CHAPTER 19

Stored Procedures—Part 2

The last chapter taught you pretty much all you need to know to start creating stored procedures. We'll enhance that knowledge in this chapter by introducing user-defined data types (UDT for short) and taking a detailed look at set-based logic. It's one of the most important things you need to understand as a database developer, and once you understand it your database development skills will accelerate—fast!

Enhancing the InsertContact Stored Procedure

The `InsertContact` stored procedure we created in the last chapter is pretty solid now. Not only does it create a contact and return the ID for that contact (in two different ways), but it also adds an optional note. Hang on a moment, though. The `ContactNotes` table is a child table of `Contacts`. This means we can add multiple notes for a contact. But our stored procedure is limiting us to adding one note when we create a new contact record. Can we change this situation? You bet your bottom dollar we can!

User-Defined Types

`INT`, `VARCHAR`, `DATE`, and `BIT` are all examples of the data types built into SQL Server. But it's also possible to create your own data types, too, building on top of the standard types.

Why Use Them?

For most purposes, UDTs are not the best way to go. Say you want to store a percentage. Yes, you could define your own `PERCENTAGE` data type. But this is adding unnecessary complexity to your database and application, especially when you consider SQL Server already provides a host of data types that could store a percentage, like floats and decimals. I find the power of UDTs really comes to the fore when I want to pass multiple values of the same type into a stored procedure.

Our `InsertContact` stored procedure can currently insert one note via the `@Note` parameter. We can expand that parameter so it can pass in multiple notes, all of which we can then insert into the `ContactNotes` table. Sounds exciting!

Creating a UDT

To create a UDT, call the `CREATE TYPE` command. There is no corresponding `ALTER TYPE` command—if you want to modify the type you need to drop it and recreate it. This can become difficult when the type is in use, so you really do need to make sure the type is correct before pushing it into general use.

The full definition for `CREATE TYPE` can be found at `https://msdn.microsoft.com/en-us/library/ms175007.aspx`.

A UDT can be one of two things: either an alias to an existing data type (much like our `PERCENTAGE` example, which could be an alias of `DECIMAL`), or a completely new data type consisting of one or more values. We'll create the latter. Our data type will model a `ContactNote`, and will have two properties.

- `ContactId` (`INT`)
- `Note` (`VARCHAR(200)`)

The only other column in the `ContactNotes` table is `NoteId`, but this is auto-generated so there is nothing to gain by adding it to the UDT.

Open a new window and create the UDT. Because our UDT has two properties, we create it as a table.

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.types WHERE [name] = 'ContactNote')
BEGIN
```

```
DROP TYPE dbo.ContactNote;
END

CREATE TYPE dbo.ContactNote
AS TABLE
(ContactId INT, Note VARCHAR(200));

GO
```

The script starts with the usual check to see if the object exists. We are querying `sys.types` in this check. Next is a very standard table declaration—indeed, `CREATE TYPE` is about the only new thing to us here. It looks very much like a function declaration without parameters (just the brackets are missing). The name of the type is important, and a schema should always be specified. Our type is called `ContactNote`.

Save this script as `c:\temp\sqlbasics\apply\26 - Create ContactNote UDT.sql`. I told you we'd create script 26! Our stored procedure, once changed, will be reliant on the `ContactNote` UDT, so it needs to exist before the stored procedure is created. If we run all of our scripts from the `00 - Apply.sql` script, it's important that the UDT script run before the stored procedure script.

Once you've saved the script, you can add it to `00 - Apply.sql`, above script 27. Run script 26 to create the type. You can view the UDT in SSMS, under **Programmability** ➤ **Types** (Figure 19-1).

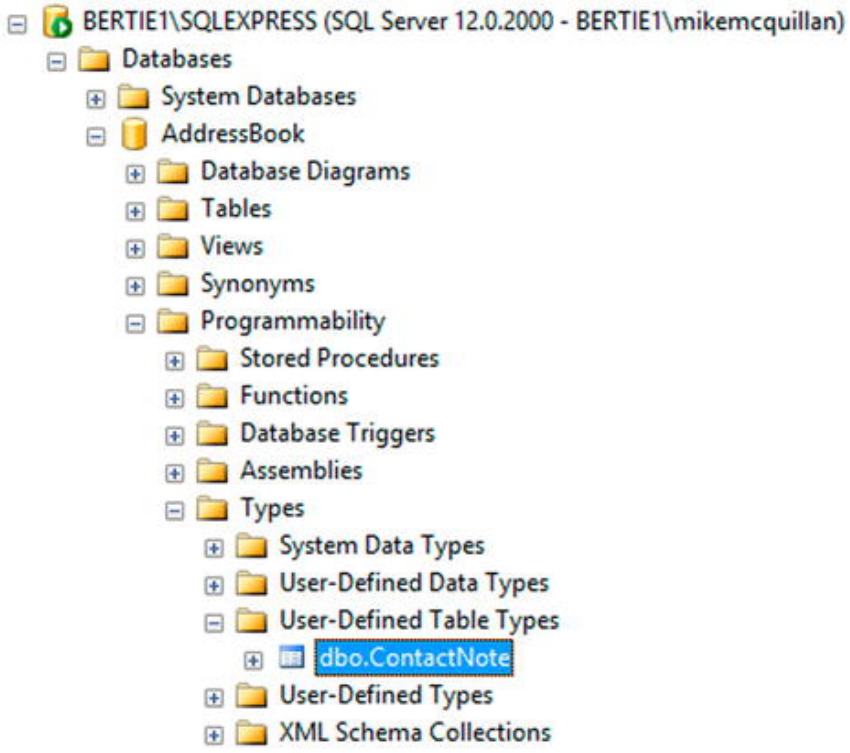

Figure 19-1. *Viewing custom user-defined table types*

Rolling Back the UDT

Before we start using the UDT, create a rollback script for it called `c:\temp\sqlbasics\rollback\26 - Create ContactNote UDT Rollback.sql`. Here's the code; make sure you add it to `00 - Rollback.sql` in the correct place (just below script 27).

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.types WHERE [name] = 'ContactNote')
BEGIN
DROP TYPE dbo.ContactNote;
END;

GO
```

Using the UDT

Use the UDT like you do any other data type. Try running this sample script:

```
USE AddressBook;

DECLARE @IntValue INT, @VarcharValue VARCHAR(10), @ContactNoteValue CONTACTNOTE;

SELECT @IntValue = 1, @VarcharValue = 'Hello';

INSERT INTO @ContactNoteValue (ContactId, Note) VALUES (@IntValue, @VarcharValue);

SELECT * FROM @ContactNoteValue;
```

This yields a single row from the `SELECT` statement, which is shown in **Figure 19-2**.

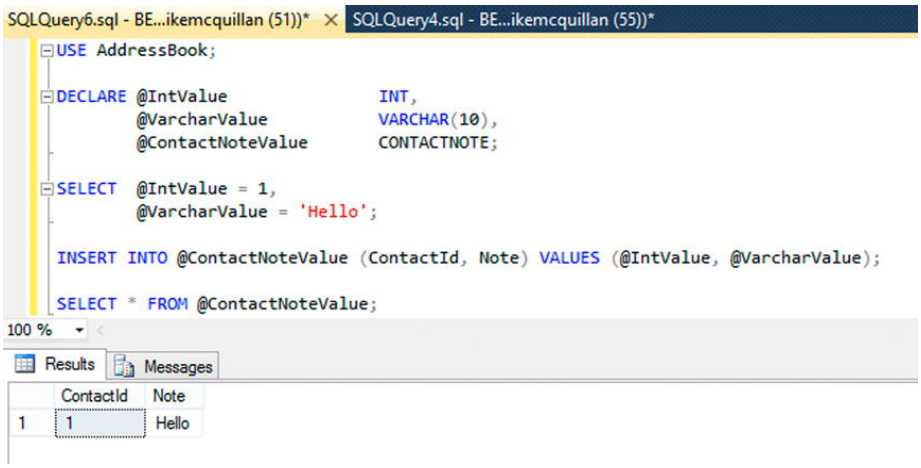

Figure 19-2. *Sample script using the* `ContactNote` *UDT*

The important thing to take away is the `CONTACTNOTE` data type is nothing more than a table, and can be used exactly like any other table. This means we can declare a table, populate it, and then pass the populated table to the `InsertContact` stored procedure.

Adding a Custom Type to a Stored Procedure

Adding the type to the stored procedure is easy. We just need to replace the `@Note` parameter with a `@Notes` parameter of type `ContactNote`. The complication comes with adding code to insert multiple records into the `ContactNotes` table. Open up apply script 27 and edit the procedure so it matches the following code. This is the first attempt at rewriting the procedure.

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.procedures WHERE [name] = 'InsertContact')
BEGIN
DROP PROCEDURE dbo.InsertContact;
END;

GO

CREATE PROCEDURE dbo.InsertContact
(
@FirstName VARCHAR(40),
@LastName VARCHAR(40),
@DateOfBirth DATE,
@AllowContactByPhone BIT = 0,
@Notes CONTACTNOTE READONLY,
@ContactId INT = 0 OUTPUT
)
AS
BEGIN

SET NOCOUNT ON;
```

```
-- Add variables to support note processing
DECLARE @TempNotes TABLE (NoteId INT IDENTITY(1,1), Note VARCHAR(200));
DECLARE @RecordCount INT, @LoopCounter INT, @NoteId INT;

-- Copy the @Notes table, which is readonly, to a table variable so we can
-- modify the data
INSERT INTO @TempNotes (Note) SELECT Note FROM @Notes;

-- Set defaults for the loop around the notes
SELECT @RecordCount = COUNT(1), @LoopCounter = 0 FROM @TempNotes;

-- Remove any notes that are empty
DELETE FROM @TempNotes WHERE LTRIM(RTRIM(COALESCE(Note, ''))) = '';

BEGIN TRANSACTION;

INSERT INTO dbo.Contacts(FirstName, LastName, DateOfBirth, AllowContactByPhone)
VALUES (@FirstName, @LastName, @DateOfBirth, @AllowContactByPhone);

SELECT @ContactId = SCOPE_IDENTITY();

PRINT 'Contact ID inserted: ' + CONVERT(VARCHAR(20), @ContactId);

-- Insert notes using WHILE loop
WHILE (@LoopCounter < @RecordCount)
BEGIN

SELECT TOP (1) @NoteId = NoteId FROM @TempNotes;

INSERT INTO dbo.ContactNotes (ContactId, Notes)
SELECT @ContactId, Note FROM @TempNotes WHERE NoteId = @NoteId;

DELETE @TempNotes WHERE NoteId = @NoteId;

SELECT @LoopCounter = @LoopCounter + 1;
END;

COMMIT TRANSACTION;

SELECT @ContactId AS ContactId;

SET NOCOUNT OFF;

END;

GO
```

Whew, our code has certainly grown! I thought using a UDT was supposed to make things easier! Let's have a walkthrough and figure out what this code is doing. The first change is in the procedure declaration.

```
CREATE PROCEDURE dbo.InsertContact
(
@FirstName VARCHAR(40),
@LastName VARCHAR(40),
@DateOfBirth DATE,
@AllowContactByPhone BIT = 0,
@Notes CONTACTNOTE READONLY,
@ContactId INT = 0 OUTPUT
)
```

We now have a `@Notes` parameter, of type `CONTACTNOTE`. The parameter is also marked as `READONLY`. We briefly met the `READONLY` keyword in the last chapter. It only applies to table-valued parameters and it has to be specified for a table-based parameter. This is because SQL Server puts a limitation on TVPs—they cannot be modified once they are passed in. You can work around this by copying the parameter contents to a temporary table or a table variable in the stored procedure code (useful if you do need to edit the rows passed in).

We then have a fairly extensive block of code, introduced at the start of the stored procedure above the `BEGIN TRANSACTION` line. We'll look at the `DECLARE` statements first. Note the single-line comment above the code:

```
-- Add variables to support note processing
DECLARE @TempNotes TABLE (NoteId INT IDENTITY(1,1), Note VARCHAR(200))
DECLARE @RecordCount INT, @LoopCounter INT, @NoteId INT;
```

This is all preparation work for the loop that will insert the notes. The `@TempNotes` variable is a table variable into which the contents of the `@Notes` parameter will be copied. Remember, `@Notes` is read-only. By copying its contents into `@TempNotes` we can manipulate the contents. `@TempNotes` contains a `NoteId`, which is used to identify the row being processed.

Next, we declare three variables. `@RecordCount` will hold the number of notes we are going to insert, and `@LoopCounter` will be used to ensure we don't become stuck in a loop when we are inserting note records. The last variable is `@NoteId`, used to hold the note in `@TempNotes` currently being processed.

```
-- Copy the @Notes table, which is readonly, to a table variable so we can
-- modify the data
INSERT INTO @TempNotes (Note) SELECT Note FROM @Notes;

-- Set defaults for the loop around the notes
SELECT @RecordCount = COUNT(1), @LoopCounter = 0 FROM @TempNotes;

-- Remove any notes that are empty
DELETE FROM @TempNotes WHERE LTRIM(RTRIM(COALESCE(Note, ''))) = '';
```

These three lines are straightforward. The first statement copies the notes from the read-only `@Notes` parameter to the `@TempNotes` table variable. The number of note records to process is then stored in `@RecordCount`, and `@LoopCounter` is set to 0, ready for the loop. Finally, there is a bit of data cleansing—the `DELETE` statement removes any notes from `@TempNotes` that don't have a value. At this point, we are ready to begin inserting records.

```
BEGIN TRANSACTION;

INSERT INTO dbo.Contacts(FirstName, LastName, DateOfBirth, AllowContactByPhone) VALUES (@FirstName,

SELECT @ContactId = SCOPE_IDENTITY();

PRINT 'Contact ID inserted: ' + CONVERT(VARCHAR(20), @ContactId);

-- Insert notes using WHILE loop
WHILE (@LoopCounter < @RecordCount)
BEGIN

SELECT TOP (1) @NoteId = NoteId FROM @TempNotes;

INSERT INTO dbo.ContactNotes (ContactId, Notes) SELECT @ContactId, Note FROM @TempNotes WHERE NoteI

DELETE @TempNotes WHERE NoteId = @NoteId;

SELECT @LoopCounter = @LoopCounter + 1;
END;

COMMIT TRANSACTION;
```

The transaction has grown from the single statement we first implemented. That statement is still present, and is executed first to insert the contact. The next two lines were also already there. The new stuff starts at the `WHILE` loop. The `WHILE` loop in T-SQL executes the same block of code one or more times, until a certain condition is set. The condition in this case is:

```
(@LoopCounter < @RecordCount)
```

`@LoopCounter` is 0 at the beginning of execution, and if `@Notes` contained two records, `@RecordCount` would be 2. So you would have 0 < 2. After the first note is processed, `@LoopCounter` would be 1. We now have 1 < 2. This still holds, so the second note is processed, and `@LoopCounter` is set to 2. Now we have 2 < 2. Two cannot be less than 2, so the loop would exit at this point.

INFINITE LOOP WARNING!

It is so easy to create an infinite loop. And I'm not even joking! The preceding example shows `@LoopCounte` increasing by 1 every time a note is processed. It is incumbent upon you, the developer, to add the line of cod

that increases the value of `@LoopCounter`. If you don't add this line your loop will never exit, and bad things will happen—trust me.

Once in the loop, the `SELECT TOP (1)` statement assigns the first `NoteId` found to `@NoteId`. The `INSERT INTO` statement inserts this record into the `ContactNotes` table. We then call a `DELETE` statement to remove the `NoteId` from `@TempNotes`, so on the next pass round we'll pick up a different `NoteId` (if there are any records left in `@TempNotes`).

The final `SELECT` line within the loop is the most important.

```
SELECT @LoopCounter = @LoopCounter + 1;
```

This increases the value of `@LoopCounter` by 1 for every pass of the loop. Without this line, the loop could never meet its exit condition and would never complete. It would carry on looping around and around, using up precious resources.

INFINITE LOOPS ARE BAD!

I must reiterate—infinite loops are a bad, BAD thing! If your code enters an infinite loop it will never exit; it will just continue looping until the service or system is restarted. It is very easy to accidentally create infinite loops in SQL Server, so be very careful. An infinite loop could ultimately cause your server to crash—a sure way of making yourself unpopular with your team!

Once the loop has completed, we finally commit the transaction. The last few lines of code in the stored procedure are lines we've seen before. We return the generated `ContactId` value and close off the procedure.

```
SELECT @ContactId AS ContactId;

SET NOCOUNT OFF;

END;
```

Executing the Stored Procedure

With the procedure finished, we're in a position to execute it. Run the `CREATE PROCEDURE` statement first to ensure the new version of the procedure has been created. Then try this script in a New Query Window:

```
USE AddressBook;

DECLARE @ContactIdOUT INT, @ContactNotes CONTACTNOTE;

INSERT INTO @ContactNotes (ContactId, Note)
VALUES
(NULL, 'Mark Kermode contributes to the BBC Radio 5 film programme'),
(NULL, 'Mark thinks The Exorcist is the best film ever made.');

EXEC dbo.InsertContact
@FirstName = 'Mark',
@LastName = 'Kermode',
@DateOfBirth = '1963-07-02',
@Notes = @ContactNotes,
@ContactId = @ContactIdOUT OUTPUT;

SELECT @ContactIdOUT AS ContactIdFromOutputVariable;

SELECT * FROM dbo.Contacts WHERE ContactId = @ContactIdOUT;
SELECT * FROM dbo.ContactNotes WHERE ContactId = @ContactIdOUT;
```

We've added a variable of type `CONTACTNOTE`, and we promptly insert two records into this table (we don't know what the `ContactId` value is for these records, so we set it to `NULL`). Then we call `InsertContact`. This call is exactly the same as it was earlier, except we pass `@ContactNotes`, containing the rows, to the `@Notes` parameter. Interestingly, we don't have to specify the `READONLY` keyword here, unlike `OUTPUT`, whi had to be specified in both the procedure definition and the procedure call.

There are three `SELECT` statements to finish off. The first was already there and returns the new `ContactId`. The other two select the new contact's details from the `Contacts` table and the `ContactNotes` table, respectively. Run it and take a look at the output (Figure 19-3).

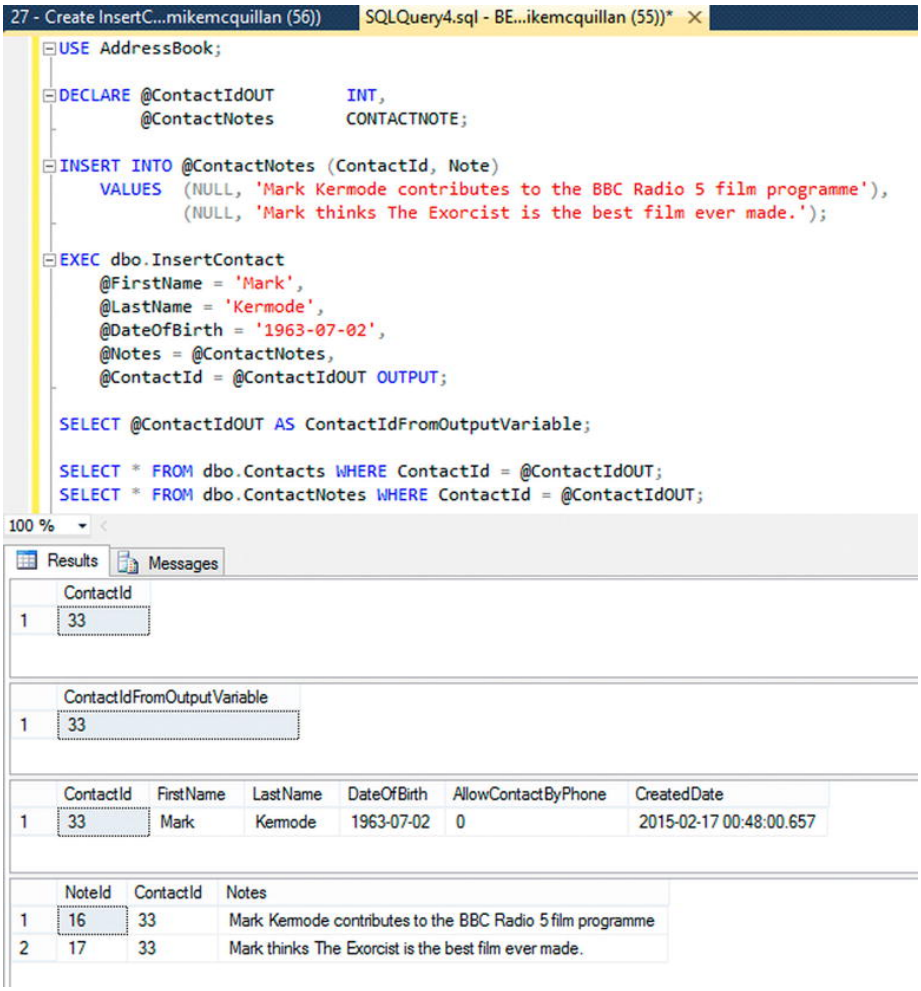

Figure 19-3. *Using the amended stored procedure to insert notes*

Your ID values may differ from mine, so don't worry if they don't match up with Figure 19-3. Four result sets have been returned. The first comes from the stored procedure, and the other three are the three statements we were just discussing. Everything looks perfect—we've successfully added both notes using the UDT. We declared a variable of type `CONTACTNOTE` (which is really a table). We popped two records into this table, passed it to the stored procedure, and the code therein did the rest. Go us!

But . . . we said (OK, *I* said) we were going to discuss set-based logic in this chapter. And so we are. Right now.

Set-Based Logic

Our stored procedure is now quite large and uses a lot of custom code to process the notes. Indeed, the notes are being processed one at a time, using the `WHILE` loop. This is madness! We can eliminate about half of the code in the stored procedure by using set-based logic.

As it stands right now, the procedure does this:

- Sets up code for the `WHILE` loop
- Inserts the contact
- Executes a `WHILE` loop to insert note records, one at a time

This is really unpleasant. If a developer makes a bad change to the code, we could be stuck in an infinite loop or we could accidentally insert the same note two or more times. It's also much slower to use a loop than set-

based logic, which apart from performance implications means we are unnecessarily keeping our transaction open longer than we need to.

Set-based logic is the answer! With this, all we will do is:

- Insert the contact
- Insert the notes

Nice and simple. Here is the new version of the stored procedure.

```
CREATE PROCEDURE dbo.InsertContact
(
 @FirstName VARCHAR(40),
 @LastName VARCHAR(40),
 @DateOfBirth DATE,
 @AllowContactByPhone BIT = 0,
 @Notes CONTACTNOTE READONLY,
 @ContactId INT = 0 OUTPUT
)
AS
BEGIN

SET NOCOUNT ON;

BEGIN TRANSACTION;

INSERT INTO dbo.Contacts(FirstName, LastName, DateOfBirth, AllowContactByPhone) VALUES (@FirstName,

SELECT @ContactId = SCOPE_IDENTITY();

PRINT 'Contact ID inserted: ' + CONVERT(VARCHAR(20), @ContactId);

INSERT INTO dbo.ContactNotes (ContactId, Notes) SELECT @ContactId, Note FROM @Notes;

COMMIT TRANSACTION;

SELECT @ContactId AS ContactId;

SET NOCOUNT OFF;

END;
```

WOW, that's a lot shorter! Much more accurate, too. There isn't really anything to discuss here, other than the `INSERT INTO dbo.ContactNotes`. Now it just uses a `SELECT FROM @Notes`! We were splitting the set into individual records earlier and processing them one by one—this is a cursor-based approach. Now we're harnessing the power of SQL Server by using the set, treating all records as a group and processing them in one batch. Run this code to create the new version of the stored procedure, and save the code as script 27.

Switch to a New Query Window and run the script in Figure 19-4. The outcome it shows is the same as the earlier version we ran; the code is just much more efficient now.

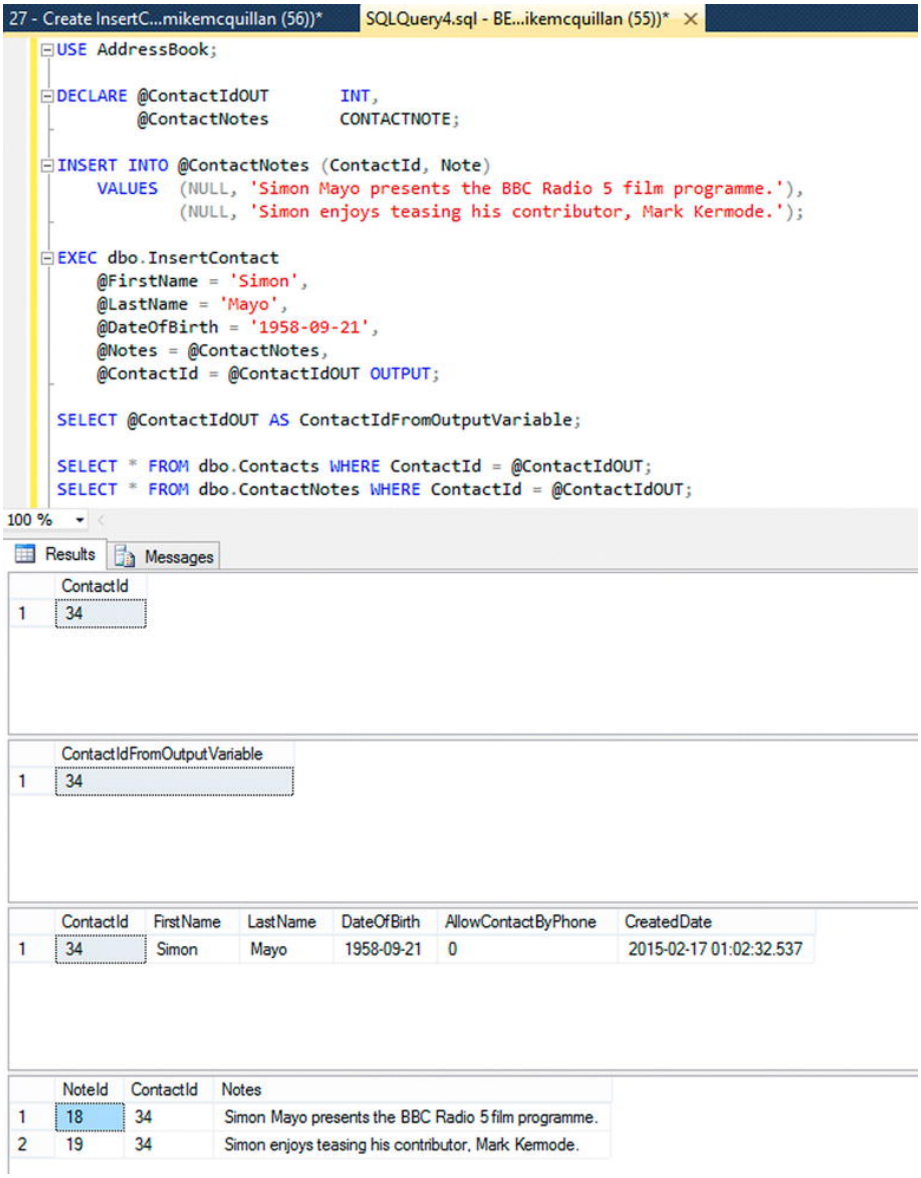

Figure 19-4. *Multiple notes with more efficient code*

This is great. We've significantly reduced the amount of code we had to write and the end result is still the same (remember, don't worry if your ID values differ from those in Figure 19-4—it just shows that you have been playing around with the contact data).

It's hard to believe, but I've lost count of how many times I've come across the "bad" code presented in the first version of our stored procedure. The response from the developer is always the same—"It works." NO IT DOESN'T! Code like this is a time bomb waiting to go off. Always explore and try to learn the best way of doing things. Don't settle for "That's the way we always do it." Challenge ideas and carry on learning. Do that and you'll have a great career. And maybe one day I'll stop finding `WHILE` loops replacing set-based logic!

Never forget: always think about the next person.

Summary

Our database is finished! There is nothing else we need to do to it. We've covered most of the things you will need to give you a solid basis with SQL Server. We have one more chapter to go, which will talk about a few things we haven't yet discussed, and give you some pointers for the future.

PREV
Chapter 18 : Stored Procedures…

NEXT
Chapter 20 : Bits and Pieces