CHAPTER 18

Stored Procedures—Part 1

Stored procedures—or SPs, as some lovingly call them—are likely to become your best friends as you spend more and more time with SQL Server. It is not uncommon to come across databases that contain hundreds of stored procedures—they're just so darned flexible! In this chapter, we'll find out exactly why stored procedures are so useful, and we'll create a couple of SPs of our own.

What Are Stored Procedures, and Why Should I Use Them?

A stored procedure is a repeatable piece of code. You'd be forgiven for thinking, "Hold everything!"—isn't that what a function does? Well yes, it is. But a function is intended to do one simple thing well, like the `ContactName` function. If you need complexity, stored procedures are the way to go. Functions cannot have side effects; stored procedures have no such limits. In a stored procedure, you can write `SELECT` statements alongside `INSERT`, `UPDATE`, and `DELETE` statements. You can incorporate business rules, such as preventing inserts if a record doesn't meet certain criteria. Stored procedures really are incredibly flexible. Some great reasons to use them are the following:

- **Performance**: Stored procedures will perform better than having the equivalent code running individually every time. SQL Server stores query plans for stored procedures, allowing it to execute them faster than ad hoc queries.
- **Encapsulation**: You can embed as much code and logic into a stored procedure as you want. All of this is hidden from the developer calling the stored procedure—they just need to know what parameters they need to pass to execute the stored procedure.
- **Improve maintainability**: You can easily change how a stored procedure works internally, without affecting any applications calling it. You just need to make sure you don't modify the inputs and outputs.
- **Multiple output options**: Stored procedures can return one or more record sets, as well as values in output parameters.
- Stored procedures can call other stored procedures.
- Stored procedures can call scalar and table-valued functions.
- You can create any type of procedure you can think of—for searching, inserting, updating, deleting. . . .
- Almost any valid piece of T-SQL can be called within a stored procedure (and for those that can't, there is normally a workaround). The only things you can't call directly are some `CREATE` or `ALTER` statements, such as `CREATE PROCEDURE` or `ALTER PROCEDURE`.

Speaking of which. . . .

CREATE PROCEDURE and ALTER PROCEDURE

To create a stored procedure, you use the `CREATE PROCEDURE` command. The full definition of this statement can be found at `https://msdn.microsoft.com/en-us/library/ms187926.aspx`, but the basic form you'll use most of the time is:

```
CREATE PROCEDURE SchemaName.ProcedureName
(
Parameters (optional)
)
AS
BEGIN

Do something...

END;
```

`ALTER PROCEDURE` is exactly the same, just replacing `CREATE` with `ALTER`. Altering a procedure maintains permissions allocated to the procedure. When altering a procedure, the entire definition must be provided. You cannot alter a small piece of a procedure; it is all or nothing.

DROP PROCEDURE

To remove a procedure from your database, call `DROP PROCEDURE`. This is as simple as providing a procedure name:

```
DROP PROCEDURE SchemaName.ProcedureName;
```

SSMS and Stored Procedures

Because of the nature of stored procedures (they contain customized T-SQL code that you write), you can't really create them through SSMS. I find it much easier to open a New Query Window and begin typing the stored procedure definition. But if you wish, you can create the outline of a stored procedure from SSMS, and you can certainly manage your stored procedures via SSMS.

Go ahead and open SSMS. Then go to hte Object Explorer and expand **Databases** ➤ **AddressBook** ➤ **Programmability** ➤ **Stored Procedures**. If you try to expand **Stored Procedures** it will be empty except for a **System Stored Procedures** item. Any stored procedures you create will appear under the **Stored Procedures** node. Once you have a stored procedure there, you can right-click it (see Figure 18-1), which will give you the ability to modify, execute, or script the stored procedure.

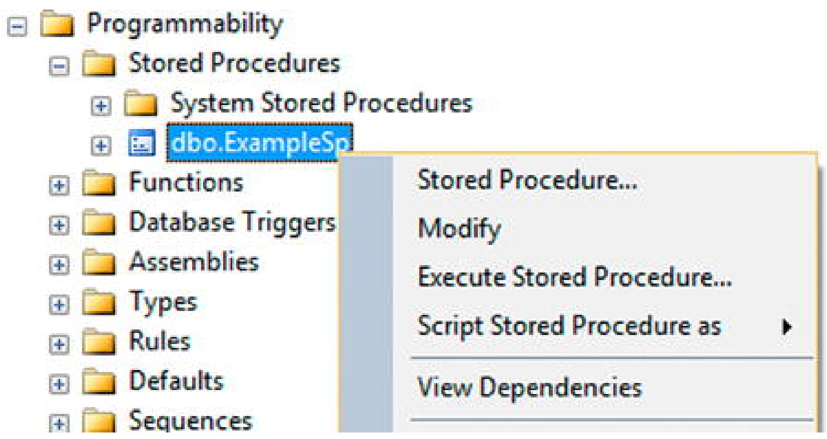
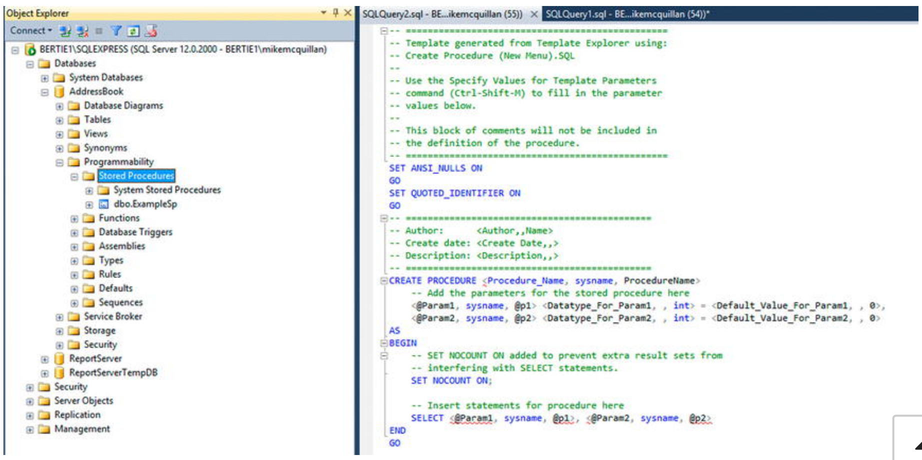


Figure 18-1. *Right-clicking on a stored procedure in SSMS*

To create a new stored procedure from SSMS, right-click the **Stored Procedures** node, and choose the **Stored Procedure** option (this also appears when you right-click an existing procedure, as in Figure 18-1). A New Query Window opens (Figure 18-2), containing the basic outline of a stored procedure.



```
-- ================================================
-- Template generated from Template Explorer using:
-- Create Procedure (New Menu).SQL
--
-- Use the Specify Values for Template Parameters
-- command (Ctrl-Shift-M) to fill in the parameter
-- values below.
--
-- This block of comments will not be included in
-- the definition of the procedure.
-- ================================================
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Author:      <Author,,Name>
-- Create date: <Create Date,,>
-- Description: <Description,,>
-- =============================================
CREATE PROCEDURE <Procedure_Name, sysname, ProcedureName>
    -- Add the parameters for the stored procedure here
    <@Param1, sysname, @p1> <Datatype_For_Param1, , int> = <Default_Value_For_Param1, , 0>,
    <@Param2, sysname, @p2> <Datatype_For_Param2, , int> = <Default_Value_For_Param2, , 0>
AS
BEGIN
    -- SET NOCOUNT ON added to prevent extra result sets from
    -- interfering with SELECT statements.
    SET NOCOUNT ON;

    -- Insert statements for procedure here
    SELECT <@Param1, sysname, @p1>, <@Param2, sysname, @p2>
END
GO
```

Figure 18-2. *Stored procedure outline via SSMS*

We won't be using the SSMS features for our stored procedures—we'll do everything manually. In the long run you'll probably find this makes more sense, but if you really, *really* want to use SSMS to create your procedure outlines, I won't stand in your way.

A First Stored Procedure

To begin our stored procedure journey, we'll create a simple stored procedure to insert a new contact. The procedure body will use code we first met in Chapter 15 when we were investigating how transactions work. We begin by declaring the procedure's name and parameters.

```
USE AddressBook;

GO

CREATE PROCEDURE dbo.InsertContact
(
@FirstName VARCHAR(40),
@LastName VARCHAR(40),
@DateOfBirth DATE,
@AllowContactByPhone BIT
)
```

There is a GO statement between the USE statement and the CREATE PROCEDURE statement. Much like CREATE FUNCTION, CREATE PROCEDURE must be the first statement in a batch.

We've declared a procedure called InsertContact, which exists within the dbo schema. It accepts four parameters, each of which corresponds to a matching column in the Contacts table.

With the procedure declared, we move on to the body of the procedure. All we want to do is insert a new record into the Contacts table. We wrap up the INSERT INTO statement in a transaction.

```
USE AddressBook;

GO

CREATE PROCEDURE dbo.InsertContact
(
@FirstName VARCHAR(40),
@LastName VARCHAR(40),
@DateOfBirth DATE,
@AllowContactByPhone BIT
)
AS
BEGIN

BEGIN TRANSACTION;

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

COMMIT TRANSACTION;

END;

GO
```

This is a very simple stored procedure—so simple, we haven't even added the insert into ContactVerificationDetails as we implemented in Chapter 15. This is left as an exercise for the reader to perform, which can be found in Appendix D.

Press F5 to run this code, and the stored procedure will be created. Now we can try to use it.

Executing a Stored Procedure

Open a New Query Window. The EXECUTE statement, or EXEC for short, is used to execute a stored procedure. I usually use EXEC, just because it is less typing. To call our InsertContact procedure, you would type:

```
EXEC dbo.InsertContact;
```

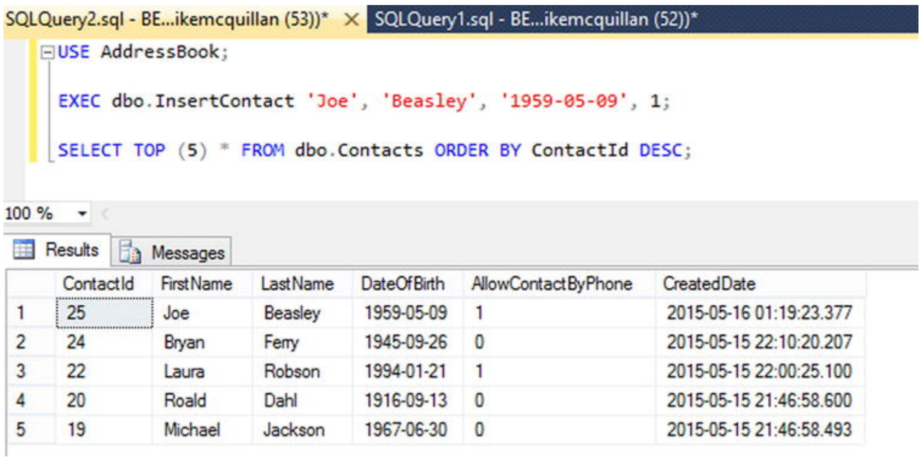
But if you run this, it fails:

```
Msg 201, Level 16, State 4, Procedure InsertContact, Line 0
Procedure or function 'InsertContact' expects parameter '@FirstName', which was not supplied.
```

D'oh! We didn't provide the parameters. There are two ways you can pass parameters to a stored procedure. Here's the lazy way:

```
EXEC dbo.InsertContact 'Joe', 'Beasley', '1959-05-09', 1;
```

If you run this, it will work, and running the `SELECT` statement in Figure 18-3 against the `Contacts` table will show you the record was successfully inserted.


SQLQuery2.sql - BE...ikemcquillan (53))*
SQLQuery1.sql - BE...ikemcquillan (52))*
100 %
Results
Messages

```
USE AddressBook;

EXEC dbo.InsertContact 'Joe', 'Beasley', '1959-05-09', 1;

SELECT TOP (5) * FROM dbo.Contacts ORDER BY ContactId DESC;
```

	ContactId	FirstName	LastName	DateOfBirth	AllowContactByPhone	CreatedDate
1	25	Joe	Beasley	1959-05-09	1	2015-05-16 01:19:23.377
2	24	Bryan	Ferry	1945-09-26	0	2015-05-15 22:10:20.207
3	22	Laura	Robson	1994-01-21	1	2015-05-15 22:00:25.100
4	20	Roald	Dahl	1916-09-13	0	2015-05-15 21:46:58.600
5	19	Michael	Jackson	1967-06-30	0	2015-05-15 21:46:58.493

Figure 18-3. *Proving the stored procedure works after running it*

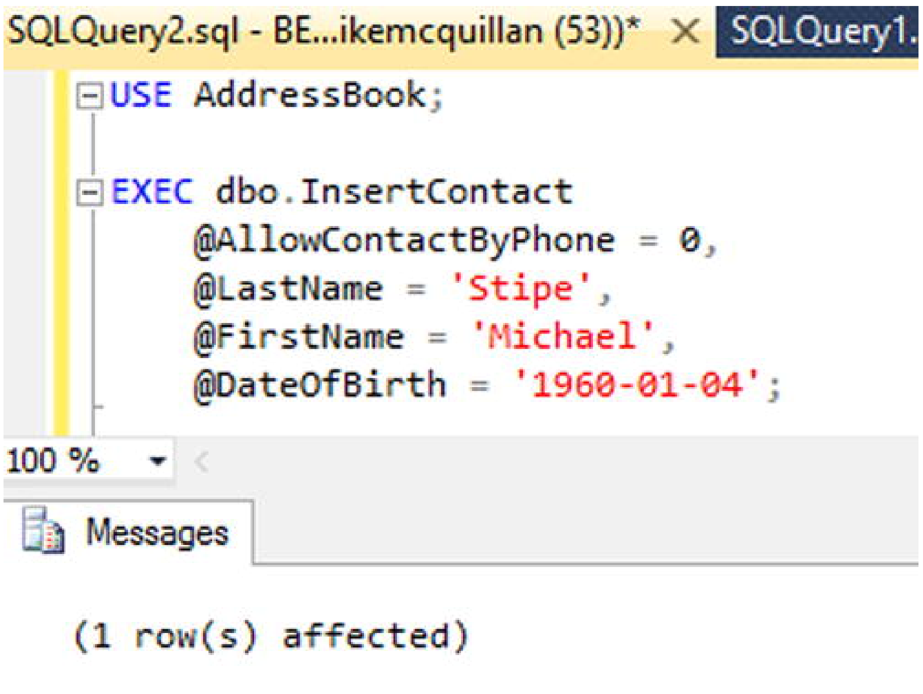
We can see the stored procedure is working—it is inserting records. But it's difficult to tell which parameter each value is being assigned to. To make things easier, we can provide the parameter names—this is called *using named parameters*.

```
EXEC dbo.InsertContact
@FirstName = 'Michael',
@LastName = 'Stipe',
@DateOfBirth = '1960-01-04',
@AllowContactByPhone = 0;
```

This is much easier to understand. You cannot mix and match named parameters—if you use named parameters, you must name every parameter (unless it has a default value, which we are coming to). One benefit of named parameters is you can change the order in which they are specified—you cannot do this if you don't name the parameters:

```
EXEC dbo.InsertContact
@AllowContactByPhone = 0,
@LastName = 'Stipe',
@FirstName = 'Michael',
@DateOfBirth = '1960-01-04';
```

No matter which order you specify the parameters, as long as they are named, the call will work. Execute the preceding statement—this time without the `SELECT` statement—and you'll be told one row was inserted (see Figure 18-4).



```
USE AddressBook;

EXEC dbo.InsertContact
    @AllowContactByPhone = 0,
    @LastName = 'Stipe',
    @FirstName = 'Michael',
    @DateOfBirth = '1960-01-04';
```

```
(1 row(s) affected)
```

Figure 18-4. *Running the stored procedure with row count returned*

That **(1 row(s) affected)** message in Figure 18-4 is actually a bad thing—some of the time.

SET NOCOUNT ON and OFF

Quite often, especially when you are developing, you do want to see how many rows have been affected by your statements. But once development is over and you are releasing your code, it is advisable to turn these informational messages off. Doing so improves performance, as SQL Server doesn't need to send the counts affected messages back over the network.

To turn the messages off, turn `SET NOCOUNT` to `ON`. It is good practice to turn this setting back `OFF` at the end of your procedures. Change the `InsertContact` stored procedure so it includes `SET NOCOUNT ON` and `SET NOCOUNT OFF`. We'll also add a check at the top of the script to `DROP` the procedure if it already exists. You guessed it, we'll query `sys.procedures` to do this.

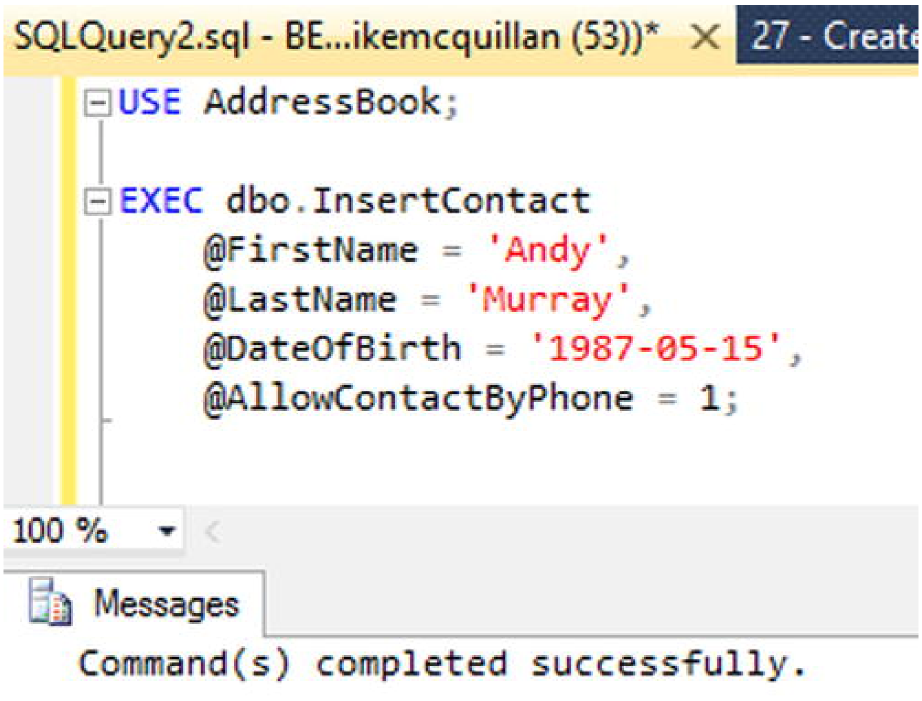
```
USE AddressBook;

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
)
AS
BEGIN

SET NOCOUNT ON;

BEGIN TRANSACTION;

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

COMMIT TRANSACTION;

SET NOCOUNT OFF;
```

```
END;

GO
```

Save this script as `c:\temp\sqlbasics\apply\27 - Create InsertContact Stored Procedure.sql`. Add it to `00 - Apply.sql` while we're about it. I know we haven't created script 26, but all will be revealed in the next chapter! Here is the code for the apply script:

```
:setvar currentFile "27 - Create InsertContact Stored Procedure.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Run script 27, and then execute the stored procedure again to add another contact. This time you'll just see the message **Command(s) completed successfully** (**Figure 18-5**).



```
USE AddressBook;

EXEC dbo.InsertContact
    @FirstName = 'Andy',
    @LastName = 'Murray',
    @DateOfBirth = '1987-05-15',
    @AllowContactByPhone = 1;
```

```
Command(s) completed successfully.
```

Figure 18-5. *Running the stored procedure with* `SET NOCOUNT ON`

For stored procedures that execute multiple DML statements, turning `SET NOCOUNT ON` can really help with performance. It's not a massive gain, but every little helps.

Schema Names

You should always use schema names when calling stored procedures (which is what we've been doing so far). This gives a slight performance improvement as SQL Server doesn't need to figure out which schema an object exists in—it can go straight to the object. This also prevents possible future problems if an object with the same name is created in a different schema. Do anything you can to extract a little more performance—especially the simple things!

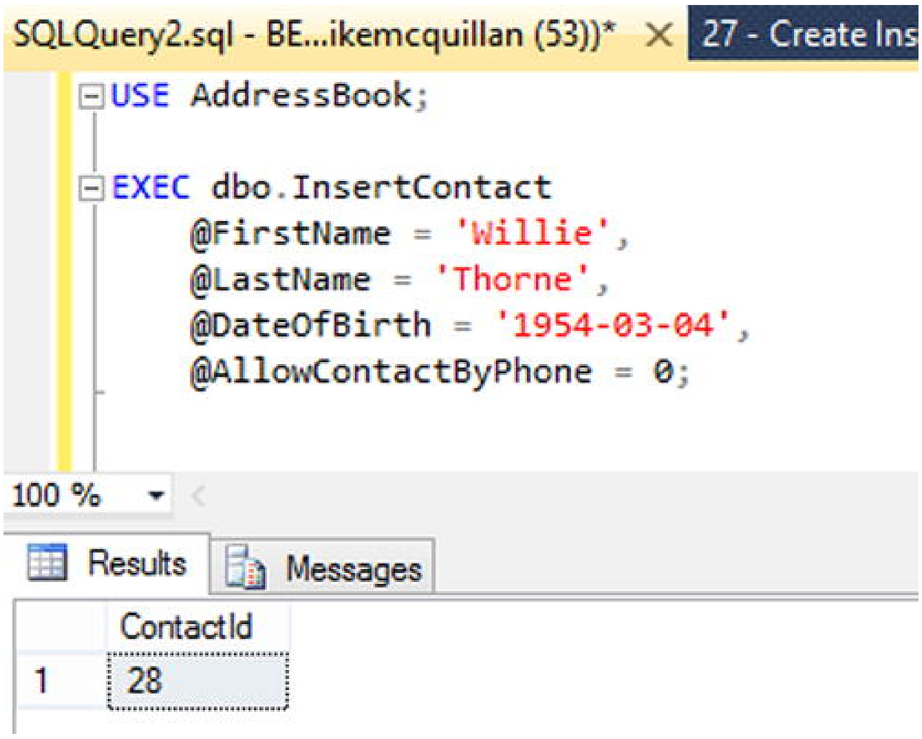
Returning Data from Stored Procedures

We can enhance our stored procedure to return data. If you think about it, what do we need to return once we've inserted a new record into Contacts? You win $64,000 if you said the `ContactId`. In most systems, you will want the ID of the record you inserted to be returned to you, so you can then process any updates or deletes using the ID (in general, you should always return the primary key value, which is usually the ID). There are a couple of ways we can return this value. To start with, we'll use the method we've already seen in earlier chapters—assigning the value to a variable using `SCOPE_IDENTITY()`, then running a `SELECT` to return that variable's value. Amend the stored procedure to match this updated listing. Don't forget to save your changes.

```
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
)
AS
BEGIN

SET NOCOUNT ON;

DECLARE @ContactId INT;

BEGIN TRANSACTION;

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

SELECT @ContactId = SCOPE_IDENTITY();

COMMIT TRANSACTION;

SELECT @ContactId AS ContactId;

SET NOCOUNT OFF;

END;

GO
```

The stored procedure now has a `@ContactId` variable, which is assigned the value of `SCOPE_IDENTITY()` after the `INSERT INTO` occurs. It's interesting to see that the `SCOPE_IDENTITY()` call happens within the transaction, but the new `SELECT` statement that returns it happens outside the transaction. Remember what we were saying earlier about transactions—keep them as short as possible. Here, it makes a lot of sense to perform the `SCOPE_IDENTITY()` assignment within the transaction, as we've just performed the insert. But we can wait until the transaction has completed before returning the record set, otherwise the record set will return before the transaction commits.

If you run this code and then return to the window holding your `EXEC` call, executing the stored procedure again now looks a little more interesting, as Figure 18-6 shows.



```
USE AddressBook;

EXEC dbo.InsertContact
     @FirstName = 'Willie',
     @LastName = 'Thorne',
     @DateOfBirth = '1954-03-04',
     @AllowContactByPhone = 0;
```

```
   ContactId
1  28
```

Figure 18-6. *Amended procedure returning the new* `ContactId`

This is very useful. Imagine a web site is calling our stored procedure. The `ContactId` is returned in a record set, and the web site can store this in a variable for future use.

It's possible to return multiple record sets from a stored procedure; every `SELECT` you write within a stored procedure that returns a record set will indeed return the record set when the stored procedure is called.

The `SELECT` output is really useful if you want a front-end system of some type to call this stored procedure. It isn't so useful if you want to use the `ContactId` from within a different SQL Server script (it can be done without too much difficulty, but it involves a temp table and is messy). Never fear, there is an easier way. We'll see this during our discussion of parameters.

Parameters

Parameters allow you to pass data into a stored procedure, but they also offer ways of returning data from a stored procedure. There are a couple of additional properties you can add to parameters, too, all of which helps make your code more robust.

Input Parameters

We've already met input parameters. The four parameters our current `InsertContact` stored procedure is using are all input parameters. You use an input parameter to pass values into the stored procedure that you want to use, such as the values we pass in so we can insert a new contact.

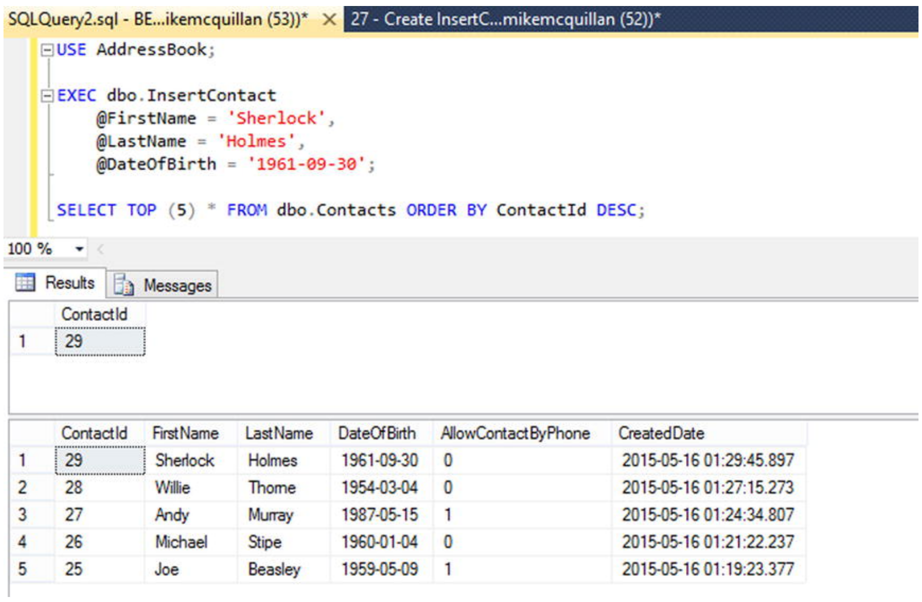
Default Parameter Values

It is possible to assign default values to parameters. This can be very useful, as it means you don't have to specify the variable name when calling the stored procedure. You can also add checks into your code to see if a variable has the default value, and if it does you may choose to perform a different type of action. We can modify our stored procedure to set a default value for the `@AllowContactByPhone` parameter.

```
CREATE PROCEDURE dbo.InsertContact
(
@FirstName VARCHAR(40),
@LastName  VARCHAR(40),
@DateOfBirth DATE,
@AllowContactByPhone     BIT = 0
)
```

If you save and run this code, you can now execute the stored procedure without specifying the `@AllowContactByPhone` parameter (obviously, if you want to set the `AllowContactByPhone` column to 1, you need to specify the parameter).

The first result set in Figure 18-7 shows `ContactId` 29. If you look at the second result set, which comes from the `SELECT` statement at the bottom of the query editor, you can see `AllowContactByPhone` has been set to 0 for `ContactId` 29.


SQLQuery2.sql - BE...ikemcquillan (53))*
27 - Create InsertC...mikemcquillan (52))*
100 %
Results
Messages

```
USE AddressBook;

EXEC dbo.InsertContact
    @FirstName = 'Sherlock',
    @LastName = 'Holmes',
    @DateOfBirth = '1961-09-30';

SELECT TOP (5) * FROM dbo.Contacts ORDER BY ContactId DESC;
```

	ContactId
1	29

	ContactId	FirstName	LastName	DateOfBirth	AllowContactByPhone	CreatedDate
1	29	Sherlock	Holmes	1961-09-30	0	2015-05-16 01:29:45.897
2	28	Willie	Thome	1954-03-04	0	2015-05-16 01:27:15.273
3	27	Andy	Murray	1987-05-15	1	2015-05-16 01:24:34.807
4	26	Michael	Stipe	1960-01-04	0	2015-05-16 01:21:22.237
5	25	Joe	Beasley	1959-05-09	1	2015-05-16 01:19:23.377

Figure 18-7. *Running the procedure without optional parameter value*

Output Parameters

Input parameters are what you normally use when building stored procedures, but it is common to come across stored procedures with output parameters, too. Output parameters give you a different way to return data from a stored procedure; they work similarly to a scalar function, in that the parameter holds an output value. They are more powerful than functions, though, as you can return multiple output parameters from a single stored procedure. If you don't want to return a result set from a stored procedure but you do need to return some values, output parameters are the only other way you can do it.

To specify an output parameter, add the `OUTPUT` keyword after the parameter declaration. We can change our stored procedure to pass a `@ContactId` output parameter into the procedure. This line would look like:

```
@ContactId INT OUTPUT
```

You can even specify a default value for an output parameter, meaning you don't have to pass it in. The default value needs to be entered before the `OUTPUT` keyword:

```
@ContactId INT = 0 OUTPUT
```

The procedure header should now read:

```
CREATE PROCEDURE dbo.InsertContact
(
 @FirstNameVARCHAR(40),
 @LastNameVARCHAR(40),
 @DateOfBirth DATE,
 @AllowContactByPhone BIT = 0,
 @ContactId INT = 0 OUTPUT
)
```

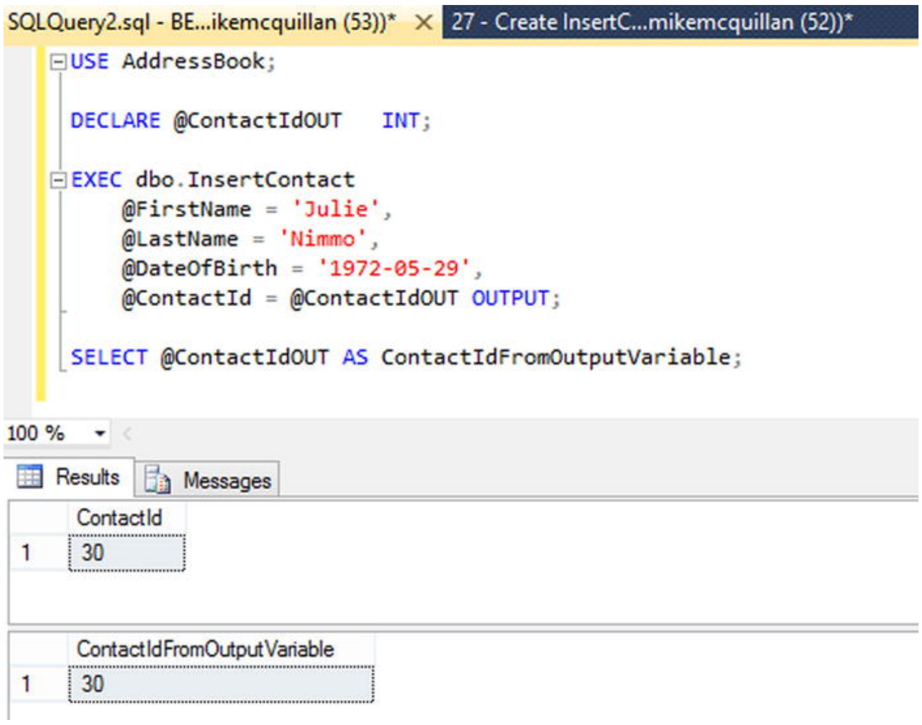
The really great thing is our stored procedure has everything in place for this output parameter to work. We need to remove the line:

```
DECLARE @ContactId     INT;
```

Once this has been removed, the stored procedure will work and the `@ContactId` output parameter will be populated! This is because this line was already present in the stored procedure:

```
SELECT @ContactId = SCOPE_IDENTITY();
```

You don't need to explicitly `RETURN` an output parameter. It just acts as a normal variable within the body of the stored procedure, so whatever the last value assigned is what will be returned within the output parameter. Run this code, return to your `EXEC` window, and change the code to match the script in Figure 18-8.


SQLQuery2.sql - BE...ikemcquillan (53))*
27 - Create InsertC...mikemcquillan (52))*
100 %
Results
Messages

```
USE AddressBook;

DECLARE @ContactIdOUT   INT;

EXEC dbo.InsertContact
    @FirstName = 'Julie',
    @LastName = 'Nimmo',
    @DateOfBirth = '1972-05-29',
    @ContactId = @ContactIdOUT OUTPUT;

SELECT @ContactIdOUT AS ContactIdFromOutputVariable;
```

```
   ContactId
1  30

   ContactIdFromOutputVariable
1  30
```

Figure 18-8. Executing the procedure and returning an output parameter

We've declared a variable called `@ContactIdOUT`, and it is this variable that is passed into the output parameter. The `OUT` is a convention we use to show that the variable is to be used for output parameter purposes. The `OUTPUT` keyword has been specified, so not only do we have to declare the `OUTPUT` keyword in the stored procedure; we also have to declare it when we call the stored procedure.

After the `EXEC` call, we run a `SELECT` statement, which outputs the contents of the `@ContactIdOUT` variable. When this has been executed, there are two result sets. Why is this?

The first result set comes from the stored procedure—it is the `SELECT` statement we added to the procedure earlier to return the newly inserted `ContactId`. The second `SELECT` statement occurs outside of the stored procedure. We have added it to demonstrate that the `@ContactIdOUT` parameter has had the inserted `ContactId` value assigned to it.

The fascinating thing here is that our stored procedure now has two ways of returning the inserted `ContactId`. It always returns the result set via the `SELECT`, but we now have the further option of returning it via an output parameter. Developers do not need to use the output parameter as it is optional, but it is there should it be needed.

Output parameters are not used very often, but they give you another one of those extra boosts that many developers I meet seem to be completely unaware of. They are especially useful if you are writing a stored procedure that will be called from other stored procedures.

READONLY Parameters

READONLY applies to table-valued parameters. Yes, you can pass in an entire table to a stored procedure! When you do this, you must specify the READONLY keyword on the parameter. Table-valued parameters cannot be modified in the procedure body; hence the need to mark them with READONLY. You'll use this parameter type in the next chapter.

Basic Debugging with PRINT

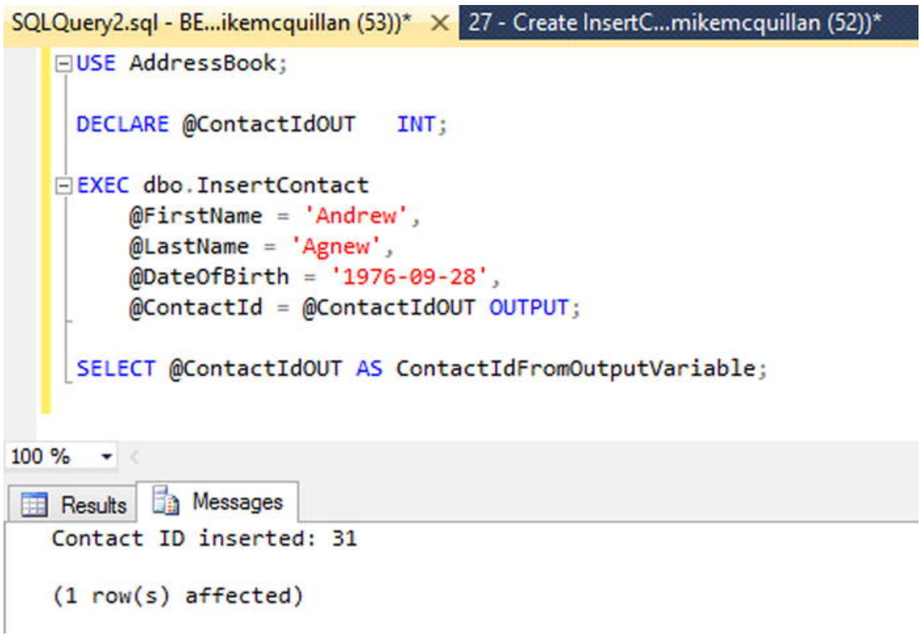
You can add PRINT statements to your stored procedures so you can see what is going on. There are far more sophisticated ways of debugging stored procedures, but PRINT offers a quick way to investigate issues (just don't forget to take the PRINT statements out when you're done!). PRINT statements are outputted to the SQL Server console when executed within Management Studio. We'll change our stored procedure to accept a new, optional parameter, @Note. This will allow us to specify a note that can be inserted into the ContactNotes table, presuming a value is provided. We'll add a PRINT statement or two so the stored procedure reports back what is going on.

```
CREATE PROCEDURE dbo.InsertContact
(
@FirstName VARCHAR(40),
@LastName VARCHAR(40),
@DateOfBirth DATE,
@AllowContactByPhone BIT = 0,
@Note VARCHAR(200) = NULL,
@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 a note if provided
IF (COALESCE(@Note, '') != '')
BEGIN
INSERT INTO dbo.ContactNotes (ContactId, Notes) VALUES (@ContactId, @Note);

PRINT 'Note inserted';
END

COMMIT TRANSACTION;

SELECT @ContactId AS ContactId;

SET NOCOUNT OFF;

END;
```

The changes we've made are not massive. We have added a new parameter, @Note. We have added this above the output parameter @ContactId. There was no need to do this, other than to follow a general convention; it is very common to see stored procedures with all input parameters declared first, followed by the output parameters.

We've added a PRINT statement to display the generated ContactId. After that, there is a block of new code. This starts by checking if the @Note variable contains a valid value; if it does, a new record is inserted into ContactNotes, using the @ContactId variable to ensure the note is linked to the correct contact when it is inserted. We also print a message to state a note was added. Run and save this code (in script 27), and then execute the stored procedure, without adding the @Note parameter. The script to execute is shown in Figure 18-9.

Once you have run this, click the **Messages** tab to view the output. Our first `PRINT` statement should be displayed.

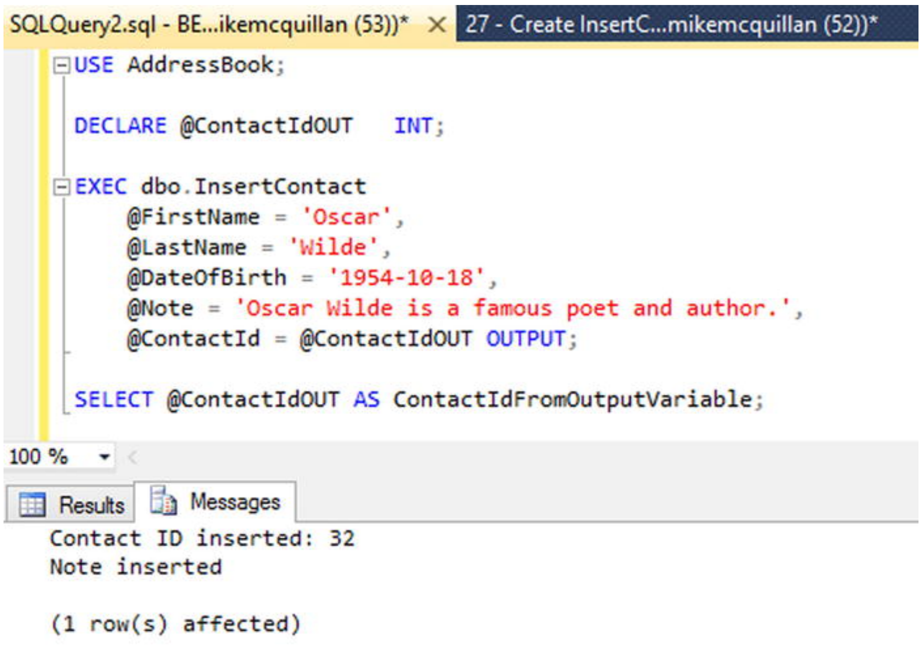

SQLQuery2.sql - BE...ikemcquillan (53))*
27 - Create InsertC...mikemcquillan (52))*
100 %
Results
Messages

```
USE AddressBook;

DECLARE @ContactIdOUT   INT;

EXEC dbo.InsertContact
     @FirstName = 'Andrew',
     @LastName = 'Agnew',
     @DateOfBirth = '1976-09-28',
     @ContactId = @ContactIdOUT OUTPUT;

SELECT @ContactIdOUT AS ContactIdFromOutputVariable;
```

```
Contact ID inserted: 31

(1 row(s) affected)
```

Figure 18-9. *Running the procedure with* `PRINT` *statements*

The **(1 row(s) affected)** message doesn't come from the stored procedure—it is generated by the `SELECT @ContactIdOUT` line. All looking good; let's add another contact, this time passing the `@Note` parameter (Figure 18-10 has the details).


SQLQuery2.sql - BE...ikemcquillan (53))*
27 - Create InsertC...mikemcquillan (52))*
100 %
Results
Messages

```
USE AddressBook;

DECLARE @ContactIdOUT   INT;

EXEC dbo.InsertContact
     @FirstName = 'Oscar',
     @LastName = 'Wilde',
     @DateOfBirth = '1954-10-18',
     @Note = 'Oscar Wilde is a famous poet and author.',
     @ContactId = @ContactIdOUT OUTPUT;

SELECT @ContactIdOUT AS ContactIdFromOutputVariable;
```

```
Contact ID inserted: 32
Note inserted

(1 row(s) affected)
```

Figure 18-10. *Running the procedure with* `PRINT` *and a note added*

And now our `PRINT` statements tell us that not only did we insert `ContactId` 32, we also added a note. I think that will do for our first stored procedure (at least, it will do for now!).

Rollback

We must create a rollback script for our stored procedure. This just contains the `IF EXISTS...DROP PROCEDURE` check.

```
USE AddressBook;

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

GO
```

Save this as `c:\temp\sqlbasics\rollback\27 - Create InsertContact Stored Procedure Rollback.sql` and add it to `00 - Rollback.sql`:

```
:setvar currentFile "27 - Create InsertContact Stored Procedure Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

Summary

We have a pretty robust stored procedure here, but we're far from done. This chapter has given us a solid introduction to stored procedures, and we've seen the various ways we can pass data into stored procedures, and how can we return data from them.

We're going to continue enhancing this stored procedure in the next chapter, so mosey on over, pardner!

PREV
Chapter 17 : Table-Valued Funct…

NEXT
Chapter 19 : Stored Procedures…