CHAPTER 25

Creating Stored Procedures with SQL Server

In this chapter, I'm going to show you how to build and debug stored procedures using SQL Server. Stored procedures are basically subroutines that you can call from your program to perform a database task. They are written in a language called Transact-SQL, which is really just SQL with a few extra statements that help you test conditions and perform loops.

In This Chapter

Introducing stored procedures

Using Transact-SQL to build stored procedures

Creating stored procedures with SQL Server

Debugging stored procedures in Query Analyzer

Introducing Stored Procedures

A *stored procedure* is a collection of SQL statements that are stored in the database server. These statements are stored in both text form and in compiled form for fast execution. They can be used in place of an SQL statement or called like a function or subroutine.

The concept of stored procedures is common to most database management systems on the market today, though the implementations are usually sufficiently different to make converting stored procedures from one vendor to another vendor a difficult task. Yet stored procedures can make a big difference in the performance of your application so many programmers rely on them for their applications.

Why use stored procedures?

Stored procedures allow you to create a block of code that can be called from any database application or database utility. This common block of code has three primary advantages: performance, convenience, and security. A stored procedure usually needs fewer resources to run when compared to a block of regular code, coupled with calls to the database server. A stored procedure is easy to use, since it typically represents

a complicated programming object that can be used as easily as a normal SQL statement. Since a stored procedure is secured just like any other database object, you can also grant others the capability to perform a task that exceeds their normal security permissions.

Improving performance

The number one reason people use stored procedures is that they are usually more efficient than explicitly including the code in your application program. This is because when you submit an SQL statement to the database server, the following steps occur each time an application program attempts to perform a task:

1. The application program transmits the SQL statement to the database server over the network.
2. The database server parses the SQL statement and then compiles it for execution.
3. The database server executes the compiled statement.
4. The execution's results are returned over the network to the application program.
5. The application program receives the results and repeats steps 1 through 4 as needed to complete the task.

With a stored procedure, this process is much different. Before the program is run, the following steps occur:

1. The stored procedure is transmitted to the database server.
2. The stored procedure is parsed and compiled.
3. The compiled code is saved for later execution.

Then when the application program is ready to perform the same task:

1. The application program transmits a request to call the stored procedure over the network.
2. The database server retrieves the compiled copy of the stored procedure and executes it.
3. The results are returned to the application program.

Note that there is only one interaction between the application program and the database server. This reduces network traffic, which can make a big difference on heavily-loaded or low-speed networks. Also, the stored procedure is compiled before the application calls it. This saves a lot of work for the database server, since parsing and compiling SQL statements can be very CPU-intensive.

Increasing convenience

Stored procedures can be called by different application programs or called by any program that is capable of directly executing SQL statements. This means that you can develop standard stored procedures that perform a task that can be shared among all of your applications. Because the logic for the stored procedure is isolated to a single place (i.e., the database server), you can change the stored procedure without necessarily changing the application that calls it. Thus, you can change your underlying database structure, while leaving your applications untouched.

Providing security

Since a stored procedure is just another database object, it can be secured using the same techniques used to secure other database objects. Thus, you can create a stored procedure that allows your users to perform a particular task that they might not otherwise be able to perform. For instance, you might create a stored procedure to insert a row in a table that your users don't normally have access to.

Introducing Transact-SQL

Transact-SQL (also known as T-SQL) is the name of Microsoft's implementation of SQL on SQL Server. In addition to the SQL statements I've used throughout this book, there are a number of extensions that allow you to build complex stored procedures. Like any programming language, Transact-SQL consists of a number of syntax elements, such as identifiers, data types, variables, functions, expressions, and statements.

Comments

It's always a good idea to include comments in your code. There are two types of comment indicators you may use: double hyphen (`--`) and slash asterisk, asterisk slash (`/* */`).Double hyphen comments are usually used at the end of a line of code, though they can be placed on a line by themselves. Everything from the double hyphens to the end of the line is treated as a comment and is ignored by the parser. For example:

```
--
-- Procedure: GetCustomerByName
-- Written by: Wayne S. Freeze
-- Date written: 28 April 2000
-- Description: This procedure returns a customer's information
--              for a particular customer id.
--
CREATE PROCEDURE GetCustomerByName (@CustId Int)
AS
Select *
From Customers
Where CustomerId = @CustId
```

Slash asterisk, asterisk slash comments can be used anywhere a space can be used; thus, they can be embedded in your code. A slash asterisk (/*) marks the start of the comment, while an asterisk slash marks the end of a comment (*/). The comment may span multiple lines, as shown below:

```
/*
 ** Procedure: GetCustomerByName
 ** Written by: Wayne S. Freeze
 ** Date written: 28 April 2000
 ** Description: This procedure returns a customer's
 ** information for a particular customer id.
 */
CREATE PROCEDURE GetCustomerByName
   (@CustId Int /* customer id must be non-negative */ )
AS
Select *
From Customers
Where CustomerId = @CustId
```

Hiding code: Sometimes when you are debugging a stored procedure, it is useful to hide blocks of code from the server so they are not executed. One easy way to do this is to insert a line containing a slash asterisk before the code you want to hide and an asterisk slash after the code.

Identifiers

An *identifier* is simply the name of a database object, such as a database, table, or column. It can also be a Transact-SQL keyword or the name of a variable or label within a stored procedure.

There are two different types of identifiers: *delimited identifiers* and *regular identifiers*. Delimited identifiers can be any combination of characters up to 128 total. You may use letters, numbers, spaces, and any special symbol except for double quotes (") or square brackets ([]). This is because you must enclose the identifier in double quotes or inside a pair of square brackets. Some examples of delimited identifiers are:

```
"My Table"
[This identifier includes a comma, an asterisk * and a period.]
```

Regular identifiers must begin with a letter, an underscore (_), an at sign (@), or a number sign (#) and can also contain up to a maximum of 128 characters. The first character signifies how the identifier is used. System functions begin with two at signs (@@). Variables begin with an at sign, while a number sign identifies a temporary table. Some examples of regular identifiers are:

```
MyTable
@LocalVariable
```

Variables

Variables in T-SQL are basically the same as they are in Visual Basic. They hold information local to the stored procedure or represent parameters passed to the stored procedures. Variables begin with an at sign (@) and must be declared before they can be used. Before you can use a variable, you must declare it as a local variable using the **Declare** statement or as a parameter using the **Create Procedure** statement.

Each variable must be assigned a valid data type that is compatible with how you plan to use it. You can choose from the same data types that you would use in a **Create Table** statement, except for **Text**, **Ntext**, and **Image**. In the following example, I declare two variables, `@Counter` and `@Name`, which are assigned **Int** and **Varchar** data types respectively.

```
Declare @Counter Int
Declare @Name Varchar(64)
```

Functions

Functions in T-SQL are identical to those in Visual Basic. They take a series of zero or more parameters and return a value to the calling program. Table 25-1 contains some of the functions that are available for you to use in your stored procedure.

Table 25-1
Selected functions in T-SQL

Function	*Description*
@@CPU_Busy	Returns the number of milliseconds of CPU time SQL Server has consumed since it was started.
@@Cursor_Rows	Returns the number of qualifying rows for the most recently opened cursor.
@@DBTS	Returns the next timestamp for the database.
@@Error	Returns the error code for the more recently executed SQL statement.
@@Fetch_Status	Returns the status of the last **Fetch** operation.
@@IO_Busy	Returns the number of milliseconds SQL Server has spent performing I/O.
@@Nestlevel	Returns the nesting level of the current stored procedure. First level has a value of zero.

Continued

Table 25-1 *(continued)*

Function	*Description*
@@Servername	Returns the name of the database server.
@@Trancount	Returns the current nesting level of a set of nested transactions.
@@Version	Returns the date, version, and processor type for the database server.
App_Name	Returns the name of the current application.
ASCII	Returns the ASCII code of the left-most character of the specified character string.
Cast	Converts a value in one data type to another data type.
Ceiling	Returns the smallest integer value greater or equal to the specified value.
Char	Returns the character corresponding to the specified numeric ASCII code value.
Col_Length	Returns the length of the specified column.
Columnproperty	Returns the requested information about the specified column.
Convert	Returns the specified value using the specified data type using the specified style.
Current_User	Returns the name of the current user.
Cursor_Status	Returns the status of the specified cursor.
Datalength	Returns the number of bytes in the specified expression.
Floor	Returns the largest integer value less than or equal to the specified value.
Getdate	Returns the current date and time as a **Datetime** value.
Host_Name	Returns the name of the current computer. (Not the database server.)
Len	Returns the number of characters in a string, excluding trailing blanks.
Lower	Converts all uppercase characters to lowercase.
Ltrim	Removes leading blanks from a string.
IsDate	Returns **True** if the specified expression contains a valid date.
IsNumeric	Returns **True** if the specified expression is a valid number.
Is_Member	Returns **True** if the current user is a member of the specified role.
Object_Id	Converts the specified database object into a numeric object identification number.

Function	*Description*
Object_Name	Returns the name of the specified object identification number.
Patindex	Returns the location of a pattern in the specified string.
Rand	Returns a random value between 0 and 1.
Round	Rounds the specified value to the specified length or precision.
Rtrim	Removes trailing blanks from a string.
Substring	Returns the specified part of a string.
Suser_Sid	Returns the security identification number for the specified login name.
Suser_Sname	Returns the login name for the current user or the login name for the specified security identification number.
Typeproperty	Returns the requested information about a data type.
Upper	Converts all lowercase characters to uppercase.

Confusing, isn't it: To prevent confusion with system functions, you should never declare a variable with double at signs.

Expressions

Expressions are used to compute a single value based on a series of local variables, parameters, columns retrieved from a table, and functions. You can assign an expression to a variable by using the **Set** statement, as shown here.

```
Set @Counter = 0
Set @MyString = Upper(Rtrim(CustomerName))
```

You may also assign a value to a local variable using the **Select** statement. The only restriction is that the **Select** statement must return only one row. Otherwise, the variable will be assigned the last value retrieved by the **Select** statement. In front of each column, you must specify the local variable, followed by an equal sign (=), and then the column name. A simple example is shown below:

```
Select @Name = Name, @EMail = EMailAddress
From Customers
Where CustomerId = @CustId
```

Flow control

T-SQL includes flow controls statements, such as **If** and **While,** to help you build your stored procedures. You can even call other stored procedures using the **Execute** statement.

If statement

You construct an **If** statement using the following syntax:

```
If <boolean_expression>
   { <sql_statement> | Begin <sql_statement_list> End }
[Else
   { <sql_statement> | Begin <sql_statement_list> End }]
```

If `<boolean_expression>` is **True**, then the statement that immediately follows the expression will be executed. Otherwise, the statement that immediately follows the **Else** clause will be executed. You can substitute a **Begin End** pair that surrounds a list of SQL statements for the single statements if you want to execute one or more statements.

Beginnings and endings: Use **Begin** and **End** clauses even if you only have a single statement. This makes it easy to include additional statements in the future.

While statement

While statements are constructed with the following syntax:

```
While <boolean_expression>
   { <sql_statement> | Begin <sql_statement_list> End }
```

The statement or block of statements delimited by the **Begin End** pair are repeated until the `<boolean_expression>` is **False**. Inside a **Begin End** pair, you can end a **While** loop early by using the **Break** statement. The **Continue** statement ignores the rest of the statements in the **While** loop and restarts the loop.

Execute statement

The basic syntax for the **Execute** statement's syntax is shown below:

```
[Execute] [<return>=] <procedure_name>
[[<parm>=]{<value>|<var>}]...
```

The **Execute** statement is used to call another stored procedure. While the actual syntax is more complex, chances are you're not going to use much more than this. Note that the **Execute** part of the statement may be omitted. If the stored procedure returns a value, you must include a local variable for `<return>`. Otherwise, this clause should be omitted.

The name of a stored procedure follows normal database rules for the most part. However, if the name of a stored procedure begins with `sp_`, then the database server will search the master database for the stored procedure rather than the local database. If the name of the stored procedure isn't qualified and it isn't found under the current user name, the database server will search for the stored procedure using the `dbo` as the owner of the stored procedure.

For example, if you are accessing the database with the user name `MyUser` and specify the stored procedure name `MyProc`, SQL Server will look for the stored procedure named `MyUser.MyProc` in the current database. If it isn't found then it will look for the a stored procedure named `dbo.MyProc` also in the current database. If it still isn't found, then it will return an error message saying the stored procedure couldn't be found. This approach allows you to test a stored procedure with the same name and move it to under `dbo` only when you're satisfied it works properly.

Now if you call the stored procedure, `sp_MyProc`, SQL Server will look for the stored procedure `dbo.sp_MyProc` in the master database. If it isn't found there, then it will look for the stored procedure `MyUser.sp_MyProc` in the current database. If it still isn't found, then it will look for the stored procedure `dbo.sp_MyProc` in the current database. The reason it works this way is that you can't override how a system stored procedure works. Overriding a system stored procedure could compromise security.

There are two ways to specify parameters: you can simply list them in the order they were defined in the stored procedure, or you may assign a value explicitly to each parameter name. Note that if you list the parameters explicitly, you need not worry about the order you use.

The following calls are identical:

```
Execute MyProc @MyVar, 24
MyProc @MyVar, 24
Execute MyProc @Parm1 = @MyVar, @Parm2 = 24
MyProc @Parm2 = 24, @Parm1 = @MyVar
```

Exectly: You can also abbreviate **Execute** as **Exec.**

Cursors

Cursors are used to allow you to scroll through a set of rows identified by a **Select** statement. The cursor maintains the current record pointer and allows you to use other statements, such as **Fetch** to retrieve information from the current record into local variables, and **Update** to change the values in the current record.

Declare Cursor

This **Declare Cursor** statement defines a pointer and can be used to access one row of data returned by a **Select** statement:

```
Declare <cursor> Cursor
[Local|Global]
[Static | Keyset | Dynamic | Fast_Forward]
[Read_Only | Scroll_Locks | Optimistic]
For <select_statement>
```

The <cursor> is a normal SQL Server identifier that will be used by other statements to access the cursor. The **Local** keyword implies that the cursor can't be accessed outside this routine, while the **Global** keyword implies that the cursor can be accessed by other stored procedures as long as the current connection to the database is still active.

The **Static** keyword means that the database server will make a temporary copy of the data in tempdb to prevent modifications to the data while you are processing it. The **Keyset** keyword instructs the database server to keep a list of pointers to the rows, which means that your stored procedure will see the current values to the rows, but not rows that were added after the cursor was opened. The **Dynamic** keyword implies that any and all changes made to the selected rows will be visible to the stored procedure. However, this also implies that the order of rows may change as new rows are added and existing rows deleted. The **Fast_Forward** keyword implies that the rows can't be changed and that the cursor may only be moved in a forward direction (i.e., you can only use the **Fetch Next** statement).

The **Read_Only** keyword ensures that the rows can't be changed. **Scroll_Locks** implies that rows are locked when they are fetched, so that updates and deletes will always succeed. Specifying **Optimistic** means that the row isn't locked until you are ready to commit the changes. This also implies that there is the possibility that the changes may fail because the rows were changed by another program.

Where did I see that before: The keywords described here refer to the cursors and locking mechanisms that are available in ADO.

Open

In order to use the cursor, you have to use the **Open** statement. Its syntax is listed below:

Open <cursor>

The **Open** statement essentially executes the **Select** statement and creates the data structures necessary to access the rows you selected.

Fetch

The syntax for the **Fetch** statement follows:

```
Fetch [Next|Prior|First|Last|Absolute <location>|Relative
<offset>]
From <cursor> Into <variable_list>
```

The **Fetch** statement is used to retrieve information from a row. If **Next**, **Prior**, **First**, **Last**, **Absolute,** or **Relative** are specified, then the movement is performed before the information is returned. If the first call to **Fetch** after opening the cursor includes

the **Next** keyword, the current record pointer is moved to the first row and **Fetch** returns the values from the first row. For **Fast_Forward** cursors, **Next** is the only allowable movement option.

The **Absolute** keyword allows you to move the current record pointer to the specified row, where the first row is **Absolute** 1 and the second row is **Absolute** 2. The last row would be known as **Absolute** –1, while the next to the last row would be found by using **Absolute** –2. The **Absolute** keyword is not legal when using a **Dynamic** cursor.

The **Relative** keyword allows you to move the current pointer relative to the current record pointer. If the current record pointer is pointing to row 7, **Relative** –2 will reposition the current record pointer to row 5, while **Relative** 3 will move the current record pointer to row 10.

The **Into** clause requires that you supply a series of local variables to receive the columns from the **Select** statement. The local variables must be compatible with the data types returned in the **Select** statement and must appear in the same order as those listed in the **Select** statement. A runtime error will occur if there are too few or too many variables listed in the **Into** clause.

Update

The syntax for using the **Update** statement with cursors is shown below:

```
Update <table> Set [<column> = <value>] ...
Where Current Of <cursor>
```

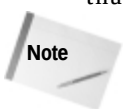
If you **Declare** a cursor that maps to a table, you can use the **Update** statement to update the row at the current record pointed to by the specified cursor by using the **Where Current Of** clause. In place of `<value>` you may use any expression of the appropriate data type, including functions and local variables.

Delete

Shown below is the syntax for using the **Delete** statement with a cursor:

```
Delete From <table>
Where Current of <cursor>
```

Substituting the table name of the table used in the **Declare Cursor** statement for `<table>` and the name of the cursor for `<cursor>` will allow you to delete the row that is pointed to by the cursor's current record pointer.



Where did my row go?: If you delete a row in a cursor's rowset and later try to read the row with a **Fetch** statement when you declared the cursor as **Static** or **Keyset**, the **@@Fetch_Status** function will return –2, meaning that the row has been deleted.

Close

The syntax for the **Close** statement is listed below:

```
Close <cursor>
```

The **Close** statement releases the results obtained when the specified cursor was opened. Any locks held also released.

Deallocate

The syntax for the **Deallocate** statement is listed below:

```
Deallocate <cursor>
```

Just because you have closed the cursor doesn't mean that the cursor is no longer available. You must **Deallocate** the cursor to free all of the resources owned by the cursor.

An example of how to use a cursor

Rather than try to build a small example for each of the above statements, I wrote a simple routine that demonstrates how to retrieve some information from your database (see Listing 25-1). This routine retrieves names from the Customers table and prints them (see Figure 25-1).

Listing 25-1: Using cursors in a simple stored procedure

```
Declare @CustName VarChar(64)
Declare @RecCount Int

Declare CustCursor Cursor
Local Fast_Forward Read_Only
For Select Name From Customers Where State = 'MD'

Open CustCursor

Set @RecCount = 0

Fetch Next From CustCursor Into @CustName
While @@Fetch_Status = 0
   Begin
   Print @CustName
   Set @RecCount = @RecCount + 1
   Fetch Next From CustCursor Into @CustName
   End
```

```
Print RTrim(Convert(VarChar(20), @RecCount)) + ' records
found.'
Close CustCursor

Deallocate CustCursor
```

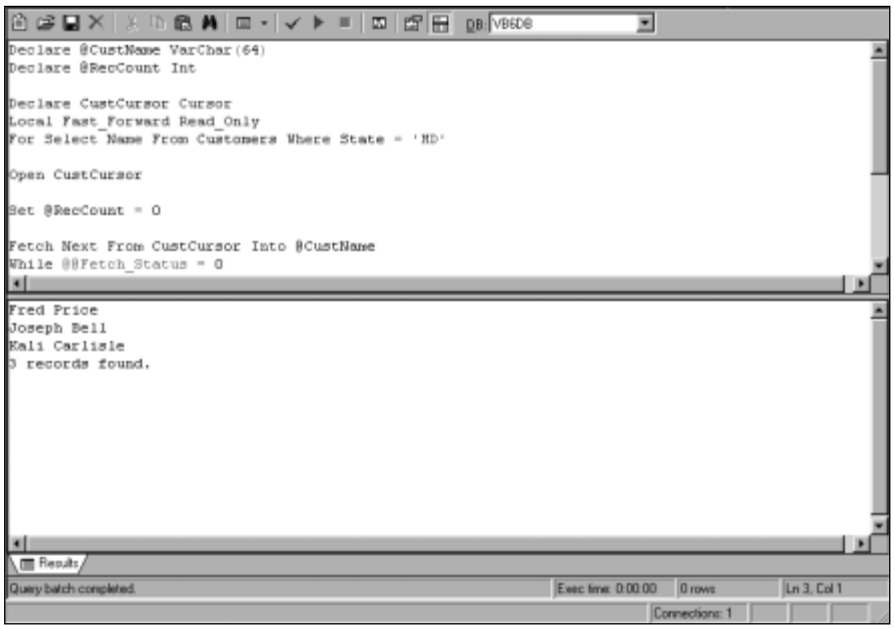


Figure 25-1: Running the sample routine

The routine begins by declaring variables to hold the customer's name and the number of records processed. Then it declares a cursor called `CustCursor` that will be used to access the information in the Customers table. To keep the amount of data to a minimum, I selected only those customers that live in Maryland. I also declare the cursor as local to this routine and choose to make it **Fast_Forward** and **Read_Only** since I'm just going to read the data in a single pass.

Next, I use the **Open** statement to open the cursor and retrieve the information from the database. After setting `@RecCount` to zero, I fetch the information from the first row into `@CustName` and start a **While** loop that will process the rest of the row in the cursor's rowset. For each row, I print the value saved in `@CustName` and use the **Fetch Next** statement to retrieve the next value. This process continues as long as the **Fetch** statement is successful (**@@Fetch_Status** =0).

At the end of the routine, I print the total number of records found, using the **Convert** function to convert the value in `@RecCount` to a string value. Then I **Close** the cursor and **Deallocate** it. This frees all of the resources associated with the cursor.

Processing transactions

Now that you know how to build simple T-SQL programs that can access individual rows in your database, I want to cover the facilities for transaction processing. As you might expect, these facilities parallel those found in ADO. Basically, you mark the beginning of a transaction and then you can either save the changes you made to the database or abort the changes without changing the database.

Begin Transaction

Begin Transaction marks the start of a transaction. This statement has the following syntax:

```
Begin Transaction [<transaction_name>]
```

You don't have to specify a value for `<transaction_name>`, but if you do, it must be a unique name following the standard rules for identifiers, though only the first 32 characters will be used. The names are only used for the outermost level of a set of nested transactions, though you may want to assign a name to any nested transactions as well to clarify which **Begin Transaction** is matched with which **Commit Transaction** or **Rollback Transaction**.

When using nested transactions, each new **Begin Transaction** increments the value in **@@Trancount**. This is important, since only the outer most transaction can commit the changes to the database. Of course, the inner transactions must complete successfully and have their results committed as well, but changes aren't actually posted to the database until the outermost transaction is committed.

Commit Transaction

The **Commit Transaction** statement saves the changes made by a transaction to the database. This statement has the following syntax:

```
Commit Transaction [<transaction_name>]
```

If the `<transaction_name>` parameter is specified, it must match the corresponding value in the **Begin Transaction** statement.

Rollback Transaction

The **Rollback Transaction** statement discards all of the changes made by a transaction to the database. This statement has the following syntax:

```
Rollback Transaction [<transaction_name>]
```

If the <transaction_name> parameter is specified, it must match the corresponding value in the **Begin Transaction** statement.

Other useful statements

There are a few other T-SQL statements that you may find useful when building a stored procedure that don't fit into any of the categories I've discussed so far.

Use

The **Use** statement specifies the database that will become the default database and has the following syntax:

```
Use <database>
```

You can use this statement at the beginning of a stored procedure to ensure that the stored procedure is running in the appropriate database. You can also use the **Use** statement to switch databases in the middle of a stored procedure.

User must exist in order to use Use: In order to switch databases, the person's login must map to a valid user, otherwise an error will occur.

Print

The **Print** statement is used to return a user-defined message to the calling program. The syntax of print is:

```
Print <string_expression>
```

where <string_expression> can be a string of text enclosed by quotes such as 'text', a local variable or function whose data type is either **Char** or **Varchar**, or an expression that evaluates to a **Char** or **Varchar** value, such as **Convert** or the string concatenation operator (+).

Some common uses of the **Print** statement are:

```
Print 'Hello Raymond'
Print Convert(Varchar(20), @@CPU_Busy)
Print 'CPU Busy is ' + Convert(Varchar(20), @@CPU_Busy)
```

Raiserror

Another way to return information to the calling program is to use the **Raiserror** statement.

```
Raiserror (<message>, <severity>, <state> [, <argument_list>])
```

Calling **Raiserror** simply sets a system flag to record that an error occurred. Your stored procedure will continue to run normally. The <message> parameter is either a numeric value that refers to a user-defined message in the sysmessages table or a string containing a custom error message. You can choose to allow values to be substituted into <message> by specifying a list of values in <argument_list> and including C style printf formatting commands in <message>.

You must also specify a severity code in <severity>. This value can range from 0 to 18 for normal users and 19 to 25 for users with the sysadmin fixed server role. Severity levels greater than 19 are considered fatal, and they will immediately terminate the connection to the database. Otherwise the exact meaning of <severity> is up to you.

But it's almost fatal: You should use a severity of 19 for non-fatal errors in stored procedures running under the sysadmin fixed server role.

The <state> value provides additional information about the particular error. It can range from 1 to 127. This value has no meaning outside the context of the error message.

Add your own errors: You can add errors to the sysmessages table by calling the sp_addmessage stored procedure. You may delete a message by calling the sp_dropmessage stored procedure.

Which way is right?: In Visual Basic, **RaiseError** is spelled with two E's, while in T-SQL **Raiserror** is spelled with one E.

Go

Technically, **Go** isn't a T-SQL statement, but a command that is used by Query Analyzer and some other query tools to execute the group of T-SQL statements that precede the **Go** command. **Go** must occupy a line by itself in order to be properly recognized.

For instance, the following statements are executed as a single batch:

```
Declare @MDCount Int

Select @MDCount = Count(*)
From Customers
Where State = 'MD'

Declare @SDCount Int

Select @SDCount = Count(*)
From Customers
Where State = 'SD'
```

```
Print 'MD Count = ' + Convert(Varchar(10),@MDCount)
Print 'SD Count = ' + Convert(varchar(10),@SDCount)
```

while these statements are executed as two independent groups:

```
Declare @MDCount Int
Select @MDCount = Count(*)
From Customers
Where State = 'MD'
Print "MD Count = " + Convert(Varchar(10),@MDCount)

Go

Declare @SDCount Int
Select @SDCount = Count(*)
From Customers
Where State = 'SD'
Print 'SD Count = ' + Convert(varchar(10),@SDCount)
```

Finally, these statements will generate an error in the first **Print** statement because the variable `@MDCount` will no longer be in scope.

```
Declare @MDCount Int
Select @MDCount = Count(*)
From Customers
Where State = 'MD'

Go

Declare @SDCount Int
Select @SDCount = Count(*)
From Customers
Where State = 'SD'

Print 'MD Count = ' + Convert(Varchar(10),@MDCount)
Print 'SD Count = ' + convert(varchar(10),@SDCount)
```

Creating and Testing Stored Procedures

While creating and testing a stored procedure isn't very difficult, it is very cumbersome. You have to develop the code using a tool like Query Analyzer, unless you're one of those perfect programmers whose code always runs correctly the first time. Then you have to save the code into the database as part of a **Create Procedure** or **Alter Procedure** statement using either Query Analyzer or Enterprise. Next, you have to build the Visual Basic program to access the stored procedure and verify that it works the way you expect it. Of course it won't, so you may have to revise the code using Query Analyzer and try it over again. Fortunately, Visual Basic includes a sophisticated T-SQL debugger to help you troubleshoot why your stored procedure didn't work as designed.

Creating stored procedures in SQL Server

Stored procedures created by using the **Create Procedure** statement are kept in system tables in your database. The **Create Procedure** statement supplies the name of the stored procedure and the list of parameters associated with it. Following the **As** clause is the list of SQL statements that comprise the stored procedure.

```
Create Procedure <procedure>
[<parameter> <data_type> [= <default>]] ...
As <sql_statement> [<sql_statement>] ...
```

You can use Enterprise Manager to create a stored procedure by following these steps:

1. Expand the icon tree to show the Stored Procedures icon beneath the database where you want to create the stored procedure. Right click on the Stored Procedures icon and select New Stored Procedure to show the Stored Procedure Properties window (see Figure 25-2).

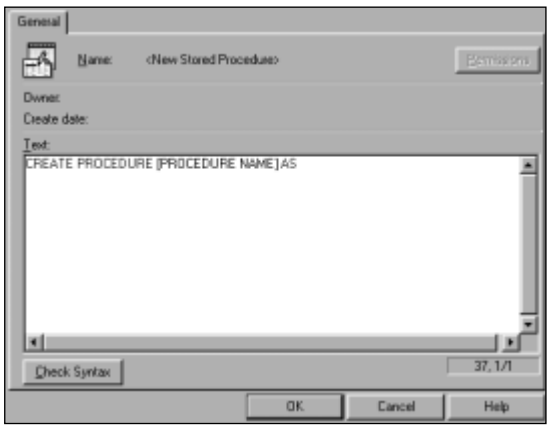


Figure 25-2: Creating a new stored procedure

2. Replace `[PROCEDURE NAME]` with the name you wish to call your stored procedures and include any parameters that belong to the procedure immediately after the name. Then add the body of your stored procedure (see Figure 25-3).
3. Press the Check Syntax button to verify that the syntax is correct. If there's an error, a message box will be displayed containing a short description of the error. Otherwise, a message box saying Syntax check successful! will be displayed.
4. Press OK to save your stored procedure.

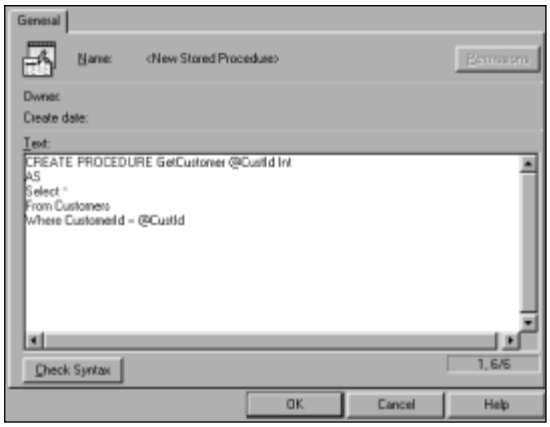


Figure 25-3: Typing your stored procedure into the Properties window

Changes anyone: You can change your stored procedure anytime by right clicking on the stored procedure name in the Details window and choosing Properties from the pop-up menu. The same Properties window you used to create your stored procedure will be displayed, though the **Create Procedure** statement will be changed to **Alter Procedure.**

Another way to do it: You can also execute the same **Create Procedure** statement in Query Analyzer to create a stored procedure.

Rename me twice: If you choose to rename your stored procedure by using the pop-up menu Rename command in Enterprise Manager, you must remember to open the Properties window for the stored procedure and correct its name in the **Alter Procedure** command.

Testing stored procedures in Query Analyzer

Query Analyzer is a general-purpose tool that allows you to run SQL statements interactively. This also includes stored procedures. There are two ways to test your procedure. First you can use the T-SQL debugger to your stored procedure. Second, you can test the block of code standalone Query Analyzer. This involves running the individual statements directly in Query Analyzer and verifying that they do what you want them to. Then you can create the stored procedure and call it using Query Analyzer.

For example, the stored procedure I just wrote can easily be run in Query Analyzer by typing its name followed by a CustomerId value (see Figure 25-4). This is a good way to make sure that you are getting the results you want before you build a Visual Basic program to use it.

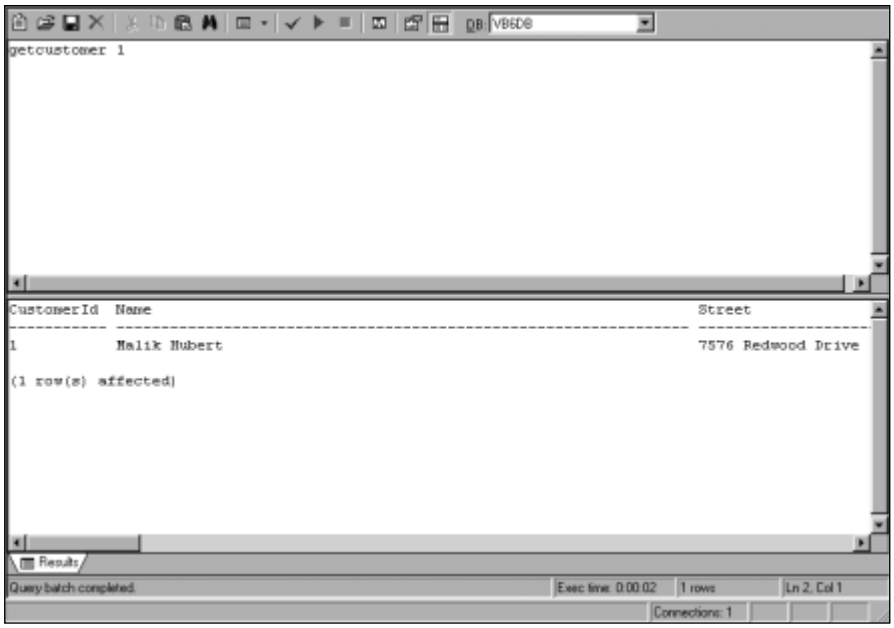


Figure 25-4: Running the stored procedure in Query Analyzer

But VB's better: Remember that Visual Basic includes a comprehensive stored procedure debugger, called the T-SQL Debugger. You can set breakpoints, examine variables and single step through the code. Refer to Chapter 13, "Using Commands and Stored Procedures," for details about how to use this powerful tool.

Thoughts on Stored Procedures in SQL Server

Stored procedures are a powerful tool in SQL Server that can make a big difference in how your application runs. They also are a way to isolate your application from the underlying database system, since you could create a series of stored procedures for each database management system you use.

However, in SQL Server, stored procedures to encapsulate simple SQL statements aren't as critical as they are in other database management systems. SQL Server 7 will remember the SQL statements you have used recently and reuse the compiled code if the query is the same. This makes parameterized queries nearly as efficient as stored procedures and somewhat easier to use.

Summary

In this chapter you learned:

- how stored procedures can improve performance, increase convenience and provide security.
- about the features of Transact-SQL.
- about the statements available in Transact-SQL that help you write stored procedures.
- how to use cursors in Transact-SQL.
- how to process transactions in Transact-SQL.
- hot to create and test stored procedures using Enterprise Manager and Query Analyzer.

✦ ✦ ✦