a stored procedure.

PRINT

The PRINT statement is similar to Visual Basic's `Debug.Print` method: it prints its argument to the Results pane and is used for debugging purposes. The output of the PRINT statement doesn't become part of the cursor returned by the procedure or T-SQL statement. The syntax of the PRINT statement is:

```
PRINT output_list
```

The *output_list* can be any combination of literals, variables, and functions. To display a message, use a statement like the following one:

```
PRINT "No rows matching your criteria were found in the table"
```

You can also display variable values along with literals, but this requires some conversions. Unlike Visual Basic's `Print()` function, the T-SQL PRINT statement can't print multiple arguments separated with commas. You must format all the information you want to print as strings, concatenate them with the `+` operator, and then print them. If you want to print a customer name (field `CustName`) and a total (field `CustTotal`), you can't use a statement like:

```
PRINT CustName, CustTotal -- WRONG!
```

Instead, you must concatenate the two values and print them as a single string. Since T-SQL is a typed language, you must first convert the numeric field to a string value and concatenate the two:

```
PRINT CustName + CONVERT(char(12), CustTotal)
```

The `CONVERT()` function converts (casts) a data type to another data type, and it's discussed later in this chapter.

NOTE Normally, all the output produced by the T-SQL statements forms the result set. This output is usually the data retrieved from the database. To include titles or any other type of information in the result set, use the PRINT statement. When you use the PRINT statement with the Query Analyzer, its output appears in the lower pane of the window. You can't use the PRINT statement to insert additional lines into a cursor.

RAISERROR

Normally, when an error occurs during the execution of a stored procedure, SQL Server displays an error message in the lower pane and aborts the execution. It is possible to raise your own error messages from within your stored procedures with the RAISERROR statement. The syntax of the statement is

```
RAISERROR errorNum, severity, state
```

Notice that the statement is spelled with one *E*. The first argument is the error's number, and it must be a value in the range 50,001 to 2,147,483,648. The first 50,000 error codes are reserved for SQL Server. The second argument is the severity of the error and must be a value from 1 to 18 (18 being the most severe custom error). The last argument is an integer in the range 1 to 127, and you can use it to return additional information about the error.

The errors raised by SQL Server have a description, too. To associate a description with your custom errors, use the *sp_addmessage* system stored procedure, whose syntax is:

```
sp_addmessage errorNum, severity, errorDescription
```

This statement adds new error messages to a database and associates them to error numbers. To add a custom error, use a statement like the following one:

```
sp_addmessage 60000, 15, "Can't accept IOU from this customer"
```

Then, every time you run into a customer you can't accept IOUs from, raise this error with the *RAISERROR* statement. Your error message will be handled just like any native SQL Server error message.

```
IF @CustomerID='BLKDE' OR @Customer='BDCST'
    RAISERROR 60000
```

T-SQL Functions

T-SQL supports functions that simplify the processing of the various data types, interacting with the server and implementing the operations you take for granted in application development. In this section we'll discuss the T-SQL functions by comparing them to the equivalent VB functions. Some T-SQL functions have no equivalent in VB, and these functions are discussed in more detail. Notice that the following functions are part of T-SQL, but they can be used outside T-SQL procedures. You can use them in the Query Analyzer's window to process, or format, the output of plain SQL statements.

STRING MANIPULATION

T-SQL supports functions for the manipulation of strings, which are similar to the equivalent string functions of Visual Basic, but many of them have different names. The following list summarizes these functions and their arguments.

ASCII(character)

Same as the *Asc()* function of VB.

CHAR(value)

Same as the *Chr()* function of VB.

CHARINDEX(pattern, expression)

Similar to the *InStr()* function of VB. The *CHARINDEX()* function returns the position of the first occurrence of the first argument in the second argument.

LEFT(string, len)

Same as the *Left()* function of VB.

RIGHT(string, len)

Same as the `Right()` function of VB.

LEN(string)

Same as the `Len()` function of VB. It returns the length of a string. The `LEN()` function will return the length of the string stored in the variable, even if the variable was declared as `char`.

DATALength(variable)

This function returns the declared length of a variable or field, unlike the `LEN()` function, which returns the length of the string stored in the variable.

LOWER(string)

Same as the `LCase()` function of VB.

UPPER(string)

Same as the `UCase()` function of VB.

LTRIM(string)

Same as the `LTrim()` function of VB.

RTRIM(string)

Same as the `RTrim()` function of VB.

SPACE(value)

Same as the `Space()` function of VB.

STR(float, length, decimal)

This function converts a float expression to a string, using *length* digits in all and *decimal* digits for the fractional part of the expression. If the numeric value has more fractional digits than specified with the *decimal* argument, the `STR()` function will round, not simply truncate the numeric value. The function `PRINT STR(3.14159, 4, 2)` will return the value 3.14, while the expression `PRINT STR(3.14159, 5, 3)` will return 3.142.

REPLACE(string1, string2, string3)

Same as the `Replace()` function of VB. Replaces all occurrences of *string2* in *string1* with *string3*.

REPLICATE(char, value)

Same as the `String()` function of VB. It returns a new string by repeating the *char* character *value* times. (Notice that the `REPLICATE` function's arguments appear in reverse order from the VB version.)

REVERSE(string)

Same as the StrReverse() function of VBA. It reverses the order of the characters in the specified string.

SUBSTRING(string, start, length)

Similar to the Mid() function of VB. It returns a segment of a text or binary value. *string* is the variable on which the SUBSTRING() function will act, and *start* and *length* are the starting position and length of the desired segment of the string. The function

```
SUBSTRING('February 21, 1999',10,2)
```

returns the string "21" (the two characters at the tenth position in the specified string).

NCHAR(value)

Returns the Unicode character whose value is *value*.

PATINDEX(pattern, expression)

Similar to the InStr() function of VB and the CHARINDEX() function of T-SQL. It returns the location of the first occurrence of the *pattern* string in the *expression* string.

STUFF(string1, start, length, string2)

Replaces part of *string1* with *string2*. The part of the first string to be replaced is determined by the position of the first character (*start*) and its length. If you execute the following lines:

```
DECLARE @a char(30)
SET @a="Visual Basic Programming"
SET @a= STUFF(@a, 1, 12, "VB.NET")
PRINT @a
```

the string "VB.NET Programming" will be printed.

QUOTENAME(string, quote_char)

Returns a Unicode string with *quote_char* as the delimiter. The function

```
QUOTENAME(ContactName, '''')
```

returns the value of the field in a pair of double quotes:

```
"Antonio Morena"
```

You can also use characters that appear in pairs to enclose the string. The function

```
QUOTENAME(ContactName, '{ }')
```

returns a string like the following one:

```
{Antonio Morena}
```

Notice that the function provides the proper closing symbol automatically and you can't specify a symbol that doesn't have a matching closing symbol. For example, you can't enclose a string in a pair of @ symbols with a statement like the following:

```
QUOTENAME(ContactName, '@')      -- ILLEGAL
```

This won't produce a result.

SOUNDEX FUNCTIONS

The following few functions are unique to T-SQL, and they are quite interesting. They're based on the SOUNDEX algorithm, which compares strings based on their sounds. The SOUNDEX algorithm measures how close two words are, rather than whether they are identical or not, and it does so by taking into consideration the first letter of each word, its consonants, and the closeness of the sound of various letters (such as *p*, *b*, *t*, and *d*).

DIFFERENCE(string1, string2)

Returns the difference between two strings as a value between 0 and 4. This function uses the SOUNDEX algorithm to compare sounds, rather than characters. If the two strings match phonetically very well, the DIFFERENCE() function will return the value 4. If they don't resemble each other even remotely, it will return the value 0.

The DIFFERENCE function is commonly used to search with names. Instead of a comparison like:

```
WHERE Authors.AuthorName = sAuthor
```

where *sAuthor* is the name supplied by the user, you can use the following comparison:

```
WHERE DIFFERENCE(Authors.AuthorName, sAuthor) = 4
```

The latter comparison will match "Mansfield" with "Mansfeeld", "Petroutsos" with "Petrutsos", and so on.

SOUNDEX(string)

Returns the four-character SOUNDEX code of a word. You can use this value to compare the closeness of two words. The SOUNDEX function converts a string to a four-character code: the first character of the code is the first character of the *string* argument, and the second through fourth characters of the code are numbers that code the consonants in the word (vowels are ignored unless they are the first letter of the string). The strings "Brannon", "Branon", and "Bramon" all have a SOUNDEX value of B655, because the sounds of *n* and *m* are very close. The word "Bradon", however, has a SOUNDEX value of B635 (it differs by a consonant in the middle of the word). The word "Branok" has a SOUNDEX value of B652, since it differs in a consonant at the end.

CONVERSION FUNCTIONS

The internal representation of the various columns is not always the most appropriate for reports. T-SQL provides two functions for converting between data types, which are described next.

CONVERT(*data_type*, *variable*, *style*)

The CONVERT() function converts a variable (or column value) to the specified data type. The *data_type* argument can be any valid SQL Server data type. If the data type is nchar, nvarchar, char, varchar, binary, or varbinary, you can specify its length as a numeric value in parentheses.

The *style* argument specifies the desired date format when you convert datetime and smalldatetime variables to character data, and its values are shown in Table 4.

TABLE 4: VALUES OF THE CONVERT() FUNCTION'S STYLE ARGUMENT		
VALUE (WITHOUT CENTURY)	VALUE (WITH CENTURY)	DATE FORMAT
1	101	mm/dd/yy
2	102	yy.mm.dd
3	103	dd/mm/yy
4	104	dd.mm.yy
5	105	dd-mm-yy
6	106	dd mon yy
7	107	mon dd, yy
8	108	hh:mm:ss
9	109	mmm dd yyyy hh:mi:ss:mmmAM (or PM)
10	110	mm-dd-yy
11	111	yy/mm/dd
12	112	yymdd
13	113	dd mon yyyy hh:mm:ss:mmm(24h)
14	114	hh:mi:ss:mmm(24h)
20	120	yyyy-mm-dd hh:mi:ss(24h)
21	121	yyyy-mm-dd hh:mi:ss:mmm(24h)

Use the CONVERT() function to convert data types before displaying them on the Query Analyzer's window, or to concatenate character and numeric values before passing them to the PRINT statement.

The CompanyName and ContactName columns in the Customers table of the Northwind database have a maximum length of 40 and 30 characters, respectively. If they are included in a query's output, they can nearly overflow the width of the Query Analyzer's window on a typical monitor. You can display them in shorter columns with a statement like the following one:

```
USE NORTHWIND
SELECT CONVERT(nchar(30), CompanyName),
       CONVERT(nchar(15), ContactName)
FROM Customers
```

The following statement retrieves the total for all companies and displays it along with the company's name:

```
USE NORTHWIND
SELECT CONVERT(char(25), CompanyName),
        CONVERT(char(10), SUM(Quantity * UnitPrice * (1 - Discount)))
FROM [Order Details], Orders, Customers
WHERE [Order Details].OrderID = Orders.OrderID AND
      Orders.CustomerID = Customers.CustomerID
GROUP BY CompanyName
ORDER BY CompanyName
```

A section of output produced by the above statement is shown here:

Around the Horn	13806.8
Berglunds snabbköp	24927.6
Blauer See Delikatessen	3421.95
Blondesddsl père et fils	18534.1
Bólido Comidas preparadas	4232.85
Bon app'	21963.3
Bottom-Dollar Markets	20801.6

If you omit the `CONVERT()` functions, the company name will be displayed in 40 spaces and the totals with too many fractional digits:

Around the Horn	13806.800003051758
Berglunds snabbköp	24927.57746887207
Blauer See Delikatessen	3421.9500045776367

CAST(variable AS data_type)

The `CAST()` function converts a variable or column value to the specified data type. This function doesn't do anything more than the `CONVERT()` function, but it's included for compatibility with the SQL-92 standard.

DATE AND TIME FUNCTIONS

Like Visual Basic, T-SQL recognizes several date- and time-manipulation functions, which are summarized next.

GETDATE()

Returns the current date and time on the machine that runs SQL Server.

DATEADD(interval, number, datetime)

Increments its *datetime* argument by the specified number of intervals. The *interval* argument can be one of the constants shown in Table 5.

TABLE 5: INTERVAL VALUES OF THE DATE AND TIME FUNCTIONS

INTERVAL	VALUE	RANGE
year	yy, yyyy	1753–9999
quarter	qq, q	1–4
month	mm, m	1–12
dayofyear	dy, y	1–366
day	dd, d	1–31
week	wk, ww	1–53
weekday	Dw	1–7
hour	Hh	0–23
minute	mi, n	0–59
second	Ss	0–59
millisecond	Ms	0–999

DATEDIFF(interval, datetime1, datetime2)

Returns the difference between two datetime arguments in various *intervals*, which can be days, hours, months, years, and so on (see Table 5).

DATEPART(interval, datetime)

Returns an integer that represents the number of specified parts of a given date.

DATENAME(interval, datetime)

Returns the name of a part of a datetime argument. The function DATENAME(month, varDate) returns the name of the month in the *varDate* argument (January, February, and so on), and the DATENAME(weekday, varDate) returns the name of the weekday in the *varDate* argument (Monday, Tuesday, and so on). You have seen an Access query that retrieves the number of orders placed on each day of the week. This query uses the Access WeekDay() function to extract the day of the week from the OrderDate field. Here’s the equivalent statement in T-SQL:

```
USE NORTHWIND
SELECT DATENAME(weekday, [OrderDate]), Count(OrderID)
FROM Orders
GROUP BY DATENAME(weekday, [OrderDate])
```

If you execute the above statement in the Query Analyzer, you will see the following output:

Friday	164
Monday	165
Thursday	168
Tuesday	168
Wednesday	165

DAY(), MONTH(), YEAR()

These functions return the weekday, month, and year part of their argument, which must be a date-time value.

OTHER FUNCTIONS

T-SQL supports some functions that have no equivalent in Visual Basic. These functions are used to manipulate field values and handle Null values, which are unique to databases.

COALESCE(expression, expression, ...)

This function returns the first non-NULL expression in its argument list. The function `COALESCE(CompanyName, ContactName)` returns the company's name, or the customer's name should the `CompanyName` be NULL. Use this function to retrieve alternate information for Null columns in your `SELECT` statements:

```
SELECT CustomerID, COALESCE(CompanyName, ContactName)
FROM Customers
```

You can also use a literal as the last argument, in case all the values retrieved from the database are Null:

```
SELECT CustomerID, COALESCE(CompanyName, ContactName, "MISSING!")
FROM Customers
```

ISNULL(column, value)

The `ISNULL()` function is totally different from the `IsNull()` function of VB. The T-SQL `ISNULL()` function accepts two arguments, a field or variable name and a value. If the first argument is not Null, then the function returns its value. If the first argument is Null, then the *value* argument is returned. If certain products haven't been assigned a value, you can still include them in a price list with a zero value:

```
SELECT ProductName, ISNULL(UnitPrice, 0.0)
FROM Products
```

This statement will select all the rows in the `Products` table along with their prices. Products without prices will appear with a zero value (instead of the "Null" literal).

NULLIF(expression1, expression2)

The `NULLIF()` function accepts two arguments and returns NULL if they are equal. If not, the first argument is returned. The `NULLIF()` function is equivalent to the following `CASE` statement:

```
CASE
  WHEN expression1 = expression2 THEN NULL
  ELSE expression1
END
```

SYSTEM FUNCTIONS

Another group of functions enables you to retrieve system information.

SUSER_ID()

Accepts the user's login name as an argument and returns the user's ID.

SUSER_NAME()

Does the opposite: it accepts the user's ID and returns the user's login name.

USER_ID() and USER_NAME()

These functions are similar, only instead of the login name, they work with the database username and database ID.

USER

Accepts no arguments and must be called without the parentheses; it returns the current user's database username. If you have logged in as "sa" your database username will most likely be "dbo."

Using Arguments

Stored procedures wouldn't be nearly as useful without the capacity to accept arguments. Stored procedures are implemented as functions: they accept one or more arguments and return one or more values to the caller. Stored procedures also return one or more cursors. A cursor is the equivalent of an ADO Recordset (not quite the same as the DataSet object of ADO.NET). It's the output produced in the Results pane of the Query Analyzer's window when you execute a SELECT statement.

The arguments passed to and from a stored procedure must be declared immediately after the CREATE PROCEDURE statement. They appear as comma-separated lists after the procedure name and before the AS keyword, as shown here:

```
CREATE PROCEDURE procedure_name
    @argument1 type1, @argument2 type2, . . .
AS
```

Notice that you don't have to use the DECLARE statement. Other variables declared in the procedure's body must be prefixed with the DECLARE keyword. Listing 5 shows a simple stored procedure that accepts two datetime arguments and returns the orders placed in the specified interval.

LISTING 5: THE ORDERSBYDATE STORED PROCEDURE

```
CREATE PROCEDURE OrdersByDate
    @StartDate datetime, @EndDate datetime
AS
SELECT * FROM Orders
WHERE OrderDate BETWEEN @StartDate AND @EndDate
```

To test this procedure, you must first attach it to the Northwind database. Select Northwind in the DB box, and then execute the above lines by pressing Ctrl+E. Then open a new query window and execute the following lines:

```
DECLARE @date1 datetime
DECLARE @date2 datetime
SET @date1='1/1/1997'
SET @date2='3/31/1997'
EXECUTE OrdersByDate @date1, @date2
```

The orders placed in the first quarter of 1997 will appear in the Results pane. Notice that you didn't have to specify the output cursor as an argument; the rows retrieved are returned automatically to the caller. Let's add an output parameter to this stored procedure. This time we'll request the number of orders placed in the same interval. Here's the CountOrdersByDate stored procedure:

```
CREATE PROCEDURE CountOrdersByDate
@StartDate datetime, @EndDate datetime,
@CountOrders int OUTPUT
AS
SELECT @CountOrders = COUNT(OrderID) FROM Orders
WHERE OrderDate BETWEEN @StartDate AND @EndDate
```

The argument that will be returned to the procedure is marked with the OUTPUT keyword. Notice also that it must be assigned a value from within the stored procedure's code. The SELECT statement assigns the values returned by the SELECT query to the *@CountOrders* variable.

To test the new procedure, execute the following lines. The output they'll produce is "There were 92 orders placed in the chosen interval."

```
DECLARE @date1 datetime
DECLARE @date2 datetime
SET @date1='1/1/1997'
SET @date2='3/31/1997'
DECLARE @orderCount int
EXECUTE CountOrdersByDate @date1, @date2, @orderCount OUTPUT
PRINT 'There were ' + CONVERT(varchar(5), @orderCount) +
      ' orders placed in the chosen interval'
```

This batch is very similar to the batch we used to test the OrdersByDate procedure, with the exception of the new argument. In addition to declaring the argument, you must specify the OUTPUT keyword to indicate that this argument will be passed back to the caller. You can specify input/output arguments, which pass information to the procedure when it's called and return information back to the caller. The INPUT keyword is the default, so you don't have to specify it explicitly.

Building SQL Statements on the Fly

The BookSearch procedure searches the Titles table of the Pubs database to locate titles that contain one or more words. It accepts up to five arguments, and it forms the proper SELECT statement

based on how many arguments are not empty. The desired SELECT statement should be something like:

```
SELECT Title FROM Titles
WHERE Title LIKE '%keyword1%' AND Title LIKE '%keyword2%'
```

There can be up to five keywords, so you can't use a predefined SQL statement in the procedure. You must build the desired statement on the fly. You must add as many LIKE clauses as there are non-empty arguments. The SearchBooks procedure must be called with five arguments, even if four of them are left empty. After building the SQL statement, it's executed with the EXECUTE statement, and the selected rows are returned to the caller. Listing 6 provides the code for the BookSearch procedure.

LISTING 6: THE BOOKSEARCH STORED PROCEDURE

```
USE PUBS
IF EXISTS (SELECT name FROM sysobjects WHERE name = 'BookSearch')
    DROP PROCEDURE BookSearch
GO
CREATE PROCEDURE BookSearch
    @Arg1 varchar(99), @Arg2 varchar(99),
    @Arg3 varchar(99), @Arg4 varchar(99), @Arg5 varchar(99)
AS
    DECLARE @SearchArg varchar(999)
    DECLARE @SQLstring varchar(999)
    SET @Arg1 = LTRIM(RTRIM(@Arg1))
    SET @Arg2 = LTRIM(RTRIM(@Arg2))
    SET @Arg3 = LTRIM(RTRIM(@Arg3))
    SET @Arg4 = LTRIM(RTRIM(@Arg4))
    SET @Arg5 = LTRIM(RTRIM(@Arg5))
    SET @SearchArg = "WHERE "
    IF LEN(@Arg1) > 0
        SET @SearchArg = @SearchArg + "Title LIKE '%" + @Arg1 + "%' AND "
    IF LEN(@Arg2) > 0
        SET @SearchArg = @SearchArg + "Title LIKE '%" + @Arg2 + "%' AND "
    IF LEN(@Arg3) > 0
        SET @SearchArg = @SearchArg + "Title LIKE '%" + @Arg3 + "%' AND "
    IF LEN(@Arg4) > 0
        SET @SearchArg = @SearchArg + "Title LIKE '%" + @Arg4 + "%' AND "
    IF LEN(@Arg5) > 0
        SET @SearchArg = @SearchArg + "Title LIKE '%" + @Arg5 + "%' AND "
    IF @SearchArg > 6
        SET @SearchArg = SUBSTRING(@SearchArg, 1, LEN(@SearchArg)-4)
    IF LEN(@SearchArg) = 6
        SET @SQLstring = "SELECT Title FROM Titles"
    ELSE
        SET @SQLstring = "SELECT Title FROM Titles " + @SearchArg
    PRINT @SQLstring
    EXECUTE (@SQLstring)
```

To test the BookSearch stored procedure, you must call it with five arguments. Some of them may be empty, but you must declare and pass five variables, as shown here:

```
DECLARE @arg1 varchar(99)
DECLARE @arg2 varchar(99)
DECLARE @arg3 varchar(99)
DECLARE @arg4 varchar(99)
DECLARE @arg5 varchar(99)
SET @arg1 = 'computer'
SET @arg2 = 'stress'
EXECUTE BookSearch @arg1, @arg2, @arg3, @arg4, @arg5
```

If all parameters are empty, then the stored procedure will select all the titles in the database. The BookSearch procedure will return a single title, as well as the SELECT statement it executed. (I've included the search argument generated by the stored procedure for testing purposes.) If you execute the lines shown above, the following will be printed in the Results pane:

```
SELECT Title FROM Titles WHERE Title
      LIKE '%computer%' AND Title LIKE '%stress%'
Title
-----
You Can Combat Computer Stress!
```

Unlike Visual Basic, T-SQL doesn't support a variable number of arguments. If you want to write a stored procedure that accepts an unknown number of arguments, you can create a long string with all the argument values and parse this string in the procedure. You'll see an example of this technique in the section "Implementing Business Rules with Stored Procedures," later in this chapter.

Transactions

A *transaction* is a series of database operations that must succeed or fail as a whole. If all operations complete successfully, then the entire transaction succeeds and the changes are committed to the database. If a single operation fails, then the entire transaction fails and no changes are written to the database. If the transaction fails, the changes are removed from the tables and replaced with their original values.

SQL implements transactions with three statements:

BEGIN TRANSACTION

This statement marks the beginning of a transaction. If the transaction fails, the tables will be restored to the state they were at the moment the BEGIN TRANSACTION statement was issued. It is implied here that the database is restored to a previous state with regard to the changes made by your application. Changes made to the database by others are not affected.

COMMIT TRANSACTION

This statement marks the successful end of a transaction. When this statement is executed, all the changes made to the database by your application since the execution of the BEGIN TRANSACTION

statement are committed finally and irrevocably to the database. You can undo any of the changes later on, but not as part of the transaction.

ROLLBACK TRANSACTION

This statement marks the end of an unsuccessful transaction. When this statement is executed, the database is restored to the state it was when the `BEGIN TRANSACTION` statement was executed, as if the statements between `BEGIN TRANSACTION` and `ROLLBACK TRANSACTION` were never executed.

The following code segment shows how the transaction-related statements are used in a batch:

```
BEGIN TRANSACTION
{ T-SQL statement }
IF @@ERROR <> 0
BEGIN
    ROLLBACK TRANSACTION
    RETURN -100
END
{T-SQL statement}
IF @@ERROR <> 0
BEGIN
    ROLLBACK TRANSACTION
    RETURN -101
END
COMMIT TRANSACTION
{ more T-SQL statements }
```

The error codes `-100` and `-101` identify error conditions. After each operation you must examine the `@@ERROR` variable. If it's not zero, then an error occurred and you must roll back the transaction. If all operations succeed, you can commit the transaction.

Implementing Business Rules with Stored Procedures

One of the buzzwords in modern application development is the *business rule*. Business rules are the basic rules used by corporations to manipulate their data. A company may have a complicated rule for figuring out volume discounts. Some companies keep track of the current inventory quantities, while others calculate it on the fly by subtracting sales from purchases. So, what's so important about business rules? Modern databases and programming techniques allow you to incorporate these rules either into the database itself, or implement them as special components that can be called, but not altered, by other applications.

In the past, business rules were implemented in the application level. A good deal of the analysis was devoted to the implementation of business rules. If the management changed one of these rules, programmers had to revise their code in several places. A misunderstanding could lead to different implementations of the same rules. A programmer might implement the business rules for the applications used at the cash register, and another programmer might implement the same rules for an ordering system over the Web.

Business rules can be implemented as stored procedures. You can implement stored procedures for all the actions to be performed against the database, embed the business rules in the code, and ask application programmers to act on the database through your procedures. Most programmers need not even understand the business rules. As long as they call stored procedures to update the underlying tables, their applications will obey these rules.

Let's say your corporation wants to enforce these two simple business rules:

- ◆ Orders can be cancelled but not altered or removed from the system under any circumstances.
- ◆ When items are sold or purchased, the Level field must be updated accordingly, so that the units in stock are available instantly.

If you grant the users, or programmers, of the database write access to the Products or Order Details tables, you can't be sure these rules are enforced. If you implement a stored procedure to add new orders and another stored procedure to cancel orders (a procedure that would mark an order as canceled but would not remove it from the table), you enforce both rules. Applications could enter new orders into the system, but they would never fail to update the Level field, because all the action would take place from within the stored procedure. To make sure they call your stored procedure to delete rows in the Orders table, give them read-only privileges for that table.

The same component can also be used by various tiers. Let's say you decide to go online. Users will place orders through the Web, the data is transmitted to the Web server, and then it is moved to the database server by calling the same stored procedure from within an ASP.NET application that runs on the Web server. Not only do you simplify the programming, but you also protect the integrity of the database. Assuming that you can't write every line of code yourself, you can't count on programmers to implement the same operations in different applications and different languages (not to mention what would happen when business rules are revised, or new ones are added).

Another benefit of incorporating business rules into the database is that you can change the rules at any time and applications will keep working. You may decide to record the sales of some special items into separate tables; you may even decide to keep canceled orders in another table and remove their details from the Order Details table. The applications need not be aware of the business rules. You change the stored procedures that implement them and walk away. All the applications that update the database through the stored procedures will automatically comply with the new business rules.

In this final section of the chapter, we'll build a couple of stored procedures for some of the most common tasks you will perform with a database like Northwind. The first one adds new customers to the Customers table—a straightforward procedure. The second one adds orders and is a fairly complicated stored procedure, because it must update two tables in a transaction.

Adding Customers

I'll start with a simpler example of a stored procedure that implements a business rule: writing a stored procedure to add customers. The stored procedure accepts as arguments all the columns of the Customers table and inserts them into a new row. The addition of the new row is implemented with the INSERT statement. If the insertion completes successfully, the error code zero is returned. If the insertion fails, the procedure returns the error code generated by the INSERT statement. Practically, the only reason this action would fail is because the ID of the customer you attempt to

add exists in the Customers table already. The AddCustomer stored procedure doesn't even examine whether a customer with the same ID exists already. If the customer exists already, the INSERT statement will fail. Besides, there's always a (very slim) chance that between the test and the actual insertion another user might add a customer with the same ID. This is a condition that can't be prevented, unless you're willing to lock the entire table for the duration of the insertion operation. The safest approach is to detect the error after the fact and notify the user.

The AddCustomer stored procedure is shown in Listing 7. At the beginning, it declares all the input arguments. Then it uses these arguments as values in an INSERT statement, which adds a new row to the Customers table. If the INSERT statement encounters an error, the procedure doesn't crash. It examines the value of the @@ERROR global variable. If the @@ERROR variable is not zero, then an error occurred. If the operation completed successfully, the @@ERROR variable is zero.

LISTING 7: THE ADDCUSTOMER.SQL STORED PROCEDURE

```
USE NORTHWIND
IF EXISTS (SELECT name FROM sysobjects
           WHERE name = 'AddCustomer')
    DROP PROCEDURE AddCustomer
GO
CREATE PROCEDURE AddCustomer
    @custID nchar(5), @custName nvarchar(40), @custContact nvarchar(30),
    @custTitle nvarchar(30), @custAddress nvarchar(60), @custcity nvarchar(15),
    @custRegion nvarchar(15), @custPostalCode nvarchar(10),
    @custCountry nvarchar(15), @custPhone nvarchar(24), @custFax nvarchar(24)
AS
DECLARE @ErrorCode int
INSERT Customers (CustomerID, CompanyName, ContactName, ContactTitle, Address,
                  City, Region, PostalCode, Country, Phone, Fax)
VALUES (@custID, @custName, @custContact, @custTitle, @custAddress, @custCity,
        @custRegion, @custPostalCode, @custCountry, @custPhone, @custFax)
SET @ErrorCode=@@ERROR
IF (@ErrorCode = 0)
    RETURN (0)
ELSE
    RETURN (@ErrorCode)
```

Notice that @@ERROR is stored in a local variable, which is used later in the code. The @@ERROR variable is updated after each line's execution. If you attempt to return the value @@ERROR with the RETURN statement, the calling application will receive an error code that's always zero. That's why you must store the value of @@ERROR after an operation if you want to use it later in your code.

To test the AddCustomer stored procedure, open the AddCustomer.sql file with the Query Analyzer and execute it by pressing Ctrl+E. This will attach the stored procedure to your database. Now

you can test it by calling it with the appropriate arguments. The `AddACustomer.sql` batch adds a new customer with the ID “SYBEX.” Listing 8 provides the code for the `AddACustomer.sql` batch.

LISTING 8: ADDING A NEW CUSTOMER WITH THE ADDACUSTOMER PROCEDURE

```

DECLARE @retCode int
DECLARE @custID nchar(5), @custName nvarchar(40)
DECLARE @custContact nvarchar(30)
DECLARE @custTitle nvarchar(30), @custAddress nvarchar(60)
DECLARE @custCity nvarchar(15), @custCountry nvarchar(15)
DECLARE @custPostalCode nvarchar(10), @custRegion nvarchar(15)
DECLARE @custPhone nvarchar(24), @custFax nvarchar(24)
-- Set customer data
SET @custID="SYBEX"
SET @custName="Sybex Inc."
SET @custContact="Tobias Smythe"
SET @custTitle="Customer Representative"
SET @custAddress="1000 Marina Village"
SET @custCity="Alameda"
SET @custRegion="CA"
SET @custPostalCode="90900"
SET @custCountry="USA"
SET @custPhone="(714) 3258233"
SET @custFax="(714) 3258233"
-- Call stored procedure to add new customer
EXECUTE @retCode = AddCustomer @custID, @custName, @custContact, @custTitle,
                             @custAddress, @custCity, @custRegion, @custPostalCode,
                             @custCountry, @custPhone, @custFax

PRINT @retCode

```

The `AddACustomer` batch will add a new customer only the first time it's executed. If you execute it again without changing the customer's ID, the error code 2627 will be returned, along with the following error message:

```

Violation of PRIMARY KEY constraint 'PK_Customers'. Cannot insert duplicate key in
object 'Customers'.
The statement has been terminated.

```

You can either change the values passed to the stored procedure, or switch to the Enterprise Manager, open the `Customers` table, and delete the newly added line.

Adding Orders

The next example is substantially more complicated. This time we'll write a procedure to add a new order. By its nature, this stored procedure must perform many tests, and it may abort the entire operation at various stages of its execution. The `AddOrder` stored procedure must accept the customer's ID, the employee's ID, the shipper's ID, the shipping address, and the order's details. If the specified

customer, employee, or shipper does not exist, the procedure must abort its execution and return an error code to the caller. If any of these tests fail, then the stored procedure exits and returns the appropriate error code (–100 if customer doesn’t exist, –101 if employee doesn’t exist and –102 if shipper doesn’t exist).

If these tests don’t fail, you can safely add the new order to the Orders table. The following operations are implemented as a transaction. If one of them fails, then neither an order nor details will be added to the corresponding tables. The stored procedure must add a new row to the Orders table, insert the current date in the OrderDate field, and then use the OrderID field’s value to add the order’s lines in the Order Details table. The OrderID field is assigned a value automatically by SQL Server when a row is added to the Orders table. You can find out the ID of the new order by examining the @@IDENTITY variable, which holds the value of the most recently added Identity value for the current connection. This value will be used to add detail rows in the Order Details table.

Then the order’s details are added to the Order Details table, one row at a time. Again, if one of them fails, the entire transaction will fail. The most common reason for failure is to submit a nonexistent product ID. If you force your application’s users to select product IDs from a list and validate the quantity and discount for each product, then none of the operations will fail.

The order details are passed to the AddOrder procedure as a long string, and this part deserves some explanation. Ideally, another stored procedure (or application) should be able to create a cursor and pass it as an argument to the AddOrder procedure. SQL Server’s stored procedures can’t accept cursors as arguments, so another technique for passing an unknown number of arguments has to be used. In this example, I’ve decided to store the ID, quantity, and discount of each product into a string variable. Each field has a fixed length in this string, so that it can be easily parsed. The product ID is stored as an integer in the first six characters of the string, the quantity as another integer in the next six characters, and the discount in the last six characters. Each order, therefore, takes up 18 characters. If you divide the length of this string by 18, you’ll get the number of detail lines. Then, you can call the SUBSTRING() function repeatedly to extract each detail’s values and insert them into the Order Details table.

Once the product ID has been extracted from the string variable, you can use it to retrieve the product’s price from the Products table. Here are T-SQL statements that retrieve the first product’s price and insert it along with the quantity and discount fields into the Order Details table:

```
SET @ProdID = SUBSTRING(@Details, 1, 6)
SET @Qty = SUBSTRING(@Details, 7, 6)
SET @Dscnt = SUBSTRING(@Details, 13, 6)
SELECT @Price=UnitPrice FROM Products WHERE ProductID=@ProdID
INSERT [Order Details] (OrderID, ProductID, Quantity, UnitPrice, Discount)
VALUES (@OrderID, @ProdID, @Qty, @Price, @Dscnt)
```

If a product with the specific ID doesn’t exist in the Products table, the procedure doesn’t take any special action. The INSERT statement will fail to add the detail line because it will violate the COLUMN FOREIGN KEY constraint FK_Order_Details_Products, and the procedure will roll back the transaction and return the error code 547.

Listing 9 shows the complete code of the AddOrder stored procedure. Apart from syntactical differences, it’s equivalent to the VB code you would use to add an order to the database.

LISTING 9: THE ADDORDER STORED PROCEDURE

```

USE NORTHWIND
IF EXISTS (SELECT name FROM sysobjects WHERE name = 'AddOrder')
    DROP PROCEDURE AddOrder
GO
CREATE PROCEDURE AddOrder
    @custID nchar(5), @empID int, @orderDate datetime,
    @shipperID int, @Details varchar(1000)
AS
    DECLARE @ErrorCode int
    DECLARE @OrderID int
    -- Add new row to the Orders table
    DECLARE @shipcompany nvarchar(40)
    DECLARE @shipAddress nvarchar(60), @shipCity nvarchar(15)
    DECLARE @shipRegion nvarchar(15), @shipPCode nvarchar(10)
    DECLARE @shipCountry nvarchar(15)
    SELECT @shipCompany=CompanyName,
           @shipAddress=Address,
           @shipCity=City,
           @shipRegion=Region,
           @shipPCode=PostalCode,
           @shipCountry=Country
           FROM Customers
           WHERE CustomerID = @custID
    IF @@ROWCOUNT = 0
        RETURN(-100)    -- Invalid Customer!
    SELECT * FROM Employees WHERE EmployeeID = @empID
    IF @@ROWCOUNT = 0
        RETURN(-101)    -- Invalid Employee!
    SELECT * FROM Shippers WHERE ShipperID = @shipperID
    IF @@ROWCOUNT = 0
        RETURN(-102)    -- Invalid Shipper!
    BEGIN TRANSACTION
    INSERT Orders (CustomerID, EmployeeID, OrderDate, ShipVia, ShipName,
                  ShipAddress, ShipCity, ShipRegion, ShipPostalCode, ShipCountry)
    VALUES (@custID, @empID, @orderDate, @ShipperID, @shipCompany, @shipAddress,
            @ShipCity, @ShipRegion, @shipPCode, @shipCountry)
    SET @ErrorCode=@@ERROR
    IF (@ErrorCode <> 0)
        BEGIN
            ROLLBACK TRANSACTION
            RETURN (-@ErrorCode)
        END
    SET @OrderID = @@IDENTITY
    -- Now add rows to the Order Details table

```

```

-- All new rows will have the same OrderID
DECLARE @TotLines int
DECLARE @currLine int
SET @currLine = 0
-- Use the CEILING function because the length of the
-- @Details variable may be less than 18 characters long !!!
SET @TotLines = Ceiling(Len(@Details)/18)
DECLARE @Qty smallint, @Dscnt real, @Price money
DECLARE @ProdID int
WHILE @currLine <= @TotLines
    BEGIN
        SET @ProdID = SUBSTRING(@Details, @currLine*18 + 1, 6)
        SET @Qty = SUBSTRING(@Details, @currLine*18 + 7, 6)
        SET @Dscnt = SUBSTRING(@Details, @currLine*18 + 13, 6)
        SET @currLine = @currLine + 1
        SELECT @Price=UnitPrice FROM Products WHERE ProductID=@ProdID
        INSERT [Order Details] (OrderID, ProductID, Quantity, UnitPrice, Discount)
            VALUES (@OrderID, @ProdID, @Qty, @Price, @Dscnt)
        SET @ErrorCode = @@ERROR
        IF (@ErrorCode <> 0) GOTO DetailError
    END
    COMMIT TRANSACTION
    RETURN (0)
DetailError:
    ROLLBACK TRANSACTION
    RETURN(@ErrorCode)

```

NOTE Here's the most important reason for using the *NewOrder* stored procedure. If you allow users and applications to add rows to the *Order Details* table, it's possible that someone might modify an existing order. This is a highly undesirable situation, and you should make sure it never happens. With the *NewOrder* procedure, users can't touch existing orders. Each order takes a new ID and all details inherit this ID, eliminating the possibility of altering (even by a programming mistake) an existing order.

Testing the AddOrder Procedure

To test the *AddOrder* stored procedure, you must declare some local variables, assign the desired values to them, and then execute the stored procedure with the *EXECUTE* statement, passing the variables as arguments. Most arguments represent simple fields, like the ID of the shipper, the customer ID, and so on. The last argument, however, is a long string, with 18 characters per detail line. In each 18-character segment of the string, you must store three fields: the product's ID, the quantity, and the discount. The *AddAnOrder* batch that exercises the *NewOrder* procedure is shown in Listing 10, and its output is shown in Figure 2.

LISTING 10: THE ADDANORDER.SQL SCRIPT

```

USE Northwind
DECLARE @retCode int
DECLARE @custID nchar(5), @empID int
DECLARE @orderDate datetime, @shipperID int
DECLARE @Details varchar(1000)
SET @shipperID=2
SET @custID='SAVEA'
SET @empID=4
SET @orderDate = '2/3/2002'
SET @Details="32    10    0.25 47    8    0.20"
SET @Details=@Details + "    75    5    0.05 76    15    0.10"
EXECUTE @retCode = NewOrder @custID, @empID, @orderDate, @shipperID, @Details
PRINT @retCode

```

FIGURE 2

Testing the
NewOrder stored
procedure with a
T-SQL batch

