**CHAPTER 16**



## Functions

Functions were introduced in SQL Server 2000 and made an immediate impact. Functions allow you to wrap up pieces of code you use all the time into a nice, compact, reusable package. In this chapter, we'll find out what types of function SQL Server allows us to create, why functions are useful, and how we can improve our code with functions. We'll also see some of the system functions SQL Server provides. By the end of this chapter, we'll know all about scalar functions, and we'll be ready to talk about table-valued functions (TVFs) in Chapter 17. We'll be half-functional!

### Function Types

Broadly speaking, there are two types of user-defined function in SQL Server:

- **Scalar functions**: These are functions that return a standard type of value, such as an `INT` or a `VARCHAR`. An addition function that returns the sum of two numbers would be a scalar function.
- **Table-valued functions**: These are scalar functions can only return one value, of the defined type. If you need to return multiple values, you can write a TVF. These return a table containing the data you requested.

Deterministic vs. Non-Deterministic Functions

UDFs are classed as being either *deterministic* or *non-deterministic*. We met these terms when we were discussing views a couple of chapters ago. A deterministic function is one that, given a particular set of inputs, will always return the same value. Here's the addition function we were just discussing:

```
CREATE FUNCTION dbo.AddNumbers
(@Number1 INT, @Number2 INT)
RETURNS INT
AS
BEGIN

RETURN COALESCE(@Number1, 0) + COALESCE(@Number2, 0);

END;
```

This function accepts two parameters, `@Number1` and `@Number2`. It returns an `INT` value. It executes one line of code, adding `@Number1` and `@Number2` together. If a `NULL` value is passed in for either of those values, the `NULL` is defaulted to 0. This means we'll always return a 0 should two `NULL` values be passed in.

This function is deterministic because no matter how many times we call it, we'll always receive the same result back from it as long as the parameter values are the same. Here's a call to the function:

```
SELECT dbo.AddNumbers(3, 4);
```

This will always return 7, because 3 + 4 = 7. If we passed in 7 and 8, we'd always receive 15 back.

Now that we know what a deterministic function is, it should be obvious what a non-deterministic function is. That's right—a function that, no matter whether the same parameter values are passed in or not, never returns the same value. Let's take a look at a non-deterministic function:

```
CREATE FUNCTION dbo.DateWithHoursAdded
(@HoursToAdd INT)
RETURNS DATETIME
AS
BEGIN

RETURN DATEADD(HOUR, COALESCE(@HoursToAdd, 0), GETDATE());
```

```
END;
```

This does nothing more than return the current time and date, but with the number of specified hours added on. We use the system function `DATEADD` to add the hours to the current time and date. If we use this call:

```
SELECT dbo.DateWithHoursAdded(3);
```

the first time I run it, I see the result shown in Figure 16-1.

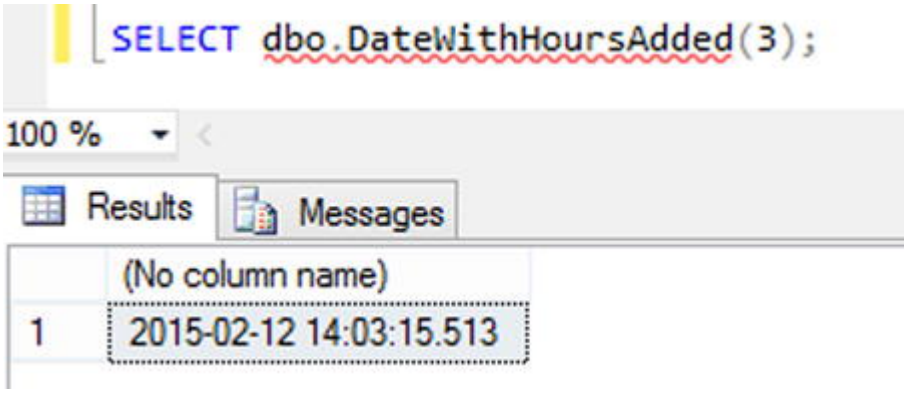



***Figure 16-1.*** *First run of the* `DateWithHoursAdded` *function*

Figure 16-2 shows the second execution, still with the same parameter value of 3.

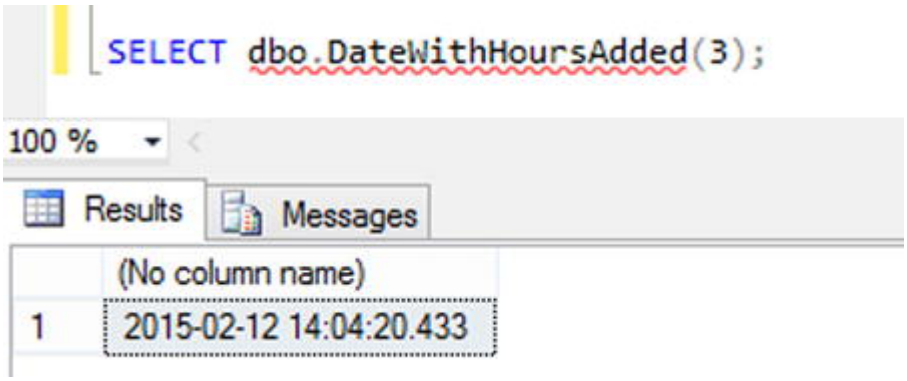



***Figure 16-2.*** *Second run of the* `DateWithHoursAdded` *function*

Because we are adding the number of hours to the current time and date, we can never return the same value when calling this function. So it's clear the function is non-deterministic.

User-Defined Functions vs. System Functions

The functions we've seen so far are user-defined functions—or UDFs—which are those we've created ourselves. SQL Server also provides a set of system functions, which are built into SQL Server and available to you out of the box. We've been using some system functions throughout the book—for instance, `GETDATE()` is a system function to return the current time and date, and `COUNT()` is a function to return a count of whatever values you pass into the function.

You can see the system functions using SSMS. In the Object Explorer, expand **Databases** ➤ **AddressBook** ➤ **Programmability** ➤ **Functions**. You'll see the four items shown in Figure 16-3.

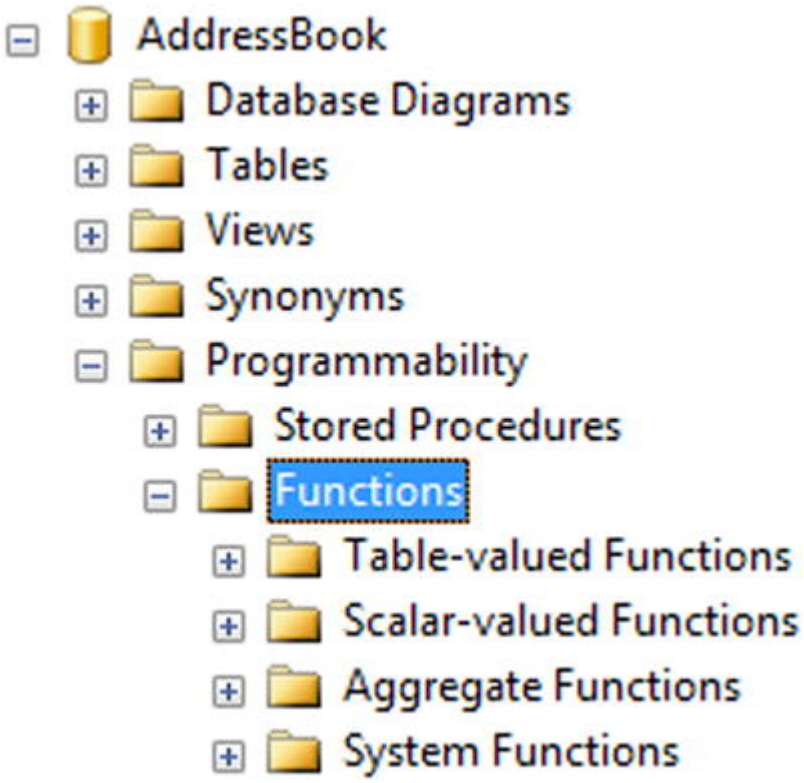



***Figure 16-3.*** *Viewing functions in SSMS*

The aforementioned TVFs and scalar-valued functions are used to house UDFs. These will be empty at the moment as we haven't actually created any UDFs. We'll see these two items again later. *Aggregate functions* are a special type of UDF that are created in .NET, allowing you to expand on the existing aggregation functions provided in SQL Server like `COUNT()` and `AVG()`. Finally, *system functions* (Figure 16-4) lists all of the functions built into SQL Server, separated by type.

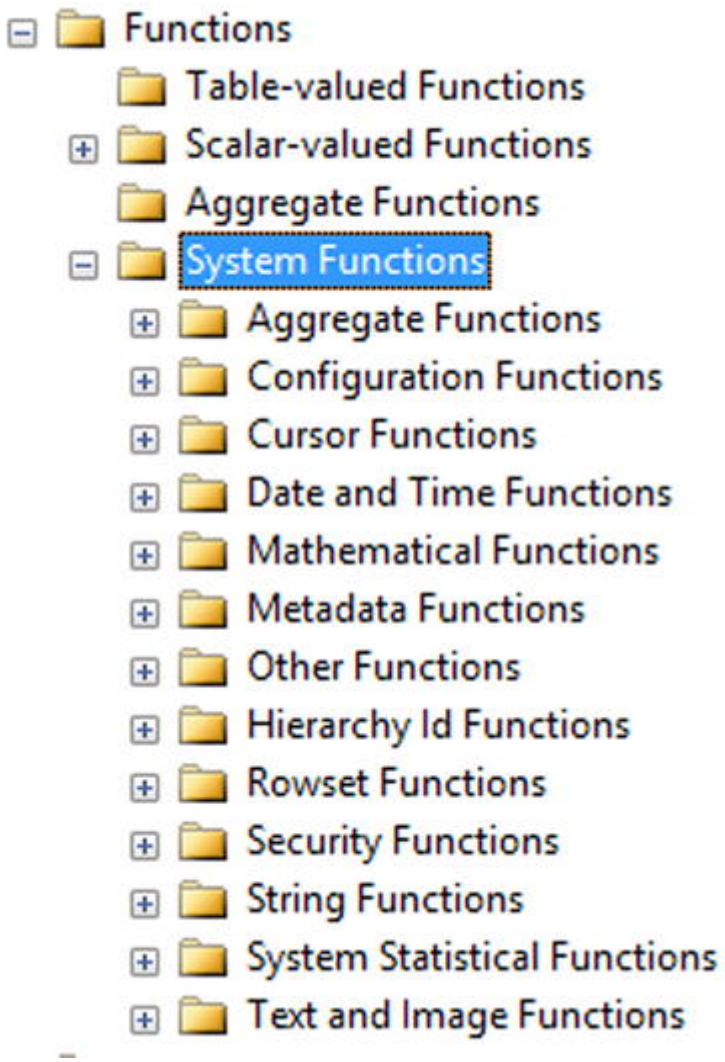



***Figure 16-4.*** *Viewing system functions in SSMS*

There are lots and lots (and lots) of system functions available. You'll find yourself using many of these in your daily work. In Figure 16-5, we're looking at the *date and time functions* available to us.

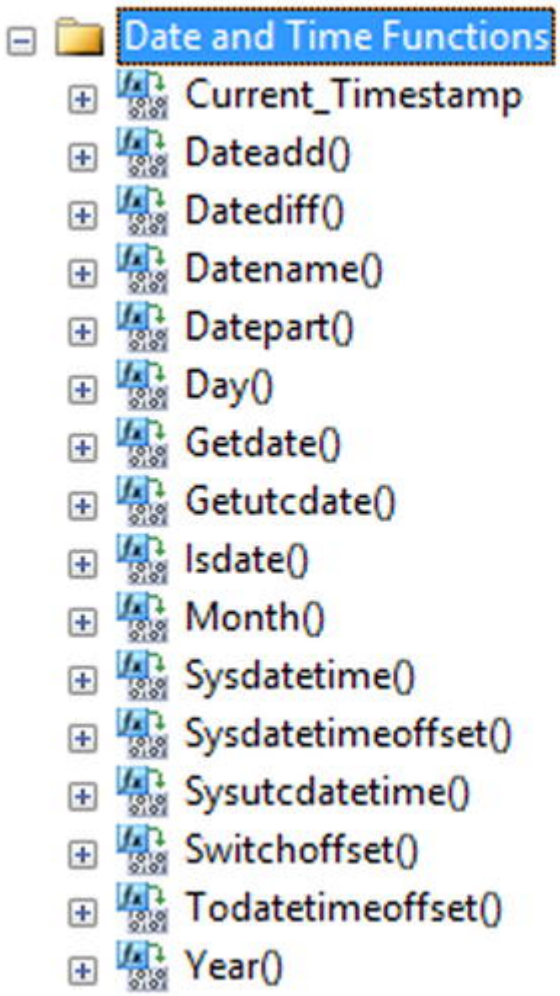



***Figure 16-5.*** *Viewing date and time system functions in SSMS*

These functions are available in any database you create. Take some time to learn what each of them does—it could save you from reinventing the wheel in the future!

**Why Are Functions Useful?**

Now that we have an idea of the types of function available to us, we're in a position to start creating functions. But why should we want to create any functions? What benefits do they give us? Let's see if we can come up with a convincing argument.

First, functions can be used as part of your `SELECT` or DML statements, or on their own. This gives them tremendous flexibility, as it allows you to use them to not only insert or update data, but to manipulate the output of data, perhaps by correctly formatting certain columns (we'll create a function just like this soon).

Functions can also be used to set default column values or to implement `CHECK` constraints. You can even create a column in a table that uses a function to obtain its value, by creating something called a *computed column*.

One other thing I like to use functions for is to enhance views, by replacing certain columns with function calls. Bear in mind this could have a negative effect on indexed views or views you wish to index, as only deterministic functions can be used in indexed views.

The short answer is that for many problems functions are fantastic, and they should be part of your SQL programming strategy. We'll go ahead and enhance our **AddressBook** database with some functions now, and we'll be using them throughout the rest of our book.

**What Can't I Do with A Function?**

We've seen why functions are great, but surely there are some downsides? Not really, but there are some limitations. The big one is that a function cannot alter the database in any way, shape, or form. You cannot create temp tables in functions, you cannot run DML statements against real tables, and you cannot run anything that causes a side effect. A side effect is deemed to be any piece of functionality that affects anything outside of the function.

As we create our functions we'll look at some possible side effects and how they affect the function.

**A First Function**

Open up a New Query Window and run this `SELECT` statement (see Figure 16-6).

```
SELECT * FROM dbo.Contacts;
```

SQLQuery1.sql - BE...ikemcquillan (54))*

```
USE AddressBook;

SELECT * FROM dbo.Contacts;
```

100 %

Results | Messages

| | ContactId | FirstName | LastName | DateOfBirth | AllowContactByPhone | CreatedDate |
|---|---|---|---|---|---|---|
| 1 | 1 | Stephen | Gerrard | 1980-05-30 | 1 | 2015-02-11 14:05:00.590 |
| 2 | 2 | Dennis | Potter | 1935-05-17 | 0 | 2015-02-11 14:05:00.590 |
| 3 | 3 | Richard | Adams | 1920-05-09 | 0 | 2015-02-11 14:05:00.590 |
| 4 | 4 | Bertie | McQuillan | 2001-06-30 | 1 | 2015-02-11 14:05:00.590 |
| 5 | 5 | Walt | Disney | 1966-12-05 | 1 | 2015-02-11 14:05:00.590 |
| 6 | 6 | Barbara | Gordon | 1952-01-11 | 0 | 2015-02-11 14:05:00.590 |
| 7 | 7 | Josephine | Bailey | 1949-05-31 | 1 | 2015-02-11 14:05:00.590 |
| 8 | 8 | Linda | Canoglu | 1959-07-11 | 1 | 2015-02-11 14:05:00.590 |
| 9 | 9 | Grace | McQuillan | 1993-09-27 | 0 | 2015-02-11 14:05:00.590 |
| 10 | 10 | Vera | Black | 1984-08-03 | 0 | 2015-02-11 14:05:00.590 |
| 11 | 11 | Angelica | Jones | 1981-02-04 | 1 | 2015-02-11 14:05:00.590 |
| 12 | 12 | Steve | Davis | 1957-08-22 | 1 | 2015-02-11 14:05:00.590 |
| 13 | 13 | Allison | Fisher | 1968-02-24 | 1 | 2015-02-11 14:05:00.590 |
| 14 | 14 | julius | Marx | 1990-10-02 | 0 | 2015-02-11 14:05:00.590 |
| 15 | 15 | george | formby | 1944-05-26 | 1 | 2015-02-11 14:05:00.590 |
| 16 | 16 | Alan | Partridge | 1965-04-14 | 0 | 2015-02-11 14:05:00.590 |
| 17 | 17 | Harper | Lee | 1986-04-28 | 1 | 2015-02-11 14:05:00.590 |
| 18 | 18 | Robert | Burns | 1959-01-25 | 0 | 2015-02-11 14:05:00.590 |
| 19 | 19 | Michael | Jackson | 1967-06-30 | 0 | 2015-02-11 14:05:00.590 |
| 20 | 20 | Roald | Dahl | 1916-09-13 | 0 | 2015-02-11 14:05:00.670 |
| 21 | 22 | Laura | Robson | 1994-01-21 | 1 | 2015-02-11 15:17:56.870 |
| 22 | 24 | Bryan | Ferry | 1945-09-26 | 0 | 2015-02-11 21:53:15.343 |

***Figure 16-6.*** *A simple `SELECT` statement*

Can you see any problems with any of the data returned by the `SELECT` in Figure 16-6? Look at rows 14 and 15. The first name in row 14 is not capitalized, and neither the first name nor the last name is capitalized in row 15. We can write a function to ensure the names are always displayed in a capitalized manner. We can also write the function so it returns the name as a single value, rather than in separate columns.

The CREATE FUNCTION Statement

The statement used to create a function will look familiar to you—it's a typical `CREATE` statement, in this case `CREATE FUNCTION`. Here's the basic definition; for the full definition visit `https://msdn.microsoft.com/en-us/library/ms186755.aspx`:

```
CREATE FUNCTION SchemaName.FunctionName
(Parameters (Optional))
RETURNS ReturnType
AS
BEGIN

Do something...

RETURN ReturnType

END;
```

A function must always be defined with a schema name. You must provide `dbo` if you haven't created any schemas of your own. Next, you can specify parameters to pass into the function. These are values that the

function uses to derive a result. The two functions we saw earlier in this chapter both accepted parameters, but you don't have to specify any if your function doesn't need them (parameters can be set as optional).

To complete the outline of the function, you have to tell SQL Server what the function is going to return. The `RETURN` keyword can be used for this. Any valid SQL Server data type can be returned, or even a table. If you're quite advanced you can create your own custom data types in SQL Server, and these can be specified as the function return type (you will see how to create custom data types in Chapter 19).

Finally, we come to the body of the function. This starts with `AS BEGIN`. You can put as many lines of code as you want within the `BEGIN` and `END` block. The last line must be a `RETURN` statement. If you are writing a TVF, you just write `RETURN`. If the function returns a scalar value, that scalar value must be specified in the `RETURN` statement.

A Scalar Function: ContactName

Our function is going to return a correctly formatted contact name, as one value. Plotting this out, we can see we need to:

- Accept `FirstName` and `LastName` values as parameters
- Return a `VARCHAR`, holding the full name
- Write some code to correctly format the first letter of both the first name and the last name, then return both values as a single string value

Seems easy enough. Here we go with the definition:

```
USE AddressBook;

GO

CREATE FUNCTION dbo.ContactName
(@FirstName VARCHAR(40), @LastName VARCHAR(40))
RETURNS VARCHAR(80)
AS
BEGIN

DECLARE @FullName VARCHAR(80);

-- Capitalise the first letter of the first name and last name
SELECT @FirstName = UPPER(LEFT(@FirstName, 1)) + RIGHT(@FirstName, LEN(@FirstName) - 1), @LastName

SELECT @FullName = @FirstName + ' ' + @LastName;

RETURN @FullName;

END;

GO
```

Note the `GO` between the `USE` and `CREATE FUNCTION` statements. This is needed as `CREATE FUNCTION` has to be the first statement in a batch (just like `CREATE VIEW`).

The function is called `dbo.ContactName`, and accepts two parameters, for first name and last name. These are both `VARCHAR(40)`, which is the same type as the corresponding columns in the `Contacts` table. Parameters must always begin with an `@` symbol. The function returns a value of type `VARCHAR(80)`. This is to cover the two merged first name and last name values: 40 + 40 = 80.

Now we come to the actual code. We start by declaring a variable called `@FullName`, of `VARCHAR(80)`. This will contain the value to be returned by the function. Next up is the capitalization code. The line that performs this process contains some things we haven't seen before, so it's worth taking a closer look:

```
SELECT @FirstName = UPPER(LEFT(@FirstName, 1)) + RIGHT(@FirstName, LEN(@FirstName) - 1), @LastName
```

The code that capitalizes the first letter of `@FirstName` and `@LastName` is exactly the same, so we'll just look at `@FirstName`. The first part is:

```
UPPER(LEFT(@FirstName, 1))
```

Assume a name of "mcquillan". The `UPPER` function capitalizes whatever string is passed into it. If "mcquillan" was passed in, "MCQUILLAN" would be returned. We are not passing in "mcquillan" though, we are passing:

```
LEFT(@FirstName, 1)
```

The `LEFT` function returns the specified number of characters from a string, starting from the left. We've told SQL Server we want 1 character from the `@FirstName` variable. This means return me the first character of the string. So a value of "mcquillan" would cause "m" to be returned. The `UPPER` function would then take this "m" and convert it to "M". So this first call has given us a capital letter.

Now we have a plus sign, which dictates that the "M" we already have will be added to whatever the code after the plus sign returns. That code is:

```
RIGHT(@FirstName, LEN(@FirstName) - 1)
```

No prizes for guessing that `RIGHT` is the opposite of `LEFT`. `LEFT` returns characters from the start of a string, and `RIGHT` returns characters from the end of the string. For `ContactName`, we want to return every character in the string—*except for the first character, which we have already processed*. The declaration of `RIGHT` is exactly the same as `LEFT`—we provide the string we want to extract characters from, and then tell SQL Server how many characters we want.

Our call to `RIGHT` is a bit different from `LEFT`—we knew when we were calling `LEFT` that we only needed the first character, so we could just specify 1 as the number of characters required. We have to calculate how many characters the call to `RIGHT` will return, as we have no idea how many characters the name passed in will contain. If we pass in "mcquillan", we want to return "cquillan". If we pass in "smith", we want to return "mith".

The `LEN` function comes to our rescue here. This handy function returns the length of any string you pass in to it. `LEN('bertie')` would return 6, but `LEN('Grace')` would return 5. Our call is:

```
LEN(@FirstName) - 1
```

This will return the entire length of the string minus 1. So passing "mcquillan" will return 8, not 9. `LEN` will initially return 9, the correct length, but we are then taking 1 off the value returned by `LEN`. This gives us the exact number of characters we want. The `RIGHT` call will return "cquillan" when "mcquillan" is passed. This is then added to the "M" our `LEFT` call generated, giving us a full value of "Mcquillan". And we've ended up with a capitalized name!

Once the capitalization process has completed, we just set `@FullName` to `@FirstName` and `@LastName`, separated by a space. Our last act is to `RETURN` the contents of `@FullName` as the function result.

That's not a bad little function. Run the script to create the function, ensuring **Command(s) completed successfully** appears.

Testing Our Function

It's very easy to test a function. Open a New Query Window and write a `SELECT` statement to call the function with some appropriate values (see Figure 16-7).

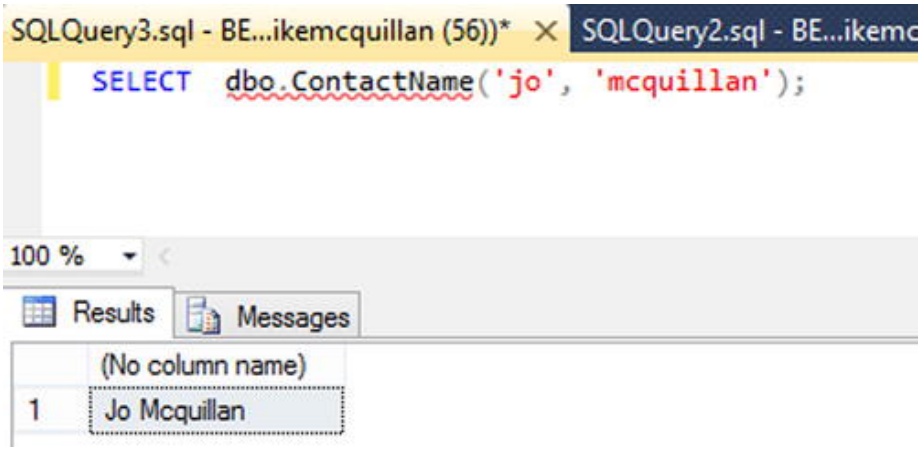



***Figure 16-7.*** *Testing a function*

Hurrah, Figure 16-7 shows us a nicely capitalized name! (It also shows us an underlined function name—remember, Ctrl+R will refresh Intellisense.) We can call the function multiple times from the same `SELECT`, too, as demonstrated in Figure 16-8.

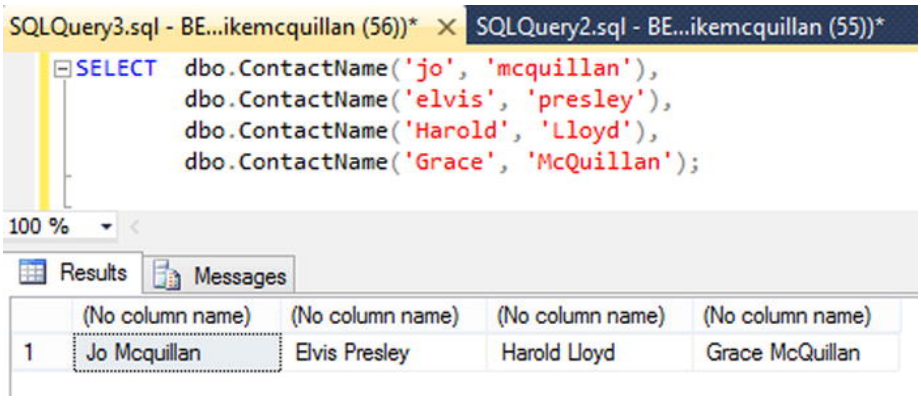



***Figure 16-8.*** *Lots of function calls from one* `SELECT`

The function is working well. Even if we pass a name already capitalized to it, the name still returns as capitalized. And it doesn't remove any capitals specified anywhere else in the string.

Now, what if we have imperfect data? Say we only pass a first name, or only pass a last name? What would happen if we just passed blank strings? Let's give it a go with a `NULL` value provided in the surname. Figure 16-9 has the result.

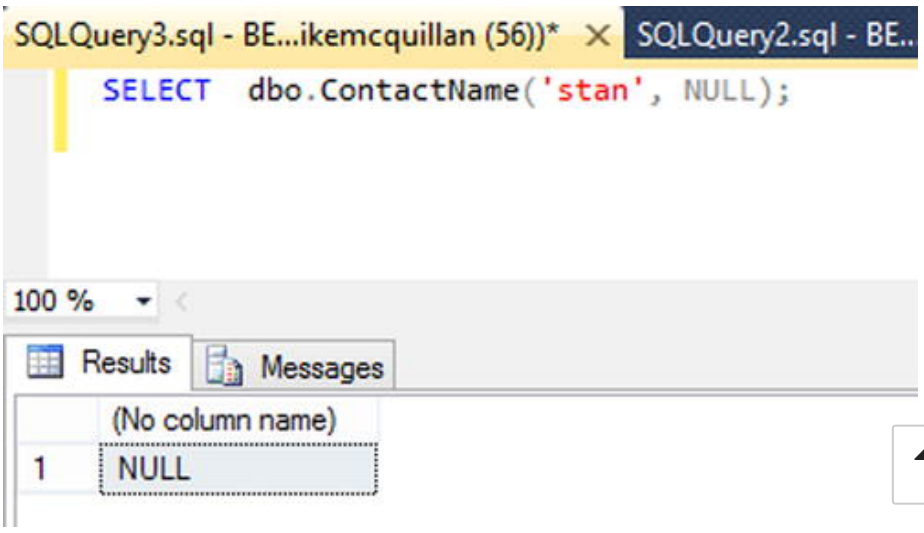


***Figure 16-9.*** *Problems with dodgy data*

Rats, we have a problem. Let's try a blank value (Figure 16-10).

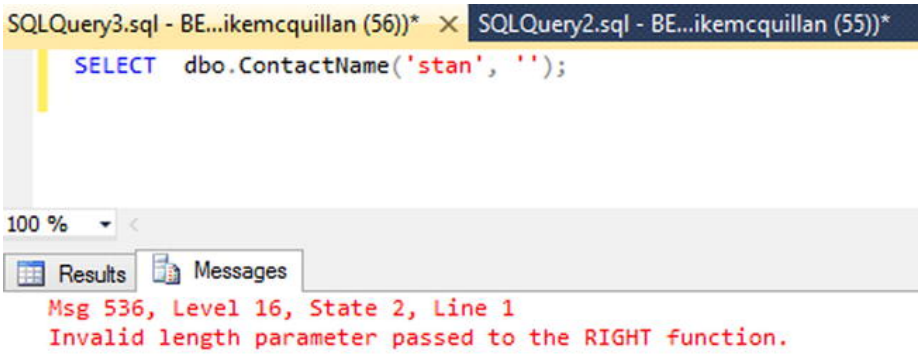



***Figure 16-10.*** *Blank values fail, too*

Figure 16-10 tells us blank values don't work, either. We need to modify the code to prevent these problems. We'll just add a line into the function so we can see what `@FirstName` and `@LastName` contain before the capitalization statement runs. They should contain exactly what we passed in. Return to the window housing the function's code, and under the line:

```
DECLARE @FullName VARCHAR(80);
```

add this `SELECT`:

```
SELECT @FirstName AS FirstName, @LastName AS LastName;
```

Run this. The function doesn't build! Instead, SQL Server tells us off. The error message is shown in Figure 16-11.

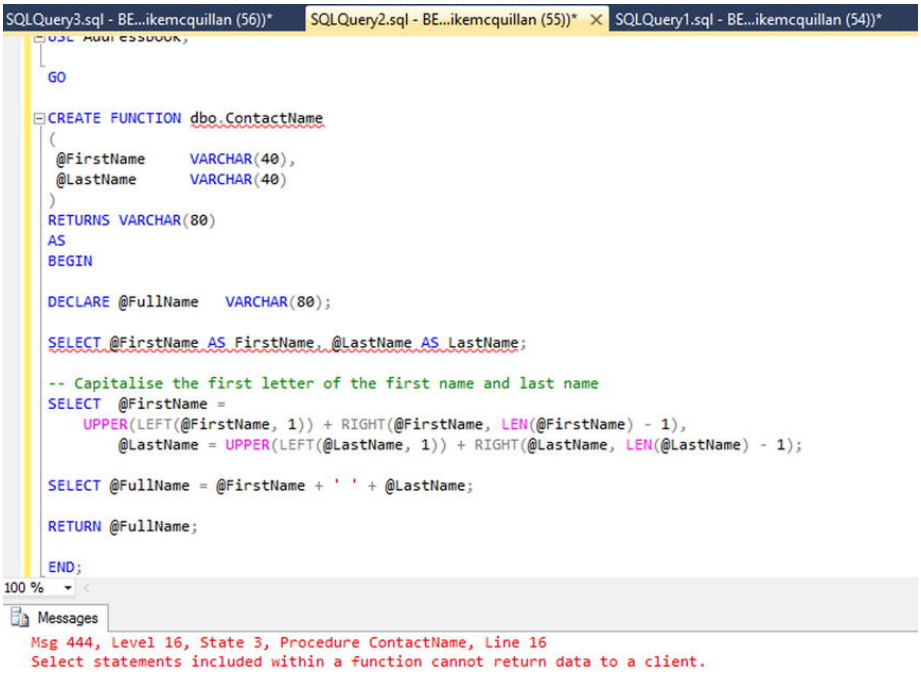



***Figure 16-11.*** *Trying to include a side effect in a function*

Side Effects

We have just created a *side effect*. We've tried to make the function do something in addition to its main purpose. The purpose of the function is to capitalize two strings and return them as one complete string. By adding the `SELECT` statement, we've tried to tell the function to return some data as well. If this safety mechanism wasn't in place, the `SELECT` statement could interfere with other code we write whenever the function is called, which would lead to all sorts of confusion.

You can try changing the `SELECT` to a `PRINT`, but it won't do any good—a `PRINT` is still a side effect. Keep the functions concise and to the point and side effects won't bother you.

Further Testing

We still need to change our function to ensure correct values are returned if `NULL` or empty strings are passed. Time to fall back to testing mechanism number 2! I use this technique a lot; I find it saves me a lot of development time. It involves removing the `CREATE FUNCTION` call and executing the function's code as a stand-alone script instead. This way, I can add as many `SELECT` and `PRINT` statements as I like. Once the code is behaving the way I expect, I can remove the side effect statements and restore the `CREATE FUNCTION` call.

Return to the window containing the function code and change it so we can run the code independently.

```
USE AddressBook;

GO

--CREATE FUNCTION dbo.ContactName
--(
DECLARE @FirstName VARCHAR(40),
@LastName VARCHAR(40)
--)
--RETURNS VARCHAR(80)
--AS
--BEGIN

SELECT @FirstName = 'stan', @LastName = NULL;

DECLARE @FullName VARCHAR(80);

-- Capitalise the first letter of the first name and last name
SELECT @FirstName = UPPER(LEFT(@FirstName, 1)) + RIGHT(@FirstName, LEN(@FirstName) - 1), @LastName

SELECT @FullName = @FirstName + ' ' + @LastName;

--RETURN @FullName;
SELECT @FullName;

--END;

GO
```

We've commented out all of the lines related to `CREATE FUNCTION`—parentheses, `BEGIN`, `END`, the lot. We've also put a `DECLARE` in front of the parameter names, changing them to normal variables. After the `DECLARE`, the first line now sets `@FirstName` and `@LastName` to `'stan'` and `NULL`, respectively. This should allow us to fix our first bug. Our final change is to comment out the `RETURN` line and replace it with a `SELECT`—`RETURN` will only work when called from the function.

Press F5 to run this and one result set appears, showing the `NULL` value we were seeing earlier (**Figure 16-12**). If we pass in a valid first name and a `NULL` last name (or vice versa), then just the name we've provided should be returned (certainly not a `NULL`!).

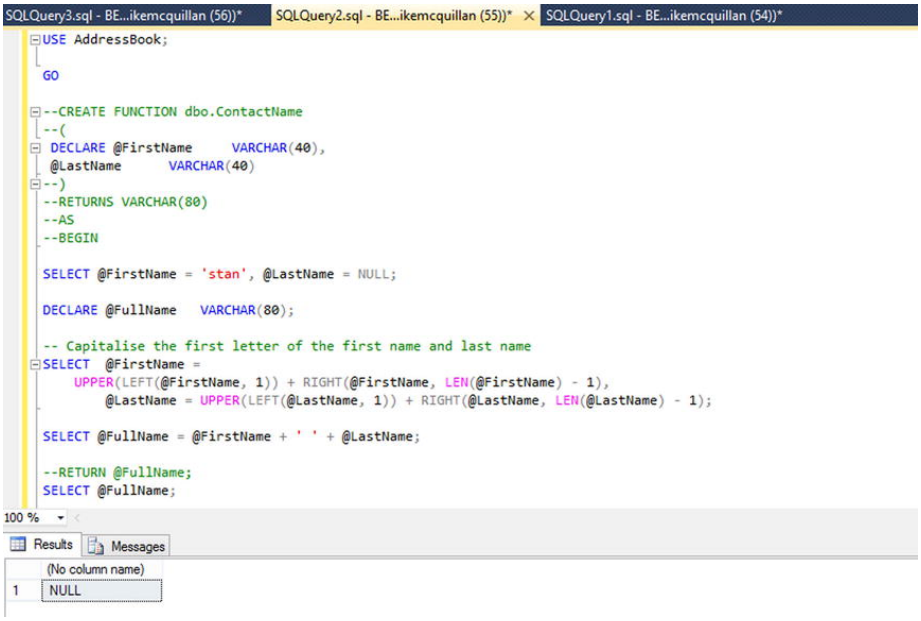



***Figure 16-12.*** *Testing the function code manually*

To fix this problem, we need a `NULL` check. Change the code as shown in bold so we convert any `NULL` values to empty strings before running the capitalization statement:

```
USE AddressBook;

GO

--CREATE FUNCTION dbo.ContactName
--(
DECLARE @FirstName VARCHAR(40),
@LastName VARCHAR(40)
--)
--RETURNS VARCHAR(80)
--AS
--BEGIN

SELECT @FirstName = 'stan', @LastName = NULL;

DECLARE @FullName VARCHAR(80);

-- Replace NULL values with empty strings
SELECT @FirstName = COALESCE(@FirstName, ''),
@LastName = COALESCE(@LastName, '');

-- Capitalise the first letter of the first name and last name
SELECT @FirstName = UPPER(LEFT(@FirstName, 1)) + RIGHT(@FirstName, LEN(@FirstName) - 1), @LastName

SELECT @FullName = @FirstName + ' ' + @LastName;

--RETURN @FullName;
SELECT @FullName;

--END;

GO
```

Run this. Still no success—this time we see the **Invalid length parameter passed to the RIGHT function** message we saw earlier (Figure 16-13). We've solved one problem, but not the second (yet!).

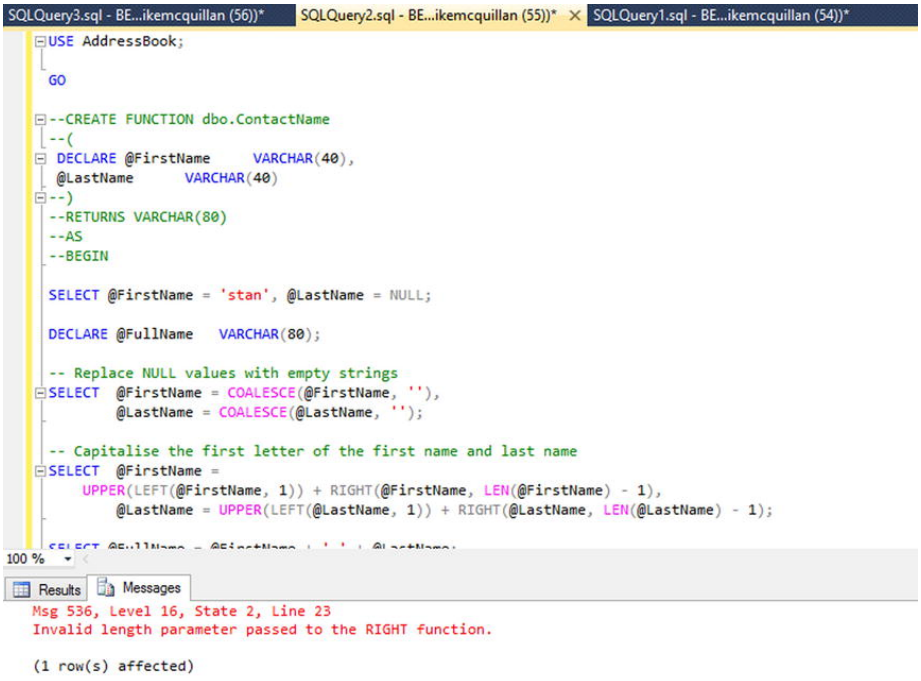



***Figure 16-13.*** *Fixing bugs in the function code*

This issue is being raised because we are trying to take 1 character away from a string of zero length; 0 – 1 = –1, which the `RIGHT` function refuses to return. Fair enough. We can work around this with a `CASE` statement. If the length of the string is greater than 0, we can execute the capitalization code. If it isn't we won't do anything to the string. Here is the updated code:

```
USE AddressBook;

GO

--CREATE FUNCTION dbo.ContactName
--(
DECLARE @FirstName VARCHAR(40),
@LastName VARCHAR(40)
--)
--RETURNS VARCHAR(80)
--AS
--BEGIN

SELECT @FirstName = 'stan', @LastName = '';

DECLARE @FullName VARCHAR(80);

SELECT @FirstName = COALESCE(@FirstName, ''),
@LastName = COALESCE(@LastName, '');

-- Capitalise the first letter of the first name and last name
SELECT @FirstName =
CASE WHEN LEN(@FirstName) > 0 THEN UPPER(LEFT(@FirstName, 1)) + RIGHT(@FirstName, LEN(@FirstName) - 1) ELSE
@LastName = CASE WHEN LEN(@LastName) > 0 THEN UPPER(LEFT(@LastName, 1)) + RIGHT(@LastName, LEN(@LastName) -
END;

SELECT @FullName = @FirstName + ' ' + @LastName;

--RETURN @FullName;
SELECT @FullName;

--END;

GO
```

Run this and you'll see what success looks like. It looks like Figure 16-14.

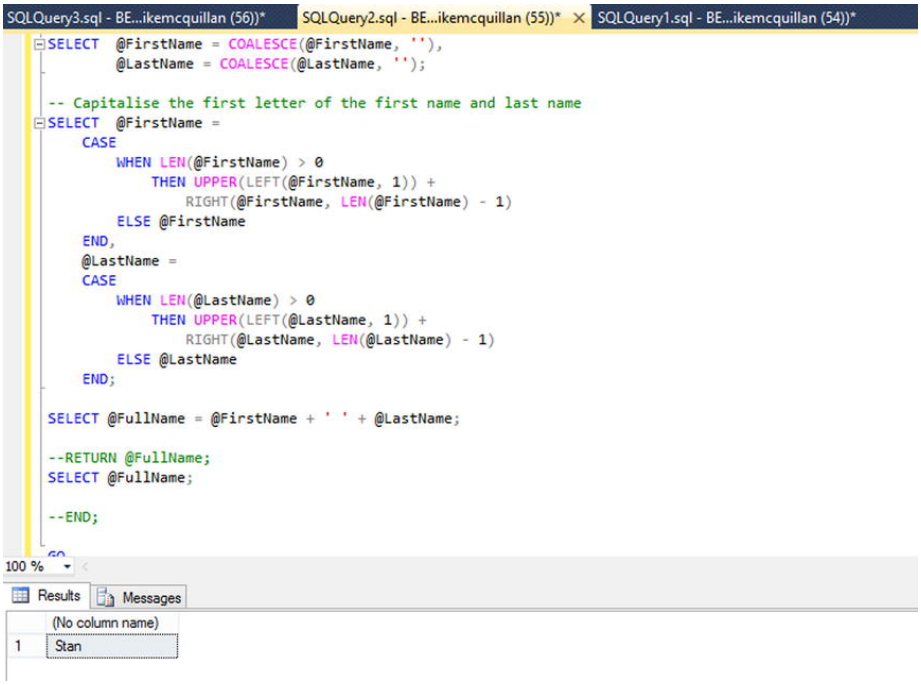



***Figure 16-14.*** *Stan returns!*

This is more like it. Test it with a few other combinations (good and bad) to make sure you are happy with the code. Once you are satisfied we can change the code back to a function. Restore the lines we commented out and remove the bits we added, like `DECLARE`. Here's the full listing.

```
USE AddressBook;

GO

CREATE FUNCTION dbo.ContactName
(@FirstName VARCHAR(40), @LastName VARCHAR(40)
)
RETURNS VARCHAR(80)
AS
BEGIN

DECLARE @FullName VARCHAR(80);

-- Replace NULL values with empty strings
SELECT @FirstName = COALESCE(@FirstName, ''), @LastName = COALESCE(@LastName, '');

-- Capitalise the first letter of the first name and last name
SELECT @FirstName =
CASE WHEN LEN(@FirstName) > 0 THEN UPPER(LEFT(@FirstName, 1)) + RIGHT(@FirstName, LEN(@FirstName) -
@LastName = CASE WHEN LEN(@LastName) > 0 THEN UPPER(LEFT(@LastName, 1)) + RIGHT(@LastName, LEN(@Las
END;

SELECT @FullName = @FirstName + ' ' + @LastName;

RETURN @FullName;

END;

GO
```

Run this and . . . it doesn't work! Instead, we see the error message displayed in Figure 16-15:

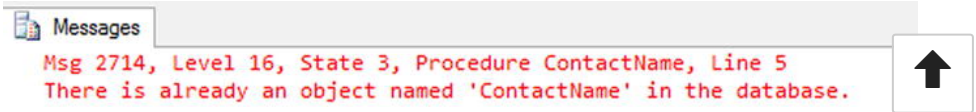


***Figure 16-15.*** *Error message: function already exists*

We could change `CREATE FUNCTION` to `ALTER FUNCTION`, which allows us to modify an existing function without dropping it. But that won't help us when we save the function as part of our script collection, so we'll drop it first, then re-create it.

### Dropping Functions

We need to check if our function exists, drop the function if it does, and then re-create it. You won't be surprised to learn that the statement used to drop a function is called `DROP FUNCTION`. What may surprise you is we cannot run a `SELECT` against `sys.functions` to check if the function exists—because there is no `sys.functions`. Instead, we have to query the `sys.objects` table instead, specifying a type of `FN` (short for `FUNCTION`).

Add this check between the lines `USE AddressBook;` and `GO` at the top of the script.

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.objects WHERE [name] = 'ContactName' AND [type] = 'FN')
BEGIN
DROP FUNCTION dbo.ContactName;
END;

GO
```

Run the script again and it will now execute as expected. Now we'll run a test query. If you still have the test script we used earlier, modify it to match the code in Figure 16-16; otherwise just type it into a New Query Window. Now two empty strings return an empty string, and so does a `NULL`.

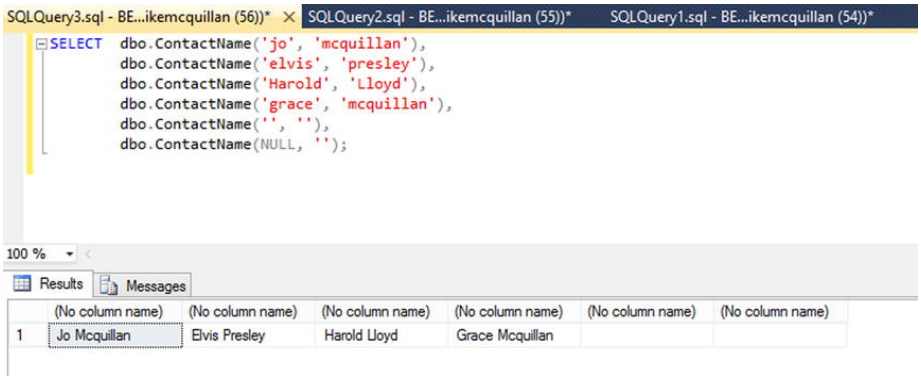



***Figure 16-16.*** *Empty string inputs return an empty string!*

### Saving the Function

Our function is now ready for use. Save it as `c:\temp\sqlbasics\apply\24 - Create ContactName Function.sql`. Add it to our growing `00 - Apply.sql` script, too.

```
:setvar currentFile "24 - Create ContactName Function.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

We also need to create a rollback for the function. In a New Query Window, paste the `DROP FUNCTION` statement we wrote earlier.

```
USE AddressBook;

IF EXISTS (SELECT 1 FROM sys.objects WHERE [name] = 'ContactName' AND [type] = 'FN')
BEGIN
DROP FUNCTION dbo.ContactName;
END;

GO
```

Save this as `c:\temp\sqlbasics\rollback\24 - Create ContactName Function Rollback.sql`. Add it to the `00 - Rollback.sql script`.

```
:setvar currentFile "24 - Create ContactName Function Rollback.sql"
PRINT 'Executing $(path)$(currentFile)';
:r $(path)$(currentFile)
```

### Using the Function in Queries

To finish off this chapter, we'll take a look at how to use our scalar function as part of a query. Do you remember the query we wrote at the start of this chapter? It was the simple `SELECT` you can see in Figure 16-17.

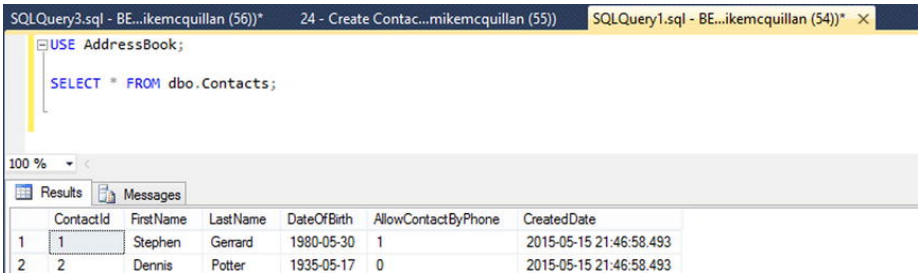



***Figure 16-17.*** *A simple `SELECT` statement (again)*

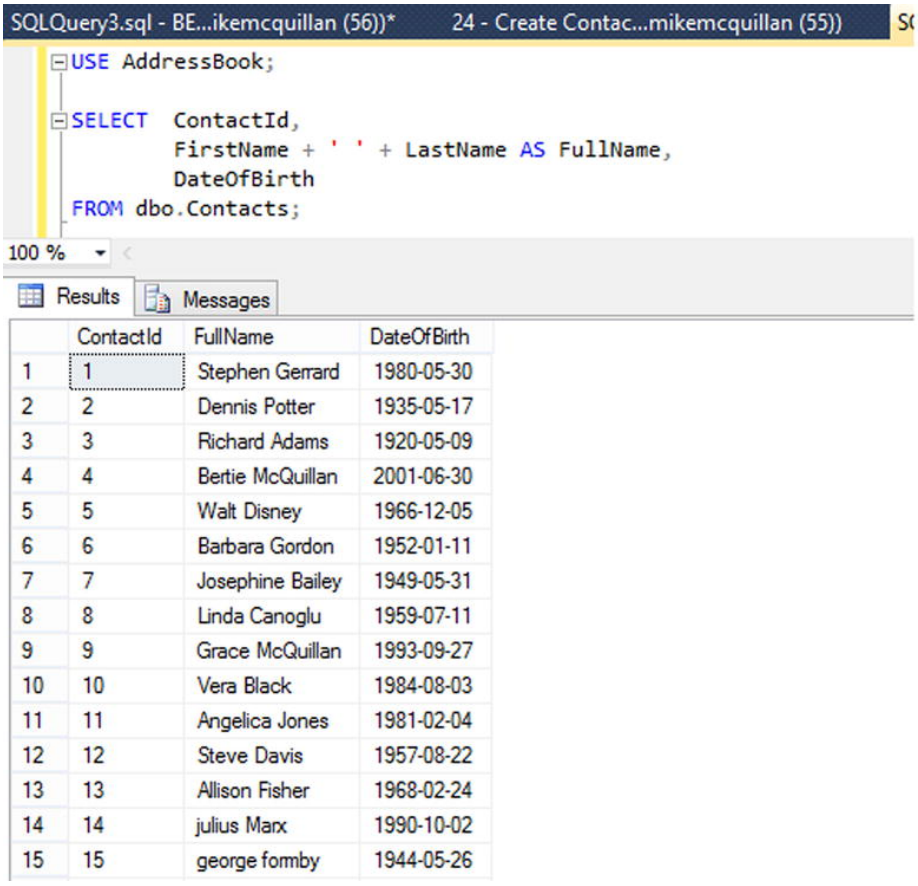
Let's say we want to return a list of contacts with the columns:

- `ContactId`
- Full name of the contact (`FirstName` and `LastName` joined up)
- `DateOfBirth`

We could write this as:

```
SELECT ContactId, FirstName + ' ' + LastName AS FullName, DateOfBirth
FROM dbo.Contacts;
```

This kind of gives us what we want (Figure 16-18):



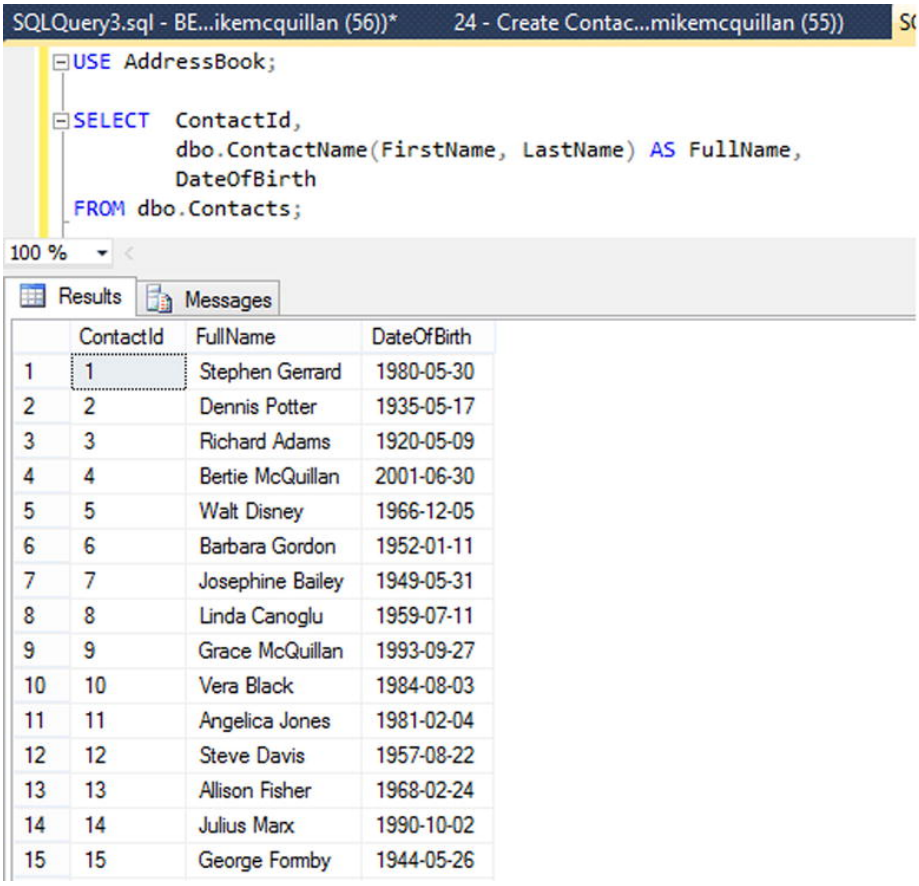

| | ContactId | FullName | DateOfBirth |
|---|---|---|---|
| 1 | 1 | Stephen Gerrard | 1980-05-30 |
| 2 | 2 | Dennis Potter | 1935-05-17 |
| 3 | 3 | Richard Adams | 1920-05-09 |
| 4 | 4 | Bertie McQuillan | 2001-06-30 |
| 5 | 5 | Walt Disney | 1966-12-05 |
| 6 | 6 | Barbara Gordon | 1952-01-11 |
| 7 | 7 | Josephine Bailey | 1949-05-31 |
| 8 | 8 | Linda Canoglu | 1959-07-11 |
| 9 | 9 | Grace McQuillan | 1993-09-27 |
| 10 | 10 | Vera Black | 1984-08-03 |
| 11 | 11 | Angelica Jones | 1981-02-04 |
| 12 | 12 | Steve Davis | 1957-08-22 |
| 13 | 13 | Allison Fisher | 1968-02-24 |
| 14 | 14 | julius Marx | 1990-10-02 |
| 15 | 15 | george formby | 1944-05-26 |

***Figure 16-18.*** *Manually concatenating the first and last names*

It hasn't capitalized the names of rows 14 and 15, though. And it isn't very reusable, either—we'll have to write that line of code whenever somebody wants a full name instead of separate first and last names. Not to worry—we can fix those problems by using our function. Here's the statement updated to use the `ContactName` function.

```
SELECT ContactId, dbo.ContactName(FirstName, LastName) AS FullName, DateOfBirth
FROM dbo.Contacts;
```

That's a bit easier to write! And it returns the data exactly as we want it too, as Figure 16-19 shows us.




| | ContactId | FullName | DateOfBirth |
|---|---|---|---|
| 1 | 1 | Stephen Gerrard | 1980-05-30 |
| 2 | 2 | Dennis Potter | 1935-05-17 |
| 3 | 3 | Richard Adams | 1920-05-09 |
| 4 | 4 | Bertie McQuillan | 2001-06-30 |
| 5 | 5 | Walt Disney | 1966-12-05 |
| 6 | 6 | Barbara Gordon | 1952-01-11 |
| 7 | 7 | Josephine Bailey | 1949-05-31 |
| 8 | 8 | Linda Canoglu | 1959-07-11 |
| 9 | 9 | Grace McQuillan | 1993-09-27 |
| 10 | 10 | Vera Black | 1984-08-03 |
| 11 | 11 | Angelica Jones | 1981-02-04 |
| 12 | 12 | Steve Davis | 1957-08-22 |
| 13 | 13 | Allison Fisher | 1968-02-24 |
| 14 | 14 | Julius Marx | 1990-10-02 |
| 15 | 15 | George Formby | 1944-05-26 |

***Figure 16-19.*** *Using the function to return a full name*

Marvelous. Now we never have to worry about anybody asking us to return the contact's full name in the future—we have a reusable function we can call as and when needed. Well done, you!

**Summary**

We've covered scalar functions in this chapter, and we've seen not only how to create them, but also how to test them, drop them, and integrate them into queries. It's pretty powerful stuff!

Scalar functions are only one-half of the functional story, though. We introduced table-valued functions at the start of the chapter, and they add another whole level of power to the functional toolkit. Stroll on to the next chapter and we'll talk TVFs.

PREV
Chapter 15 : Transactions
NEXT
Chapter 17 : Table-Valued Funct…

© 2017 Safari. Terms of Service / Privacy Policy