User-Defined Functions

User-defined functions (UDFs) can be used to ensure data integrity, and are (in my opinion) one of the best additions made to SQL Server in quite some time.

Basically, UDFs are closely related to stored procedures, but instead of being a resource primarily intended to modify data and return results, they are T-SQL routines that return a value. As you'll see, the SQL Server compiler won't allow you to execute any data modification code from within the UDF, and as such UDFs can only be used for read-only operations. The value returned by a UDF can either be a simple scalar or a table or result set.

Basic Code Structure

What makes UDFs fantastic is that they can be embedded into queries, thereby reducing your need for huge queries with many lines of repeating code. A single function can hold routines that can be called from the query at will. UDFs solve problems that in earlier versions of SQL Server were either impossible or very difficult to implement. UDFs have the following basic structure:

```
CREATE FUNCTION [<ownerName>.]<functionName>
(
   [<parameters>] --just like in stored procedures
)
RETURNS <data type>
AS
BEGIN
   [<statements>]
   RETURN <expression>
END
```

When creating these functions, as opposed to stored procedures, there are a few very important differences in syntax that you need to remember:

- You use `CREATE FUNCTION` instead of `CREATE PROCEDURE`.
- Parentheses are always required regardless of whether there are parameters.
- `RETURN` clauses are used to declare what data type will be returned. Note that the `RETURN` statement is *mandatory* for a UDF, unlike in the case of a stored procedure.
- `BEGIN` and `END` are required for scalar and multistatement table valued functions.

There are three distinct versions of UDFs, which I'll show you in turn.

Scalar Functions

Scalar functions return a single value. They can be used in a variety of places to implement simple or complex functionality not normally found in SQL Server.

The first example of a scalar function is one of the simplest possible examples: incrementing an integer value. The function simply takes a single integer parameter, increments it by one, and then returns this value:

```
CREATE FUNCTION integer$increment
(
   @integerVal int
)
RETURNS int AS
BEGIN
   SET @integerVal = @integerVal + 1
   RETURN @integerVal
END
GO
```

Note that I used the dollar sign naming system for functions. I name all functions, triggers, and stored procedures in this manner, to separate the object (in this case integer) from the method (in this case increment). This is simply a personal preference.

You can execute this using the classic execute syntax from previous versions of SQL Server:

```
DECLARE @integer int
EXEC @integer = integer$increment @integerVal = 1
SELECT @integer AS value
```

which will return

```
value
-----------
2
```

Now that you've created a function and executed it, it's logical that you would want to call it from a `SELECT` statement. However, this is trickier than it might at first appear. If you try the following statement:

```
SELECT integer$increment (1) AS value
```

Then you get the following message:

```
Server: Msg 195, Level 15, State 10, Line 1
'integer$increment' is not a recognized function name.
```

UDFs require the specification of owner in order to successfully execute.

```
SELECT dbo.integer$increment (1) AS value
```

You now get your desired result:

```
value
-----------
2
```

There is one other usage note that I have to mention. It isn't allowed to pass user-named parameters to functions as can be done with stored procedures. So the following two methods of calling the code won't work:

```
SELECT dbo.integer$increment @integerVal = 1 AS value
SELECT dbo.integer$increment (@integerVal = 1) AS value
```

Executing them will return

```
Server: Msg 170, Level 15, State 1, Line 1
Line 1: Incorrect syntax near '@integerVal'.

Server: Msg 137, Level 15, State 1, Line 3
Must declare the variable '@integerVal'.
```

Unfortunately, only the first error message makes any sense (the second one is trying to pass the Boolean value of (`@integerValue = 1`) to the function. Strangely, you can only use the `@parmname =` syntax with the `EXEC` statement, as you saw an example of earlier.

As a final example involving the incrementing function, you'll see how to increment a value three times, all in the same statement (note the nesting of calls):

```
SELECT dbo.integer$increment(dbo.integer$increment(dbo.integer$increment (1)))
➔ AS value
```

which as expected will return

```
value
-----------
4
```

Let's now look at a more complex example of a function that I'm sure many readers would have wanted to implement at one time: a function to take a string and capitalize the first letter of each word within it:

```
CREATE FUNCTION string$properCase
(
   @inputString varchar(2000)
)
RETURNS varchar(2000) AS
BEGIN
   -- set the whole string to lower
   SET @inputString = LOWER(@inputstring)
   -- then use stuff to replace the first character
   SET @inputString =
   --STUFF in the uppercased character in to the next character,
   --replacing the lowercased letter
   STUFF(@inputString,1,1,UPPER(SUBSTRING(@inputString,1,1)))

   --@i is for the loop counter, initialized to 2
   DECLARE @i int
   SET @i = 1

   --loop from the second character to the end of the string
   WHILE @i < LEN(@inputString)
   BEGIN
      --if the character is a space
      IF SUBSTRING(@inputString,@i,1) = ' '
      BEGIN
         --STUFF in the uppercased character into the next character
         SET @inputString = STUFF(@inputString,@i +
         1,1,UPPER(SUBSTRING(@inputString,@i + 1,1)))
      END
      --increment the loop counter
      SET @i = @i + 1
   END
   RETURN @inputString
END

GO
```

So you can then execute the following statement:

```
SELECT name, dbo.string$properCase(name) AS artistProper
FROM artist
```

and get back the name as it was, and the name as it is now.

```
name                 artist
-------------------- ----------------------
the beatles          The Beatles
the who              The Who
```

Although this is certainly not the perfect function for changing a string into a title case, you could easily extend this to take care of all of the "special" cases that arise (like McCartney, or MacDougall).

The last example I'll show you accesses the album table, counts the number of records in the table, and returns the number. What is nice about this function is that the caller won't be aware that the table is being accessed, because this is done from within the UDF. If you create the following function:

```
CREATE FUNCTION album$recordCount
(
   @artistId int
)
RETURNS int AS
BEGIN
   DECLARE @count AS int -- variable to hold output

   --count the number of rows in the table
   SELECT @count = COUNT(*)
   FROM album
   WHERE artistId = @artistId

   RETURN @count
END
GO
```

then you can execute it using a single `SELECT` statement:

```
SELECT dbo.album$recordCount(1) AS albumCount
```

which will return

```
albumCount
-----------
2
```

However, if you were to execute it like this:

```
SELECT name,
   artistId,
   dbo.album$recordCount(artistId) AS albumCount
FROM album
```

it would return

```
ArtistId    albumCount
----------- -----------------------
1           2
2           1
```

The UDF will execute one time for each row. This isn't a problem for such a small query, but performance will become an issue when you have bigger numbers of large tables.

Inline Table-Valued Functions

An inline table-valued function, also known as a single-statement table-valued function, is a more complex version of a scalar function, because it returns a table rather than a single value. In the following code example, you build a function that returns a table of album keys, filtered out by artist.

> *I should note in passing that I'll refer to a function returning a table, whereas a result set comes from a procedure.*

```
CREATE FUNCTION album$returnKeysByArtist
(
    @artistId int
)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN (
    SELECT albumId
    FROM dbo.album
    WHERE artistId = @artistId )
```

Now, you can execute this function. But how? This is totally different from any resource you have had in SQL Server before now. You can't execute it like a procedure:

```
EXEC dbo.album$returnKeysbyArtist @artistId = 1
```

Doing it this way returns an error:

```
Server: Msg 2809, Level 18, State 1, Line 1
The request for procedure 'album$returnKeysByArtist' failed because 'album$returnKeysByArtist' is
a function object.
```

What you *can* do, however, is substitute the call to the table-valued function in the place of a table in any query. You therefore execute the function like this:

```
SELECT album.albumId, album.name
FROM album
   JOIN dbo.album$returnKeysbyArtist(1) AS functionTest
   ON album.albumId = functionTest.albumId
```

which produces the following:

```
albumId     name
----------- ----------------------------
1           the white album
2           revolver
```

Is this better than a normal join? Not really. This isn't a standard database programming operation, and it would likely hinder the query optimizer from producing the most optimized code. However, it's a pretty cool way to build a custom piece of code that you can add to your toolbox if you have very small sets of data that you wish to use in your queries.

Schema Binding

In the declaration of your function, you flew past something that might be an unfamiliar term:

```
WITH SCHEMABINDING
```

Schema binding is a new feature that SQL Server provides for functions and views, allowing the taking of items that the view is dependent upon (such as tables or views) and "binding" them to that view. In this way, nobody can make alterations to those objects unless the schema bound view is dropped. This addresses a very common problem that has occurred since the first days of client-server programming. What if the server changes? How will the client deal with the change?

For example, in your existing function, you've used the album table. If you wanted to drop the album table:

```
DROP TABLE album
```

you get the following error:

```
Server: Msg 3729, Level 16, State 1, Line 2
Could not DROP TABLE 'album'. It is being referenced by object 'album$returnKeysbyArtist'.
```

Note that this is not a bad thing. Indeed, this demonstrates SQL Server's inbuilt mechanism for protecting its data.

So what if you wanted to simply alter the table? The UDF will disallow any attempt to alter a used column, although you *will* be allowed to either add a column or modify one that is unused. To illustrate this, I'll show you how to add a `description` column and then remove it to leave the table as it was:

```
ALTER TABLE album
ADD description varchar(100) NULL
ALTER TABLE album
DROP COLUMN description
```

which executes just fine. But if you try to drop a column that is accessed by a function, such as the `artistId`:

```
ALTER TABLE album
DROP COLUMN artistId
```

SQL Server knows that this column is being used by a function and tells you so:

```
Server: Msg 5074, Level 16, State 3, Line 1
The object 'album$returnKeysByArtist' is dependent on column 'artistId'.
ALTER TABLE DROP COLUMN artistId failed because one or more objects access this column.
```

A best practice would be to always use schema binding on a production system that clients will be accessing. Obviously if it will be needed to drop and re-create a table frequently, then schema binding might not be much of an advantage, but most tables in a deployed database will rarely, if ever, change, and if they do, it's always wise to know the code that accesses them. There will be no performance issues other than the speed with which the schema bound object can be modified.

If the need arises to modify a table or column, either

- Turn off `SCHEMABINDING`, carry out the required modifications, and turn it back on again, or
- Drop the function, modify the table or column(s), and then re-create the schema binding.

An object may be schema-bound if the following conditions are true:

- All objects employed are schema -bound.
- All objects are accessed using the two-part name (`owner.objectName`).
- All objects are in the same database.
- The user must have proper permissions to reference all the objects in the code.

For example, if you tried to alter your function in the following manner:

```
ALTER FUNCTION album$returnKeysByArtist
(
   @artistId int
)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN (
   SELECT albumId
   FROM album ---- used to be dbo.album
   WHERE artistId = @artistId)
```

This will cause the following error:

```
Server: Msg 4512, Level 16, State 3, Procedure album$returnKeysByArtist, Line 9
Cannot schema bind function 'album$returnKeysByArtist' because name 'album' is invalid for
schema binding. Names must be in two-part format and object cannot reference itself.
```

To remove schema binding, simply execute the `ALTER FUNCTION` without a `WITH SCHEMABINDING` clause; for example, take the first function you built:

```
CREATE FUNCTION album$returnKeysByArtist
(
   @artistId int
)
RETURNS TABLE
WITH SCHEMABINDING
AS
RETURN (
   SELECT albumId
   FROM dbo.album
   WHERE artistId = @artistId )
```

To change this to run without schema binding, you should execute this code as

```
CREATE FUNCTION album$returnKeysByArtist
(
   @artistId int
)
RETURNS TABLE
AS
RETURN (
   SELECT albumId
   FROM dbo.album
   WHERE artistId = @artistId )
```

Multistatement Table-Valued Functions

If single-statement table-valued functions were more complex scalar functions, multistatement table-valued functions are even more complex! What makes them great is that you can define functions with relatively unlimited commands and build a single table, the nature of which you define in the function declaration. In this example, you'll return all of the albums for a particular artist *and* set all of the names to uppercase:

```
CREATE FUNCTION album$returnNamesInUpperCase
(
   @artistId int
)
RETURNS @outTable table (
   albumId int,
   name varchar(60) )
WITH SCHEMABINDING
AS
BEGIN
   --add all of the albums to the output table
   INSERT INTO @outTable
   SELECT albumId, name
   FROM dbo.album
   WHERE artistId = @artistId

   --second superfluous update to show that it
   --is a multi-statement function
   UPDATE @outTable
   SET name = UPPER(name)
   RETURN
END
GO
```

When you execute the following statement:

```
SELECT * FROM dbo.album$returnNamesInUpperCase(1)
```

it returns

```
albumId     name
----------- -----------------------------------
1           THE WHITE ALBUM
2           REVOLVER
```

Their usage is much like the single-value table-valued functions, but they open up a number of new possibilities. Operations that were previously done using temporary tables with single-use code can now be done in a function, where they can be reused, and without coding messy temporary tables.

What User-Defined Functions Can't Do

There are a few things that you can't do using UDFs. The first thing to note is that there are a few important functions built-in to SQL Server that can't be included in UDFs. The reasons for this are fairly simple: they are not deterministic. (Deterministic functions are those functions that when executed with a given set of parameters always return the same set of values.) The reasoning behind this is sensible. UDFs can be used to build calculated columns, or columns in views, and each of these can now be indexed. Indexing can only occur if the columns can be guaranteed to return the same values for each function call.

The following table lists the nondeterministic functions that can't be used in UDFs:

@@CONNECTIONS	@@TOTAL_ERRORS	@@CPU_BUSY	@@TOTAL_READ
@@IDLE	@@TOTAL_WRITE	@@IO_BUSY	@@MAX_CONNECTIONS
@@PACK_RECEIVED	@@PACK_SENT	@@PACKET_ERRORS	@@TIMETICKS
TEXTPTR	NEWID	RAND	GETDATE
GETUTCDATE			

I won't list all of the functions in SQL Server and their deterministic status. This is fully covered in SQL Server 2000 Books Online under the *Deterministic and Nondeterministic Functions* topic.

A second key point is that the state of the server and all data must be exactly the same as when the statements are executed. Hence any changes that extend to objects that aren't local to the function (such as sending back results, data, or data structures, cursors that aren't local to the function, or any other modification to system state) can't be made. To illustrate this, let's create a function that will update all of the albums in your tables to "the Beatles", and then return the number of values you've updated.

```
CREATE FUNCTION album$setAllToTheBeatles
(
)
RETURNS int AS
BEGIN
   DECLARE @rowcount int

   UPDATE album
   SET artistId = 1
   WHERE artistId <> 1

   RETURN 1
END
```

This will fail upon compile because of the `UPDATE`:

```
Server: Msg 443, Level 16, State 2, Procedure album$setAllToTheBeatles, Line 9
Illegal use of 'UPDATE' within a FUNCTION.
```

Summary

User-defined functions are so important because they give you some very important additions to the T-SQL language. I have shown you three types of functions:

- **Scalar functions:** Return a single value, just like all of the built-in system functions that are a longstanding part of SQL Server. They allow you to extend SQL Server, and place commands at a row level (in the `SELECT` or `WHERE` clause) or even in Constraints (a topic I cover in some length in Chapter 11).
- **Inline table-valued:** More or less a parameterized view. Must be based on a single `SELECT` statement, but may have parameters. Good tool when you need to have view functionality, but can always filter out rows using a given set of criteria.
- **Multistatement table-valued:** Views on steroids. So often a view wasn't possible because of the need to create some temporary data, or do some data manipulation that couldn't be done in a single statement.

You learn how to use user-defined functions in Chapter 11 when building constraints.