Chapter 8

Stored Procedures and User-Defined Functions

In This Chapter

- Procedural Extensions
- Stored Procedures
- User-Defined Functions
This chapter introduces batches and routines. A batch is a sequence of Transact-SQL statements and procedural extensions. A routine can be either a stored procedure or a user-defined function (UDF). The beginning of the chapter introduces all procedural extensions supported by the Database Engine. After that, procedural extensions are used, together with Transact-SQL statements, to show how batches can be implemented. A batch can be stored as a database object, as either a stored procedure or a UDF. Some stored procedures are written by users, and others are provided by Microsoft and are referred to as system stored procedures. In contrast to user-defined stored procedures, UDFs return a value to a caller. All routines can be written either in Transact-SQL or in another programming language such as C# or Visual Basic. The end of the chapter introduces table-valued parameters.

**Procedural Extensions**

The preceding chapters introduced Transact-SQL statements that belong to the data definition language and the data manipulation language. Most of these statements can be grouped together to build a batch. As previously mentioned, a batch is a sequence of Transact-SQL statements and procedural extensions that are sent to the database system for execution together. The number of statements in a batch is limited by the size of the compiled batch object. The main advantage of a batch over a group of singleton statements is that executing all statements at once brings significant performance benefits.

There are a number of restrictions concerning the appearance of different Transact-SQL statements inside a batch. The most important is that the data definition statements CREATE VIEW, CREATE PROCEDURE, and CREATE TRIGGER must each be the only statement in a batch.

**NOTE**

To separate DDL statements from one another, use the GO statement.

The following sections describe each procedural extension of the Transact-SQL language separately.

**Block of Statements**

A block allows the building of units with one or more Transact-SQL statements. Every block begins with the BEGIN statement and terminates with the END statement, as shown in the following example:
BEGIN
  statement_1
  statement_2
...
END

A block can be used inside the IF statement to allow the execution of more than one statement, depending on a certain condition (see Example 8.1).

### IF Statement

The Transact-SQL statement IF corresponds to the statement with the same name that is supported by almost all programming languages. IF executes one Transact-SQL statement (or more, enclosed in a block) if a Boolean expression, which follows the keyword IF, evaluates to TRUE. If the IF statement contains an ELSE statement, a second group of statements can be executed if the Boolean expression evaluates to FALSE.

**Note**

Before you start to execute batches, stored procedures, and UDFs in this chapter, please re-create the entire `sample` database.

#### Example 8.1

```sql
USE sample;
IF (SELECT COUNT(*)
    FROM works_on
    WHERE project_no = 'p1'
    GROUP BY project_no ) > 3
PRINT 'The number of employees in the project p1 is 4 or more'
ELSE BEGIN
PRINT 'The following employees work for the project p1'
SELECT emp_fname, emp_lname
FROM employee, works_on
WHERE employee.emp_no = works_on.emp_no
AND project_no = 'p1'
END
```
Example 8.1 shows the use of a block inside the IF statement. The Boolean expression in the IF statement,

\[
\text{(SELECT COUNT(*) FROM works_on WHERE project_no = 'p1' GROUP BY project_no) > 3}
\]

is evaluated to TRUE for the sample database. Therefore, the single PRINT statement in the IF part is executed. Notice that this example uses a subquery to return the number of rows (using the COUNT aggregate function) that satisfy the WHERE condition (project_no='p1'). The result of Example 8.1 is

The number of employees in the project p1 is four or more

**NOTE**

The ELSE part of the IF statement in Example 8.1 contains two statements: PRINT and SELECT. Therefore, the block with the BEGIN and END statements is required to enclose the two statements. (The PRINT statement is another statement that belongs to procedural extensions; it returns a user-defined message.)

### WHILE Statement

The WHILE statement repeatedly executes one Transact-SQL statement (or more, enclosed in a block) while the Boolean expression evaluates to TRUE. In other words, if the expression is true, the statement (or block) is executed, and then the expression is evaluated again to determine if the statement (or block) should be executed again. This process repeats until the expression evaluates to FALSE.

A block within the WHILE statement can optionally contain one of two statements used to control the execution of the statements within the block: BREAK or CONTINUE. The BREAK statement stops the execution of the statements inside the block and starts the execution of the statement immediately following this block. The CONTINUE statement stops only the current execution of the statements in the block and starts the execution of the block from its beginning.

Example 8.2 shows the use of the WHILE statement.

**EXAMPLE 8.2**

```sql
USE sample;
WHILE (SELECT SUM(budget) FROM project) < 500000
BEGIN
    UPDATE project SET budget = budget * 1.1
```
Chapter 8: Stored Procedures and User-Defined Functions

IF (SELECT MAX(budget) 
    FROM project) > 240000 
    BREAK 
ELSE CONTINUE 
END

In Example 8.2, the budget of all projects will be increased by 10 percent until the sum of budgets is greater than $500,000. However, the repeated execution will be stopped if the budget of one of the projects is greater than $240,000. The execution of Example 8.2 gives the following output:

(3 rows affected)
(3 rows affected)
(3 rows affected)

NOTE
If you want to suppress the output, such as that in Example 8.2 (indicating the number of affected rows in SQL statements), use the SET NOCOUNT ON statement.

Local Variables

Local variables are an important procedural extension to the Transact-SQL language. They are used to store values (of any type) within a batch or a routine. They are “local” because they can be referenced only within the same batch in which they were declared. (The Database Engine also supports global variables, which are described in Chapter 4.)

Every local variable in a batch must be defined using the DECLARE statement. (For the syntax of the DECLARE statement, see Example 8.3.) The definition of each variable contains its name and the corresponding data type. Variables are always referenced in a batch using the prefix @. The assignment of a value to a local variable is done

- Using the special form of the SELECT statement
- Using the SET statement
- Directly in the DECLARE statement using the = sign (for instance, @extra_budget MONEY = 1500)

The usage of the first two statements for a value assignment is demonstrated in Example 8.3.
The result is

Budget for p1 increased by @extra_budget

The batch in Example 8.3 calculates the average of all project budgets and compares this value with the budget of project p1. If the latter value is smaller than the calculated value, the budget of project p1 will be increased by the value of the local variable @extra_budget.

**Miscellaneous Procedural Statements**

The procedural extensions of the Transact-SQL language also contain the following statements:

- RETURN
- GOTO
- RAISEERROR
- WAITFOR

The RETURN statement has the same functionality inside a batch as the BREAK statement inside WHILE. This means that the RETURN statement causes the execution of the batch to terminate and the first statement following the end of the batch to begin executing.
Chapter 8: Stored Procedures and User-Defined Functions

The GOTO statement branches to a label, which stands in front of a Transact-SQL statement within a batch. The RAISEERROR statement generates a user-defined error message and sets a system error flag. A user-defined error number must be greater than 50000. (All error numbers <= 50000 are system defined and are reserved by the Database Engine.) The error values are stored in the global variable @@error. (Example 17.3 shows the use of the RAISEERROR statement.)

The WAITFOR statement defines either the time interval (if the DELAY option is used) or a specified time (if the TIME option is used) that the system has to wait before executing the next statement in the batch. The syntax of this statement is

\[
\text{WAITFOR \{DELAY 'time' | TIME 'time' | TIMEOUT 'timeout' \}}
\]

The DELAY option tells the database system to wait until the specified amount of time has passed. TIME specifies a time in one of the acceptable formats for temporal data. TIMEOUT specifies the amount of time, in milliseconds, to wait for a message to arrive in the queue. (Example 13.5 shows the use of the WAITFOR statement.)

Exception Handling with TRY, CATCH, and THROW

Versions of SQL Server previous to SQL Server 2005 required error handling code after every Transact-SQL statement that might produce an error. (You can handle errors using the @@error global variable. Example 13.1 shows the use of this variable.) Starting with SQL Server 2005, you can capture and handle exceptions using two statements, TRY and CATCH. This section first explains what “exception” means and then discusses how these two statements work.

An exception is a problem (usually an error) that prevents the continuation of a program. With such a problem, you cannot continue processing because there is not enough information needed to handle the problem. For this reason, the existing problem will be relegated to another part of the program, which will handle the exception.

The role of the TRY statement is to capture the exception. (Because this process usually comprises several statements, the term “TRY block” typically is used instead of “TRY statement.”) If an exception occurs within the TRY block, the part of the system called the exception handler delivers the exception to the other part of the program, which will handle the exception. This program part is denoted by the keyword CATCH and is therefore called the CATCH block.

**NOTE**

Exception handling using the TRY and CATCH statements is the common way that modern programming languages like C# and Java treat errors.
Exception handling with the TRY and CATCH blocks gives a programmer a lot of benefits, such as:

- Exceptions provide a clean way to check for errors without cluttering code
- Exceptions provide a mechanism to signal errors directly rather than using some side effects
- Exceptions can be seen by the programmer and checked during the compilation process

SQL Server 2012 introduces the third statement in relation to handling errors: THROW. This statement allows you to throw an exception caught in the exception handling block. Simply stated, the THROW statement is another return mechanism, which behaves similarly to the already described RAISEERROR statement.

Example 8.4 shows how exception handling with the TRY/CATCH/THROW works. It shows how you can use exception handling to insert all statements in a batch or to roll back the entire statement group if an error occurs. The example is based on the referential integrity between the department and employee tables. For this reason, you have to create both tables using the PRIMARY KEY and FOREIGN KEY constraints, as done in Example 5.11.

**EXAMPLE 8.4**

```sql
USE sample;
BEGIN TRY
    BEGIN TRANSACTION
    insert into employee values(11111, 'Ann', 'Smith', 'd2');
    insert into employee values(22222, 'Matthew', 'Jones', 'd4'); -- referential integrity error
    insert into employee values(33333, 'John', 'Barrimore', 'd2');
    COMMIT TRANSACTION
    PRINT 'Transaction committed'
END TRY
BEGIN CATCH
    ROLLBACK
    PRINT 'Transaction rolled back';
    THROW
END CATCH
```

After the execution of the batch in Example 8.4, all three statements in the batch won't be executed at all, and the output of this example is
Transaction rolled back

Msg 547, Level 16, State 0, Line 4
The INSERT statement conflicted with the FOREIGN KEY constraint "foreign_emp". The conflict occurred in database "sample", table "dbo.department", column 'dept_no'.

The execution of Example 8.4 works as follows. The first INSERT statement is executed successfully. Then, the second statement causes the referential integrity error. Because all three statements are written inside the TRY block, the exception is “thrown” and the exception handler starts the CATCH block. CATCH rolls back all statements and prints the corresponding message. After that the THROW statement returns the execution of the batch to the caller. For this reason, the content of the employee table won’t change.

NOTE
The statements BEGIN TRANSACTION, COMMIT TRANSACTION, and ROLLBACK are Transact-SQL statements concerning transactions. These statements start, commit, and roll back transactions, respectively. See Chapter 13 for the discussion of these statements and transactions generally.

Example 8.5 shows the batch that supports server-side paging (for the description of server-side paging, see Chapter 6).

**EXAMPLE 8.5**

USE AdventureWorks;
DECLARE
    @PageSize TINYINT = 20,
    @CurrentPage INT = 4;
SELECT BusinessEntityID, JobTitle, BirthDate
FROM HumanResources.Employee
WHERE Gender = 'F'
ORDER BY JobTitle
OFFSET (@PageSize * (@CurrentPage - 1)) ROWS
    FETCH NEXT @PageSize ROWS ONLY;

The batch in Example 8.5 uses the AdventureWorks database and its Employee table to show how generic server-side paging can be implemented. The @PageSize variable is used with the FETCH NEXT statement to specify the number of rows per page (20, in this case). The other variable, @CurrentPage, specifies which particular page should be displayed. In this example, the content of the third page will be displayed.
Stored Procedures

A stored procedure is a special kind of batch written in Transact-SQL, using the SQL language and its procedural extensions. The main difference between a batch and a stored procedure is that the latter is stored as a database object. In other words, stored procedures are saved on the server side to improve the performance and consistency of repetitive tasks.

The Database Engine supports stored procedures and system procedures. Stored procedures are created in the same way as all other database objects—that is, by using the DDL. System procedures are provided with the Database Engine and can be used to access and modify the information in the system catalog. This section describes (user-defined) stored procedures, while system procedures are explained in the next chapter.

When a stored procedure is created, an optional list of parameters can be defined. The procedure accepts the corresponding arguments each time it is invoked. Stored procedures can optionally return a value, which displays the user-defined information or, in the case of an error, the corresponding error message.

A stored procedure is precompiled before it is stored as an object in the database. The precompiled form is stored in the database and used whenever the stored procedure is executed. This property of stored procedures offers an important benefit: the repeated compilation of a procedure is (almost always) eliminated, and the execution performance is therefore increased. This property of stored procedures offers another benefit concerning the volume of data that must be sent to and from the database system. It might take less than 50 bytes to call a stored procedure containing several thousand bytes of statements. The accumulated effect of this savings when multiple users are performing repetitive tasks can be quite significant.

Stored procedures can also be used for the following purposes:

- To control access authorization
- To create an audit trail of activities in database tables

The use of stored procedures provides security control above and beyond the use of the GRANT and REVOKE statements (see Chapter 12), which define different access privileges for a user. This is because the authorization to execute a stored procedure is independent of the authorization to modify the objects that the stored procedure contains, as described in the next section.

Stored procedures that audit write and/or read operations concerning a table are an additional security feature of the database. With the use of such procedures, the database administrator can track modifications made by users or application programs.
Creation and Execution of Stored Procedures

Stored procedures are created with the CREATE PROCEDURE statement, which has the following syntax:

```
CREATE PROCEDURE [schema_name.]proc_name
    ([@param1] type1 [ VARYING] [= default1] [OUTPUT])
    {, …}
    [WITH {RECOMPILE | ENCRYPTION | EXECUTE AS 'user_name'}]
    [FOR REPLICATION]
AS batch | EXTERNAL NAME method_name
```

- `schema_name` is the name of the schema to which the ownership of the created stored procedure is assigned. `proc_name` is the name of the new stored procedure.
- `@param1` is a parameter, while `type1` specifies its data type. The parameter in a stored procedure has the same logical meaning as the local variable for a batch. Parameters are values passed from the caller of the stored procedure and are used within the stored procedure. `default1` specifies the optional default value of the corresponding parameter. (Default can also be NULL.)
- The `OUTPUT` option indicates that the parameter is a return parameter and can be returned to the calling procedure or to the system (see Example 8.9 later in this section).
- As you already know, the precompiled form of a procedure is stored in the database and used whenever the stored procedure is executed. If you want to generate the compiled form each time the procedure is executed, use the WITH RECOMPILE option.

**NOTE**

The use of the WITH RECOMPILE option destroys one of the most important benefits of the stored procedures: the performance advantage gained by a single precompilation. For this reason, the WITH RECOMPILE option should be used only when database objects used by the stored procedure are modified frequently or when the parameters used by the stored procedure are volatile.

The EXECUTE AS clause specifies the security context under which to execute the stored procedure after it is accessed. By specifying the context in which the procedure is executed, you can control which user account the Database Engine uses to validate permissions on objects referenced by the procedure.

By default, only the members of the `sysadmin` fixed server role, and the `db_owner` and `db_ddladmin` fixed database roles, can use the CREATE PROCEDURE statement. However, the members of these roles may assign this privilege to other users...
by using the GRANT CREATE PROCEDURE statement. (For the discussion of user permissions, fixed server roles, and fixed database roles, see Chapter 12.)

Example 8.6 shows the creation of the simple stored procedure for the **project** table.

**EXAMPLE 8.6**

```sql
USE sample;
GO
CREATE PROCEDURE increase_budget (@percent INT=5)
    AS UPDATE project
        SET budget = budget + budget*@percent/100;
```

**NOTE**

The GO statement is used to separate two batches. (The CREATE PROCEDURE statement must be the first statement in the batch.)

The stored procedure **increase_budget** increases the budgets of all projects for a certain percentage value that is defined using the parameter **@percent**. The procedure also defines the default value (5), which is used if there is no argument at the execution time of the procedure.

**NOTE**

It is possible to create stored procedures that reference nonexistent tables. This feature allows you to debug procedure code without creating the underlying tables first, or even connecting to the target server.

In contrast to “base” stored procedures that are placed in the current database, it is possible to create temporary stored procedures that are always placed in the temporary system database called **tempdb**. You might create a temporary stored procedure to avoid executing a particular group of statements repeatedly within a connection. You can create local or global temporary procedures by preceding the procedure name with a single pound sign (**#proc_name**) for local temporary procedures and a double pound sign (**##proc_name**, for example) for global temporary procedures. A local temporary stored procedure can be executed only by the user who created it, and only during the same connection. A global temporary procedure can be executed by all users, but only until the last connection executing it (usually the creator’s) ends.

The life cycle of a stored procedure has two phases: its creation and its execution. Each procedure is created once and executed many times. The EXECUTE statement executes an existing procedure. The execution of a stored procedure is allowed for each
Chapter 8: Stored Procedures and User-Defined Functions

user who either is the owner of or has the EXECUTE privilege for the procedure (see Chapter 12). The EXECUTE statement has the following syntax:

```
[[EXECUTE]] [@return_status =] (proc_name
   | @proc_name_var)
   {[[[@parameter1 =] value | [@parameter1 =] variable [OUTPUT]]] | DEFAULT}..
   [WITH RECOMPILE]
```

All options in the EXECUTE statement, other than `return_status`, have the equivalent logical meaning as the options with the same names in the CREATE PROCEDURE statement. `return_status` is an optional integer variable that stores the return status of a procedure. The value of a parameter can be assigned using either a value (`value`) or a local variable (`@variable`). The order of parameter values is not relevant if they are named, but if they are not named, parameter values must be supplied in the order defined in the CREATE PROCEDURE statement.

The DEFAULT clause supplies the default value of the parameter as defined in the procedure. When the procedure expects a value for a parameter that does not have a defined default and either a parameter is missing or the DEFAULT keyword is specified, an error occurs.

**NOTE**

*When the EXECUTE statement is the first statement in a batch, the word “EXECUTE” can be omitted from the statement. Despite this, it would be safer to include this word in every batch you write.*

Example 8.7 shows the use of the EXECUTE statement.

**EXAMPLE 8.7**

```
USE sample;
EXECUTE increase_budget 10;
```

The EXECUTE statement in Example 8.7 executes the stored procedure `increase_budget` (Example 8.6) and increases the budgets of all projects by 10 percent each.

Example 8.8 shows the creation of a procedure that references the tables `employee` and `works_on`.

**EXAMPLE 8.8**

```
USE sample;
GO
CREATE PROCEDURE modify_empno (@old_no INTEGER, @new_no INTEGER)
   AS UPDATE employee
       SET emp_no = @new_no
```
WHERE emp_no = @old_no
UPDATE works_on
SET emp_no = @new_no
WHERE emp_no = @old_no

The procedure `modify_empno` in Example 8.8 demonstrates the use of stored procedures as part of the maintenance of the referential integrity (in this case, between the `employee` and `works_on` tables). Such a stored procedure can be used inside the definition of a trigger, which actually maintains the referential integrity (see Example 14.3).

Example 8.9 shows the use of the OUTPUT clause.

**Example 8.9**

```sql
USE sample;
GO
CREATE PROCEDURE delete_emp @employee_no INT, @counter INT OUTPUT
AS SELECT @counter = COUNT(*)
    FROM works_on
    WHERE emp_no = @employee_no
DELETE FROM employee
    WHERE emp_no = @employee_no
DELETE FROM works_on
    WHERE emp_no = @employee_no
```

This stored procedure can be executed using the following statements:

```sql
DECLARE @quantity INT
EXECUTE delete_emp @employee_no=28559, @counter=@quantity OUTPUT
```

The preceding example contains the creation of the `delete_emp` procedure as well as its execution. This procedure calculates the number of projects on which the employee (with the employee number `@employee_no`) works. The calculated value is then assigned to the `@counter` parameter. After the deletion of all rows with the assigned employee number from the `employee` and `works_on` tables, the calculated value will be assigned to the `@quantity` variable.

**NOTE**

The value of the parameter will be returned to the calling procedure if the OUTPUT option is used. In Example 8.9, the `delete_emp` procedure passes the `@counter` parameter to the calling statement, so the procedure returns the value to the system. Therefore, the `@counter` parameter must be declared with the OUTPUT option in the procedure as well as in the EXECUTE statement.
The EXECUTE Statement with RESULT SETS Clause

SQL Server 2012 introduces the WITH RESULT SETS clause for the EXECUTE statement. Using this clause, you can change conditionally the form of the result set of a stored procedure.

The following two examples help to explain this clause. Example 8.10 is an introductory example that shows how the output looks when the WITH RESULT SETS clause is omitted.

**EXAMPLE 8.10**

USE sample;
GO
CREATE PROCEDURE employees_in_dept (@dept CHAR(4))
AS SELECT emp_no, emp_lname
    FROM employee
    WHERE dept_no IN (SELECT @dept FROM department
                        GROUP BY dept_no)

*employees_in_dept* is a simple stored procedure that displays the numbers and family names of all employees working for a particular department. (The department number is a parameter of the procedure and must be specified when the procedure is invoked.) The result of this procedure is a table with two columns, named according to the names of the corresponding columns (*emp_no* and *emp_lname*). To change these names (and their data types, too), SQL Server 2012 supports the new WITH RESULTS SETS clause. Example 8.11 shows the use of this clause.

**EXAMPLE 8.11**

USE sample;
EXEC employees_in_dept 'd1'
    WITH RESULT SETS
    ( ([EMPLOYEE NUMBER] INT NOT NULL,
        [NAME OF EMPLOYEE] CHAR(20) NOT NULL));

The output is

<table>
<thead>
<tr>
<th>EMPLOYEE NUMBER</th>
<th>NAME OF EMPLOYEE</th>
</tr>
</thead>
<tbody>
<tr>
<td>18316</td>
<td>Barrimore</td>
</tr>
<tr>
<td>28559</td>
<td>Moser</td>
</tr>
</tbody>
</table>
As you can see, the WITH RESULT SETS clause in Example 8.11 allows you to change the name and data types of columns displayed in the result set. Therefore, this new functionality gives you the flexibility to execute stored procedures and place the output result sets into a new table.

Changing the Structure of Stored Procedures

The Database Engine also supports the ALTER PROCEDURE statement, which modifies the structure of a stored procedure. The ALTER PROCEDURE statement is usually used to modify Transact-SQL statements inside a procedure. All options of the ALTER PROCEDURE statement correspond to the options with the same name in the CREATE PROCEDURE statement. The main purpose of this statement is to avoid reassignment of existing privileges for the stored procedure.

NOTE

The Database Engine supports the CURSOR data type. You use this data type to declare cursors inside a stored procedure. A cursor is a programming construct that is used to store the output of a query (usually a set of rows) and to allow end-user applications to display the rows record by record. A detailed discussion of cursors is outside of the scope of this book.

A stored procedure (or a group of stored procedures with the same name) is removed using the DROP PROCEDURE statement. Only the owner of the stored procedure and the members of the db_owner and sysadmin fixed roles can remove the procedure.

Stored Procedures and CLR

SQL Server supports the Common Language Runtime (CLR), which allows you to develop different database objects (stored procedures, user-defined functions, triggers, user-defined aggregates, and user-defined types) using C# and Visual Basic. CLR also allows you to execute these database objects using the common run-time system.

NOTE

You enable and disable the use of CLR through the clr_enabled option of the sp_configure system procedure. Execute the RECONFIGURE statement to update the running configuration value.

Example 8.12 shows how you can enable the use of CLR with the sp_configure system procedure.
Example 8.12

USE sample;
EXEC sp_configure 'clr_enabled', 1
RECONFIGURE

To implement, compile, and store procedures using CLR, you have to execute the following four steps in the given order:

1. Implement a stored procedure using C# (or Visual Basic) and compile the program, using the corresponding compiler.
2. Use the CREATE ASSEMBLY statement to create the corresponding executable file.
3. Store the procedure as a server object using the CREATE PROCEDURE statement.
4. Execute the procedure using the EXECUTE statement.

Figure 8-1 shows how CLR works. You use a development environment such as Visual Studio to implement your program. After the implementation, start the C# or
Visual Basic compiler to generate the object code. This code will be stored in a .dll file, which is the source for the CREATE ASSEMBLY statement. After the execution of this statement, you get the intermediate code. In the next step you use the CREATE PROCEDURE statement to store the executable as a database object. Finally, the stored procedure can be executed using the already-introduced EXECUTE statement.

Examples 8.13 through 8.17 demonstrate the whole process just described. Example 8.13 shows the C# program that will be used to demonstrate how you apply CLR to implement and deploy stored procedures.

**Example 8.13**

```csharp
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
using Microsoft.SqlServer.Server;

public partial class StoredProcedures {
    [SqlProcedure]
    public static int GetEmployeeCount()
    {
        int iRows;
        SqlConnection conn = new SqlConnection("Context Connection=true");
        conn.Open();
        SqlCommand sqlCmd = conn.CreateCommand();
        sqlCmd.CommandText = "select count(*) as 'Employee Count' " + "from employee";
        iRows = (int)sqlCmd.ExecuteScalar();
        conn.Close();
        return iRows;
    }
}
```

This program uses a query to calculate the number of rows in the employee table. The `using` directives at the beginning of the program specify namespaces, such as System.Data. These directives allow you to specify class names in the source program without referencing the corresponding namespace. The StoredProcedures class is then defined, which is written with a `[SqlProcedure]` attribute. This attribute tells the compiler that the class is a stored procedure. Inside that class is defined a method called GetEmployeeCount(). The connection to the database system is established using the `conn` instance of the SqlConnection class. The `Open()` method is applied to that
instance to open the connection. The `CreateCommand()` method, applied to `conn`, allows you to access the `SqlCommand` instance called `sqlCmd`.

The following lines of code

```csharp
sqlCmd.CommandText = "select count(*) as 'Employee Count' " + "from employee;"
iRows = (int)sqlCmd.ExecuteScalar();
```

use the SELECT statement to find the number of rows in the `employee` table and to display the result. The command text is specified by setting the `CommandText` property of the `SqlCmd` instance returned by the call to the `CreateCommand()` method. Next, the `ExecuteScalar()` method of the `SqlCommand` instance is called. This returns a scalar value, which is finally converted to the `int` data type and assigned to the `iRows` variable.

Example 8.14 shows the first step in deploying stored procedures using CLR.

**EXAMPLE 8.14**

csc /target:library GetEmployeeCount.cs

```
/reference:"C:\Program Files\Microsoft SQL Server\MSSQL11.MSSQLSERVER\MSSQL\Binn\sqlaccess.dll"
```

Example 8.14 demonstrates how to compile the C# method called `GetEmployeeCount()` (Example 8.13). (Actually, this command can be used generally to compile any C# program, if you set the appropriate name for the source program.) `csc` is the command that is used to invoke the C# compiler. You invoke the `csc` command at the Windows command line. Before starting the command, you have to specify the location of the compiler using the `PATH` environment variable. At the time of writing this book, the C# compiler (the `csc.exe` file) can be found in the `C:\WINDOWS\Microsoft.NET\Framework` directory. (You should select the appropriate version of the compiler.)

The `/target` option specifies the name of the C# program, while the `/reference` option defines the .dll file, which is necessary for the compilation process.

Example 8.15 shows the next step in creating the stored procedure. (Before you execute this example, copy the existing .dll file to the root of the C: drive.)

**EXAMPLE 8.15**

```sql
USE sample;
GO
CREATE ASSEMBLY GetEmployeeCount
FROM 'C:\GetEmployeeCount.dll' WITH PERMISSION_SET = SAFE
```

The CREATE ASSEMBLY statement uses the managed code as the source to create the corresponding object, against which CLR stored procedures, UDFs, and triggers can be created. This statement has the following syntax:

```sql
CREATE ASSEMBLY assembly_name [ AUTHORIZATION owner_name ]
FROM { dll_file}
[WITH PERMISSION_SET = { SAFE | EXTERNAL_ACCESS | UNSAFE }]
```

`assembly_name` is the name of the assembly. The optional AUTHORIZATION clause specifies the name of a role as owner of the assembly. The FROM clause specifies the path where the assembly being uploaded is located. (Example 8.15 copies the .dll file generated from the source program from the Framework directory to the root of the C: drive.)

The WITH PERMISSION_SET clause is a very important clause of the CREATE ASSEMBLY statement and should always be set. It specifies a set of code access permissions granted to the assembly. SAFE is the most restrictive permission set. Code executed by an assembly with this permission cannot access external system resources, such as files. EXTERNAL_ACCESS allows assemblies to access certain external system resources, while UNSAFE allows unrestricted access to resources, both within and outside the database system.

**NOTE**

In order to store the information concerning assembly code, a user must have the ability to execute the CREATE ASSEMBLY statement. The user (or role) executing the statement is the owner of the assembly. It is possible to assign an assembly to another user by using the AUTHORIZATION clause of the CREATE SCHEMA statement.

The Database Engine also supports the ALTER ASSEMBLY and DROP ASSEMBLY statements. You can use the ALTER ASSEMBLY statement to refresh the system catalog to the latest copy of .NET modules holding its implementation. This statement also adds or removes files associated with the corresponding assembly. The DROP ASSEMBLY statement removes the specified assembly and all its associated files from the current database.

Example 8.16 creates the stored procedures based on the managed code implemented in Example 8.13.

**EXAMPLE 8.16**

```sql
USE sample;
GO
CREATE PROCEDURE GetEmployeeCount
AS EXTERNAL NAME GetEmployeeCount.StoredProcedures.GetEmployeeCount
```
Chapter 8: Stored Procedures and User-Defined Functions

The CREATE PROCEDURE statement in Example 8.16 is different from the same statement used in Examples 8.6 and 8.8, because it contains the EXTERNAL NAME option. This option specifies that the code is generated using CLR. The name in this clause is a three-part name:

\[
\text{assembly\_name.class\_name.method\_name}
\]

- \text{assembly\_name} is the name of the assembly (see Example 8.15).
- \text{class\_name} is the name of the public class (see Example 8.13).
- \text{method\_name}, which is optional, is the name of the method, which is specified inside the class.

Example 8.17 is used to execute the \text{GetEmployeeCount} procedure.

\textbf{Example 8.17}

```sql
USE sample;
DECLARE @ret INT
EXECUTE @ret=GetEmployeeCount
PRINT @ret
```

The PRINT statement returns the current number of the rows in the \text{employee} table.

\textbf{User-Defined Functions}

In programming languages, there are generally two types of routines:

- Stored procedures
- User-defined functions (UDFs)

As discussed in the previous major section of this chapter, stored procedures are made up of several statements that have zero or more input parameters but usually do not return any output parameters. In contrast, functions always have one return value. This section describes the creation and use of UDFs.
Creation and Execution of User-Defined Functions

UDFs are created with the CREATE FUNCTION statement, which has the following syntax:

```
CREATE FUNCTION [schema_name.]function_name
    [([@param] type [= default]) {,...}
     RETURNS {scalar_type | [@variable] TABLE}
     [WITH {ENCRYPTION | SCHEMABINDING}
     [AS] (block | RETURN (select_statement))]
```

- `schema_name` is the name of the schema to which the ownership of the created UDF is assigned. `function_name` is the name of the new function. `@param` is an input parameter, while `type` specifies its data type. Parameters are values passed from the caller of the UDF and are used within the function. `default` specifies the optional default value of the corresponding parameter. (Default can also be NULL.)

The RETURNS clause defines a data type of the value returned by the UDF. This data type can be any of the standard data types supported by the database system, including the TABLE data type. (The only standard data type that you cannot use is TIMESTAMP.)

UDFs are either scalar-valued or table-valued. A scalar-valued function returns an atomic (scalar) value. This means that in the RETURNS clause of a scalar-valued function, you specify one of the standard data types. Functions are table-valued if the RETURNS clause returns a set of rows (see the next subsection).

The WITH ENCRYPTION option encrypts the information in the system catalog that contains the text of the CREATE FUNCTION statement. In that case, you cannot view the text used to create the function. (Use this option to enhance the security of your database system.)

The alternative clause, WITH SCHEMABINDING, binds the UDF to the database objects that it references. Any attempt to modify the structure of the database object that the function references fails. (The binding of the function to the database objects it references is removed only when the function is altered, so the SCHEMABINDING option is no longer specified.)

Database objects that are referenced by a function must fulfill the following conditions if you want to use the SCHEMABINDING clause during the creation of that function:

- All views and UDFs referenced by the function must be schema-bound.
- All database objects (tables, views, or UDFs) must be in the same database as the function.
**block** is the BEGIN/END block that contains the implementation of the function. The final statement of the block must be a RETURN statement with an argument. (The value of the argument is the value returned by the function.) In the body of a BEGIN/END block, only the following statements are allowed:

- Assignment statements such as SET
- Control-of-flow statements such as WHILE and IF
- DECLARE statements defining local data variables
- SELECT statements containing SELECT lists with expressions that assign to variables that are local to the function
- INSERT, UPDATE, and DELETE statements modifying variables of the TABLE data type that are local to the function

By default, only the members of the *sysadmin* fixed server role and the *db_owner* and *db_ddladmin* fixed database roles can use the CREATE FUNCTION statement. However, the members of these roles may assign this privilege to other users by using the GRANT CREATE FUNCTION statement (see Chapter 12).

Example 8.18 shows the creation of the function called *compute_costs*.

**EXAMPLE 8.18**

```sql
-- This function computes additional total costs that arise
-- if budgets of projects increase
USE sample;
GO
CREATE FUNCTION compute_costs (@percent INT =10) 
  RETURNS DECIMAL(16,2) 
BEGIN 
  DECLARE @additional_costs DEC (14,2), @sum_budget dec(16,2) 
  SELECT @sum_budget = SUM (budget) FROM project 
  SET @additional_costs = @sum_budget * @percent/100 
  RETURN @additional_costs 
END
```

The function *compute_costs* computes additional costs that arise when all budgets of projects increase. The single input variable, *@percent*, specifies the percentage of increase of budgets. The BEGIN/END block first declares two local variables: *@additional_costs* and *@sum_budget*. The function then assigns to *@sum_budget* the sum of all budgets, using the SELECT statement. After that, the function computes total additional costs and returns this value using the RETURN statement.
Invoking User-Defined Functions

Each UDF can be invoked in Transact-SQL statements, such as SELECT, INSERT, UPDATE, or DELETE. To invoke a function, specify the name of it, followed by parentheses. Within the parentheses, you can specify one or more arguments. Arguments are values or expressions that are passed to the input parameters that are defined immediately after the function name. When you invoke a function, and all parameters have no default values, you must supply argument values for all of the parameters and you must specify the argument values in the same sequence in which the parameters are defined in the CREATE FUNCTION statement.

Example 8.19 shows the use of the compute_costs function (Example 8.18) in a SELECT statement.

**Example 8.19**

```
USE sample;
SELECT project_no, project_name 
FROM project 
WHERE budget < dbo.compute_costs(25)
```

The result is

<table>
<thead>
<tr>
<th>project_no</th>
<th>project_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>p2</td>
<td>Gemini</td>
</tr>
</tbody>
</table>

The SELECT statement in Example 8.19 displays names and numbers of all projects where the budget is lower than the total additional costs of all projects for a given percentage.

**NOTE**

Each function used in a Transact-SQL statement must be specified using its two-part name—that is, *schema_name.function_name*.

Table-Valued Functions

As you already know, functions are table-valued if the RETURNS clause returns a set of rows. Depending on how the body of the function is defined, table-valued functions can be classified as inline or multistatement functions. If the RETURNS clause specifies TABLE with no accompanying list of columns, the function is an inline function. Inline functions return the result set of a SELECT statement as a variable.
Chapter 8: Stored Procedures and User-Defined Functions

of the TABLE data type (see Example 8.20). A multistatement table-valued function includes a name followed by TABLE. (The name defines an internal variable of the type TABLE.) You can use this variable to insert rows into it and then return the variable as the return value of the function.

Example 8.20 shows a function that returns a variable of the TABLE data type.

**EXAMPLE 8.20**

USE sample;
GO
CREATE FUNCTION employees_in_project (@pr_number CHAR(4))
RETURNS TABLE
AS RETURN (SELECT emp_fname, emp_lname
    FROM works_on, employee
    WHERE employee.emp_no = works_on.emp_no
    AND project_no = @pr_number)

The `employees_in_project` function is used to display names of all employees that belong to a particular project. The input parameter `@pr_number` specifies a project number. While the function generally returns a set of rows, the RETURNS clause contains the TABLE data type. (Note that the BEGIN/END block in Example 8.20 must be omitted, while the RETURN clause contains a SELECT statement.)

Example 8.21 shows the use of the `employees_in_project` function.

**EXAMPLE 8.21**

USE sample;
SELECT *
FROM employees_in_project('p3')

The result is

<table>
<thead>
<tr>
<th>emp_fname</th>
<th>emp_lname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ann</td>
<td>Jones</td>
</tr>
<tr>
<td>Elsa</td>
<td>Bertoni</td>
</tr>
<tr>
<td>Elke</td>
<td>Hansel</td>
</tr>
</tbody>
</table>

Table-Valued Functions and APPLY

The APPLY operator is a relational operator that allows you to invoke a table-valued function for each row of a table expression. This operator is specified in the FROM
clause of the corresponding SELECT statement in the same way as the JOIN operator is applied. There are two forms of the APPLY operator:

- **CROSS APPLY**
- **OUTER APPLY**

The CROSS APPLY operator returns those rows from the inner (left) table expression that match rows in the outer (right) table expression. Therefore, the CROSS APPLY operator is logically the same as the INNER JOIN operator.

The OUTER APPLY operator returns all the rows from the inner (left) table expression. (For the rows for which there are no corresponding matches in the outer table expression, it contains NULL values in columns of the outer table expression.) OUTER APPLY is logically equivalent to LEFT OUTER JOIN.

Examples 8.22 and 8.23 show how you can use APPLY.

**EXAMPLE 8.22**

```sql
-- generate function
CREATE FUNCTION dbo.fn_getjob(@empid AS INT)
    RETURNS TABLE AS
RETURN
    SELECT job
    FROM works_on
    WHERE emp_no = @empid
    AND job IS NOT NULL AND project_no = 'p1';
```

The `fn_getjob()` function in Example 8.22 returns the set of rows from the `works_on` table. This result set is “joined” in Example 8.23 with the content of the `employee` table.

**EXAMPLE 8.23**

```sql
-- use CROSS APPLY
SELECT E.emp_no, emp_fname, emp_lname, job
FROM employee as E
    CROSS APPLY dbo.fn_getjob(E.emp_no) AS A

-- use OUTER APPLY
SELECT E.emp_no, emp_fname, emp_lname, job
FROM employee as E
    OUTER APPLY dbo.fn_getjob(E.emp_no) AS A
```
The result is

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_fname</th>
<th>emp_lname</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>10102</td>
<td>Ann</td>
<td>Jones</td>
<td>Analyst</td>
</tr>
<tr>
<td>29346</td>
<td>James</td>
<td>James</td>
<td>Clerk</td>
</tr>
<tr>
<td>9031</td>
<td>Elsa</td>
<td>Bertoni</td>
<td>Manager</td>
</tr>
<tr>
<td>28559</td>
<td>Sybill</td>
<td>Moser</td>
<td>NULL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>emp_no</th>
<th>emp_fname</th>
<th>emp_lname</th>
<th>job</th>
</tr>
</thead>
<tbody>
<tr>
<td>25348</td>
<td>Matthew</td>
<td>Smith</td>
<td>NULL</td>
</tr>
<tr>
<td>10102</td>
<td>Ann</td>
<td>Jones</td>
<td>Analyst</td>
</tr>
<tr>
<td>18316</td>
<td>John</td>
<td>Barimore</td>
<td>NULL</td>
</tr>
<tr>
<td>29346</td>
<td>James</td>
<td>James</td>
<td>Clerk</td>
</tr>
<tr>
<td>9031</td>
<td>Elsa</td>
<td>Bertoni</td>
<td>Manager</td>
</tr>
<tr>
<td>2581</td>
<td>Elke</td>
<td>Hansel</td>
<td>NULL</td>
</tr>
<tr>
<td>28559</td>
<td>Sybill</td>
<td>Moser</td>
<td>NULL</td>
</tr>
</tbody>
</table>

In the first query of Example 8.23, the result set of the table-valued function `fn_getjob()` is “joined” with the content of the `employee` table using the CROSS APPLY operator. `fn_getjob()` acts as the right input, and the `employee` table acts as the left input. The right input is evaluated for each row from the left input, and the rows produced are combined for the final output.

The second query is similar to the first one, but uses OUTER APPLY, which corresponds to the outer join operation of two tables.

### Table-Valued Parameters

In all versions previous to SQL Server 2008, it was difficult to send many parameters to a routine. In that case you had to use a temporary table, insert the values into it, and then call the routine. Since SQL Server 2008, you can use table-valued parameters to simplify this task. These parameters are used to deliver a result set to the corresponding routine.

Example 8.24 shows the use of a table-valued parameter.

**EXAMPLE 8.24**

```sql
USE sample;
GO
CREATE TYPE departmentType AS TABLE
```
Example 8.24 first defines the type called departmentType as a table. This means that its type is the TABLE data type, so rows can be inserted in it. In the insertProc procedure, the @Dallas variable, which is of the departmentType type, is specified. (The READONLY clause specifies that the content of the table variable cannot be modified.) In the subsequent batch, data is added to the table variable, and after that the procedure is executed. The procedure, when executed, inserts rows from the table variable into the temporary table #dallasTable. The content of the temporary table is as follows:

<table>
<thead>
<tr>
<th>dept_no</th>
<th>dept_name</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>Research</td>
<td>Dallas</td>
</tr>
<tr>
<td>d3</td>
<td>Marketing</td>
<td>Dallas</td>
</tr>
</tbody>
</table>

The use of table-valued parameters gives you the following benefits:

- It simplifies the programming model in relation to routines.
- It reduces the round trips to the server.
- The resulting table can have different numbers of rows.
Chapter 8: Stored Procedures and User-Defined Functions

Changing the Structure of UDFs

The Transact-SQL language also supports the ALTER FUNCTION statement, which modifies the structure of a UDF. This statement is usually used to remove the schema binding. All options of the ALTER FUNCTION statement correspond to the options with the same name in the CREATE FUNCTION statement.

A UDF is removed using the DROP FUNCTION statement. Only the owner of the function (or the members of the db_owner and sysadmin fixed database roles) can remove the function.

User-Defined Functions and CLR

The discussion in “Stored Procedures and CLR” earlier in the chapter is also valid for UDFs. The only difference is that you use the CREATE FUNCTION statement (instead of CREATE PROCEDURE) to store a UDF as a database object. Also, UDFs are used in a different context from that of stored procedures, because UDFs always have a return value.

Example 8.25 shows the C# program used to demonstrate how UDFs are compiled and deployed.

**EXAMPLE 8.25**

```csharp
using System;
using System.Data.Sql;
using System.Data.SqlTypes;
public class budgetPercent
{
    private const float percent = 10;
    public static SqlDouble computeBudget(float budget)
    {
        float budgetNew;
        budgetNew = budget * percent;
        return budgetNew;
    }
}
```

The C# source program in Example 8.25 shows a UDF that calculates the new budget of a project using the old budget and the percentage increase. (The description of the C# program is omitted because this program is analog to the program in Example 8.13.) Example 8.26 shows the CREATE ASSEMBLY statement, which is necessary if you want to create a database object.
EXAMPLE 8.26

USE sample;
GO
CREATE ASSEMBLY computeBudget
FROM 'C:\computeBudget.dll'
WITH PERMISSION_SET = SAFE

The CREATE FUNCTION statement in Example 8.27 stores the `computeBudget` assembly as the database object, which can be used subsequently in data manipulation statements, such as SELECT, as shown in Example 8.28.

EXAMPLE 8.27

USE sample;
GO
CREATE FUNCTION ReturncomputeBudget (@budget Real)
RETURNS FLOAT
AS EXTERNAL NAME computeBudget.budgetPercent.computeBudget

EXAMPLE 8.28

USE sample;
SELECT dbo.ReturncomputeBudget (321.50)

The result is 3215.

NOTE

You can invoke an existing UDF at several places inside a SELECT statement. Example 8.19 shows its use with the WHERE clause, Example 8.21 in the FROM clause, and Example 8.28 in the SELECT list.

Summary

A stored procedure is a special kind of batch, written either in the Transact-SQL language or using the Common Language Runtime (CLR). Stored procedures are used for the following purposes:

- To control access authorization
- To create an audit trail of activities in database tables
Chapter 8: Stored Procedures and User-Defined Functions

- To enforce consistency and business rules with respect to data modification
- To improve the performance of repetitive tasks

User-defined functions have a lot in common with stored procedures. The main difference is that UDFs do not support parameters but return a single data value, which can also be a table.

Microsoft suggests using Transact-SQL as the default language for creating server-side objects. (CLR is recommended as an alternative only when your program contains a lot of computation.)

The next chapter discusses the system catalog of the Database Engine.

Exercises

E.8.1
Create a batch that inserts 3000 rows in the `employee` table. The values of the `emp_no` column should be unique and between 1 and 3000. All values of the columns `emp_lname`, `emp_fname`, and `dept_no` should be set to 'Jane', 'Smith', and 'd1', respectively.

E.8.2
Modify the batch from E.8.1 so that the values of the `emp_no` column should be generated randomly using the RAND function. (Hint: Use the temporal system functions DATEPART and GETDATE to generate the random values.)