LAB 02. T/SQL PROGRAMMING

Before practice, install Adventureworks database based on the following instruction:
https://www.tutorialgateway.org/download-and-install-adventureworks-database/
Link download database:
https://github.com/Microsoft/sql-server-samples/releases/tag/adventureworks

I. BUILT-IN FUNCTIONS

1. DATE – DATETIME FUNCTIONS

- To get curent date/time
  GETDATE() – SYSDATETIME()

Try to see differences:

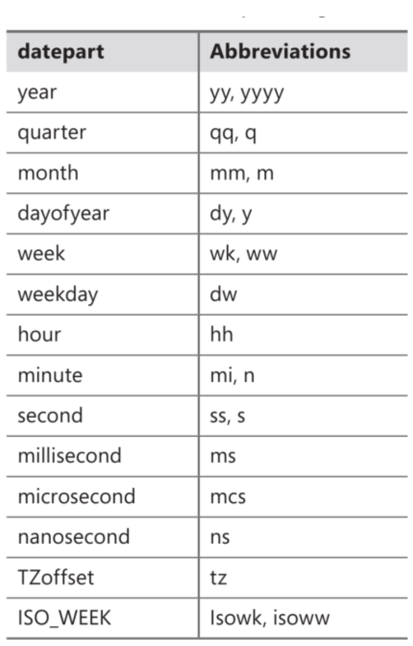
```
SELECT GETDATE() AS GETDATE, SYSDATETIME() AS SYSDATETIME
```

The GETDATE() function returns a DATETIME data type, and the SYSDATETIME function returns a *datetime2*

- To return parts of datetime:

```
SELECT
     DAY(GETDATE()) AS DAY,
     MONTH(GETDATE()) AS MONTH,
     YEAR(GETDATE()) AS YEAR,
     DATENAME(WEEKDAY, GETDATE()) AS DATENAMEWeekDay,
     DATEPART(M, GETDATE()) AS DATEPART,
     DATEPART(WEEKDAY, GETDATE()) AS DatePartWeekDay,
     DATENAME(MONTH, GETDATE()) AS DateNameMonth
```

The return values for the frst three functions, DAY, MONTH, and YEAR, are obvious. However, the last two functions, DATENAME and DATEPART, offer a little more functionality. Unlike the first three functions, both DATENAME and DATEPART accept an additional parameter known as *datepart* . The *datepart* parameter tells the function which part of the date to return.
The valid arguments for Datepart:



datepart	Abbreviations
year	yy, yyyy
quarter	qq, q
month	mm, m
dayofyear	dy, y
week	wk, ww
weekday	dw
hour	hh
minute	mi, n
second	ss, s
millisecond	ms
microsecond	mcs
nanosecond	ns
TZoffset	tz
ISO_WEEK	Isowk, isoww

- To derive dates from parts:

```
SELECT
     DATEFROMPARTS ( 1972, 5, 26) AS DATEFROMPARTS,
     DATETIME2FROMPARTS ( 1972, 5, 26, 7, 14, 16, 10, 3 ) AS DATETIME2FROMPARTS,
     DATETIMEFROMPARTS ( 1972, 5, 26, 7, 14, 16, 10) AS DATETIMEFROMPARTS,
     DATETIMEOFFSETFROMPARTS ( 1972, 5, 26, 7, 14, 16, 10, 12, 0, 3 ) AS
DATETIMEOFFSETFROMPARTS,
     SMALLDATETIMEFROMPARTS ( 1972, 5, 26, 7, 14) SMALLDATETIMEFROMPARTS,
     TIMEFROMPARTS(7, 14, 16, 10, 3) TIMEFROMPARTS
```

- Differencing, modifying, and validating date values

Try for example:

```
SELECT
     DATEDIFF(dd, GETDATE(), '5/26/2013') AS DaysUntilMyBirthday,
     DATEADD(y, 1, GETDATE()) AS DateAdd,
     EOMONTH(GETDATE()) AS EOMonth, --New to SQL Server 2012
     ISDATE(GETDATE()) AS IsValidDate,
     ISDATE('13/1/2122') AS InvalidDate
```

Using DATEDIFF, you are able to fnd out how many days, months, or years exist between two date values. The *datepart* argument, which is *dd*, determines which date part to return. DATEADD also uses a *datepart* argument; however, it can add or subtract from a date value. The EOMONTH function, which is new to SQL Server 2012, returns the last day of the month for a given date value. Finally, you can determine whether or not a date is valid by using the ISDATE function. The following fgure shows the results of the previous query. Notice that 1 is returned when the date is valid, and 0 is returned for invalid dates

2. CONVERSION functions

CAST – CONVERT – TRY_CAST- TRY_CONVERT

- Using cast

```
USE AdventureWorks2012;
SELECT TOP(10)
      SalesOrderNumber,
      TotalDue,
      CAST(TotalDue AS decimal(10,2)) AS TotalDueCast,
      OrderDate,
      CAST(OrderDate AS DATE) AS OrderDateCast
FROM Sales.SalesOrderHeader;
```

While the CONVERT and CAST functions perform the same primary function, the CONVERT function offers some added flexibility over CAST in that you can format the output of your result set using the *style* argument. You can apply styles to *date*, *time*, *real*, *float*, *money*, *xml*, and *binary* data types
http://msdn.microsoft.com/en-us/library/ms187928

- Using convert

```
SELECT
     CONVERT(VARCHAR(20), GETDATE()) AS [Default],
     CONVERT(VARCHAR(20), GETDATE(), 100) AS DefaultWithStyle,
     CONVERT(VARCHAR(10), GETDATE(), 103) AS BritishFrenchStyle,
     CONVERT(VARCHAR(8), GETDATE(), 105) AS ItalianStyle,
     CONVERT(VARCHAR(8), GETDATE(), 112) AS ISOStyle,
     CONVERT(VARCHAR(15), CAST('111111.22' AS MONEY), 1) AS MoneyWithCommas
```

What you should notice immediately is that the CONVERT function accepts three arguments. The frst argument is the target data type or the date type you want to convert a given value to. The second argument is the actual value that will be converted, and the fnal argument is the style. This fnal argument is optional, and if it is not provided, SQL Server will use default values.
The first fve columns represent date conversions to different country styles. The last column illustrates the use of the CONVERT function to add commas to a value that is of the *money* data type. In the query, the CAST function is used in the last line of code to convert the string to *money*, and then the CONVERT function is used to convert that value to back to a string with commas

- Using new conversion functions

Instead of failing the execution of the query, the new TRY functions provide a more elegant approach in returning NULL values.

```
SELECT
     TRY_CAST('PATRICK' AS INT) TryCast,
     TRY_CONVERT(DATETIME, '13/2/2999', 112) AS TryConvert,
     PARSE('Saturday, 26 May 2012' AS DATETIME USING 'en-US') AS Parse,
     TRY_PARSE('Patricks BirthDay' AS DATETIME USING 'en-US') AS TryParse
```

You should use PARSE only when converting from strings to date/time and number data types

3. **STRING functions**
   - Using CONCAT

```
SELECT
     'LEBLANC '+', '+' PATRICK' RawValues,
     RTRIM('LEBLANC ')+', '+LTRIM(' PATRICK') TrimValue,
     LEFT('PatrickDTomorr', 7) [Left],
     RIGHT('DTomorrLeBlanc', 7) [Right],
     SUBSTRING('DTomorrPatrick',8,LEN('DTomorrPatrick')) [SubString],
     '12/'+CAST(1 AS VARCHAR)+'/2012' WithoutConcat,
     CONCAT('12/',1,'/2012') WithConcat
```

The first two columns in the result set concatenate two string values. The difference between the two is that one contains spaces before and after the comma, and the other does not. By using the RTRIM and LTRIM functions, you can remove spaces to the right (RTRIM) and left (LTRIM) of the string. The next two columns use the LEFT and RIGHT functions, which simply return the leftmost or rightmost value of the provided string value based on the second argument. In the preceding query, the LEFT function returns the first seven characters starting from the left, and the RIGHT function does the same but starting from the right.

http://msdn.microsoft.com/en-us/library/ms181984

4. **Logical functions**
   - Using Choose and IFF function

```
declare @choosevar int = 3
SELECT
     CHOOSE(@choosevar, 'ONE', 'TWO', 'PATRICK', 'THREE') [Choose],
     IIF(DATENAME(MONTH, GETDATE()) = 'July', 'The 4th is this month', 'No Fireworks') AS
[IIF]
```

In the preceding query, you are able to select the third item of a list of strings by using the CHOOSE function. If you changed 3 to 1, the function would return the frst value, ONE, from the list of strings instead of the third.
In the fnal line of code in the preceding query, the IIF function is used to determine which of the two strings to return. The frst argument used in the IIF function evaluates to either true or false In this example, an expression is used to determine if the current month is July. If the expression evaluates to true, the frst string value is returned; otherwise, the second value is returned.

5. **Common table expressions**

A common table expression (CTE) is a temporary result set that is defned during the execution of a SELECT, INSERT, UPDATE, DELETE, or CREATE VIEW statement. The CTE is available only for the duration of the query and is not stored like other objects in the database.

WITH <expression_name> [(column_name [,...n])]
AS
(CTE_query_definition)

expression_name is how the CTE will be referenced in the query, and it is required. The column listing is optional but recommended; if the column names in the query defnition are not unique, you will get an error when executing the query. You can either use the column listing to correct the problem or fx it in the query defnition with aliasing.

```
USE AdventureWorks2012;
WITH EmployeePOs (EmployeeID, [Total Due])
AS
(
     SELECT
          poh.EmployeeID,
          CONVERT(varchar(20), SUM(poh.TotalDue),1)
     FROM Purchasing.PurchaseOrderHeader poh
     GROUP BY
          poh.EmployeeID
)
SELECT *
FROM EmployeePOs
```

6. Table variables

They are typically used to store small amounts of data (less than 500 rows) and are only available within the scope of the batch, function, or stored procedure in which they are declared.

```
DECLARE @local_variable [AS] table
(
     [(column_definition) [,...n])]
)
```

```
DECLARE @EmployeePOs AS TABLE
(
     EmployeeID int,
     TotalDue money
)

INSERT INTO @EmployeePOs
SELECT
     poh.EmployeeID,
     CONVERT(varchar(20), SUM(poh.TotalDue),1)
FROM Purchasing.PurchaseOrderHeader poh
GROUP BY
     poh.EmployeeID
```

```
SELECT
     ep.EmployeeID,
     p.FirstName,
     p.LastName,
     ep.[TotalDue]
FROM @EmployeePOs ep
INNER JOIN Person.Person p
     ON ep.EmployeeID = p.BusinessEntityID
```

7. Temporary table

Local temporary tables are available within the scope of the current session and are dropped at the end of a session. They must be prefxed with a pound (#) sign.

Global temporary tables are available for all sessions and are dropped when the session that created them and all referencing sessions are closed. They must be prefxed with two pound (##) signs.

```
USE AdventureWorks2012;
CREATE TABLE #EmployeePOs
(
     EmployeeID int,
     TotalDue money
)

INSERT INTO #EmployeePOs
SELECT
       poh.EmployeeID,
       CONVERT(varchar(20), SUM(poh.TotalDue),1)
  FROM Purchasing.PurchaseOrderHeader poh
  GROUP BY
       poh.EmployeeID

SELECT
     ep.EmployeeID,
     p.FirstName,
     p.LastName,
     ep.[TotalDue]
FROM #EmployeePOs ep
INNER JOIN Person.Person p
     ON ep.EmployeeID = p.BusinessEntityID
```

II. Handling T-SQL Errors

You can wrap the code in the TRY block, and if an error occurs, the control is sent to the CATCH block. Within the CATCH block, you should enclose T-SQL code that will handle the errors.

```
BEGIN TRY
     { sql_statement |statement_block}
END TRY
BEGIN CATCH
     [{ sql_statement |statement_block}]
END CATCH
```

Microsoft introduced the THROW statement, which raises an exception and transfers execution to a CATCH block.

```
THROW [ { error_number | @local_variable },
   { message | @local_variable },
  { state | @local_variable }
] [ ; ]
```

error_number must be between 50,000 and 2,147,483,647, and it can be a constant or variable, but it's optional when implementing error handling using T-SQL. *message* describes the error and can be a string or variable. *state* must be between 0 and 255 and can be a constant or variable. The statement preceding the THROW statement must end with a semicolon (;).

- Try to see differences

```
BEGIN TRY
     SELECT 1/0;
END TRY
BEGIN CATCH

END CATCH
```

```
BEGIN TRY
     SELECT 1/0;
END TRY
BEGIN CATCH
     THROW;
END CATCH
```

III. Controlling flow keywords

- BEGIN...END
- BREAK
- CONTINUE
- GOTO
- IF...ELSE
- RETURN
- WAITFOR
- WHILE

BEGIN...END

The BEGIN...END keyword coupling simply encloses a group or series of T-SQL statements. BEGIN... END blocks can be nested.

```
BEGIN
{
  sql_Statement | statement_block
}
END
```

```
USE AdventureWorks2012;
BEGIN
     DECLARE @StartingHireDate datetime = '12/31/2001'

     SELECT e.BusinessEntityID, p.FirstName, p.LastName, e.HireDate
     FROM HumanResources.Employee e
     INNER JOIN Person.Person p
          ON e.BusinessEntityID = p.BusinessEntityID
     WHERE HireDate <= @StartingHireDate
END
```

IF...ELSE

The IF...ELSE block simply tells the programming language to perform a T-SQL statement or a set of statements if the specified condition is met, or another T-SQL statement or set of statements if it is not. The IF can exist without the ELSE, but the ELSE cannot exist without the IF.

```
IF Boolean_expression { sql_statement | statement_block }
 [ ELSE { sql_statement | statement_block } ]
```

```
IF(DATENAME(M, GETDATE())='December')
BEGIN
     SELECT 'Time for the holidays!!!!' Results
END
ELSE
BEGIN
     SELECT 'Not sure what''s going on now :(' Results
END
```

WHILE

WHILE is a looping mechanism based on a Boolean expression. As long as the expression evaluates to true, the specified T-SQL statement or code block will execute. Two optional keywords, BREAK and CONTINUE, can be included with the WHILE keyword to assist in controlling logic inside the loop. If at any point during the WHILE loop the BREAK keyword causes the execution of the query to exit, any T-SQL code following the END keyword will be executed. The CONTINUE keyword, on the other hand, causes the loop to restart. Any statements after the CONTINUE keyword are ignored.

```
WHILE Boolean_expression
  { sql_statement | statement_block | BREAK | CONTINUE }
```

```
DECLARE @count int = 0
WHILE (@count < 10)
BEGIN
     SET @count = @count + 1;
          IF(@count < 5)
          BEGIN
               SELECT @count AS Counter
               CONTINUE;
          END
          ELSE
               BREAK;
END
```

IV. User defined functions

1. User defined scalar function

A user-defined scalar function is a routine that returns a single value. These functions are often used to centralize the logic of a complex calculation that may be used by several other database or application resources. The syntax is as follows:

```
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ][ type_schema_name. ] parameter_data_type
 [ = default ] [ READONLY ] }
 [ ,...n ]
 ]
)
RETURNS return_data_type
 [ WITH <function_option> [ ,...n ] ]
 [ AS ]
 BEGIN
 function_body
 RETURN scalar_expression
 END
[ ; ]
```

Parameterizing functions

Although this section specifically discusses scalar functions, parameterizing applies to both types of functions. A parameter, in the scope of T-SQL function programming, is an input value that can be passed from the calling function into the code. A parameter can be set to a constant, a column from a table, an expression, and other values. Functions can contain three types of parameters:

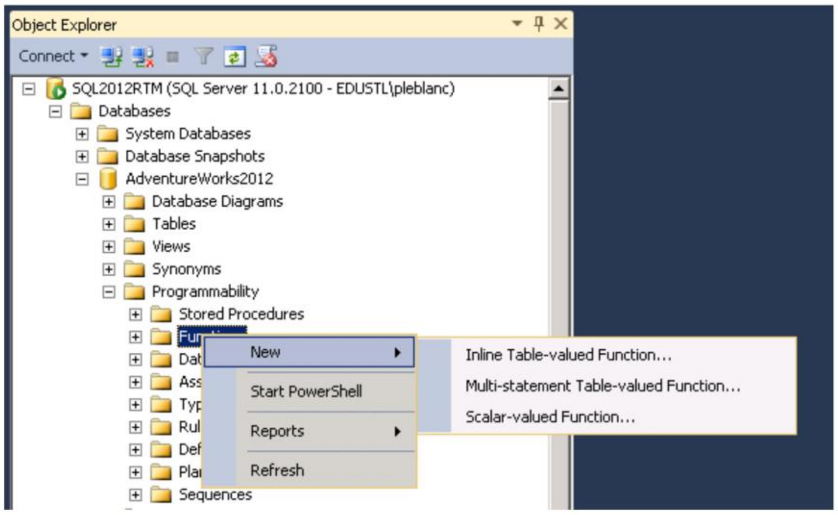
- **Input** This is the value passed into the body of the function.
- **Optional** As the name indicates, this parameter is not required to execute the function.
- **Default** This parameter indicates when a value is assigned to the parameter during creation. In other words, it is a value that is specified when the function is created.

The following sample script demonstrates how to specify each parameter:

```
--Input Parameter
CREATE FUNCTION dbo.Input
@parameter1 int
…
--Optional Parameter
CREATE FUNCTION dbo.Optional
@parameter1 int = NULL
…

--Default Parameter
CREATE FUNCTION dbo.Default
@parameter1 int = 1
…
```

To create functions: Right-click the Functions folder and select New | Scalar-valued Function.



```
USE AdventureWorks2012
-- =================================================
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- =============================================
-- Author: Patrick LeBlanc
-- Create date: 7/8/2012
-- Description: Scalar function that will be used to return employee age
-- =============================================
CREATE FUNCTION dbo.GetEmployeeAge
(
    @BirthDate datetime
)
RETURNS int
AS
BEGIN
     -- Declare the return variable here
     DECLARE @Age int
     -- Add the T-SQL statements to compute the return value here
     SELECT @Age = DATEDIFF(DAY, @BirthDate, GETDATE())
     -- Return the result of the function
     RETURN @Age
END
GO
```

Open a new query window and paste in the following code:

```
USE AdventureWorks2012;
SELECT TOP(10)
     p.FirstName, p.LastName, e.BirthDate,
 dbo.GetEmployeeAge(BirthDate) EmployeeAge
FROM HumanResources.Employee e
INNER JOIN Person.Person p
     ON e.BusinessEntityID = p.BusinessEntityID
```

If a default value was assigned to the function, the function call would resemble the following syntax:

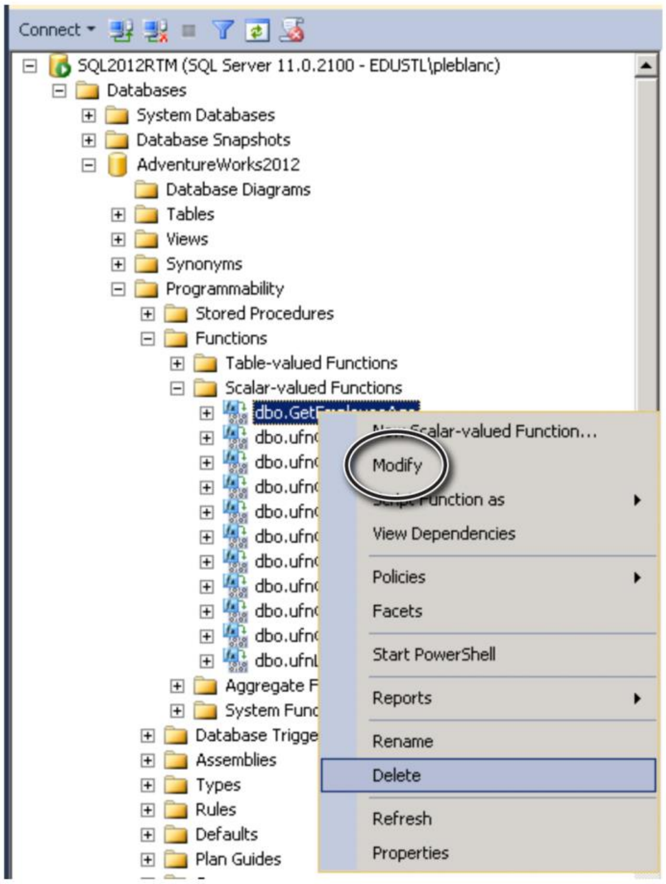
```
dbo.GetEmployeeAge(DEFAULT)
```

To drop the function, execute the following T-SQL. (Alternatively, you can right-click the function in Object Explorer and select Delete from the context menu.)

```
DROP FUNCTION dbo.GetEmployeeAge
```

To modify the functions:

Right-click the dbo.GetEmployeeAge function and select Modify from the context menu.



Executing scalar functions

Scalar functions can be called using two methods:

- Within a SELECT statement (as demonstrated in step 10 in the "Create, alter, and drop a user-defined scalar function" exercise)
- By using the EXECUTE keyword

Regardless of the method you use to select the output, if the parameter values are consistent, the results from either execution will be the same.

Calling scalar functions inline

As previously stated, a scalar function can be included in a SELECT statement. The parameters can be a column, constant, or expression.

```
SELECT dbo.GetEmployeeAge ('5/26/1972')
```

With multiple parameters:

```
USE [AdventureWorks2012]
GO
IF(OBJECT_ID('dbo.GetEmployeeAge')) IS NOT NULL
     DROP FUNCTION dbo.GetEmployeeAge
GO
CREATE FUNCTION [dbo].[GetEmployeeAge]
(
     @BirthDate datetime = '5/26/1972', --DEFAULT
     @Temp datetime = NULL --OPTIONAL
)
RETURNS int
AS
BEGIN
     -- Declare the return variable here
     DECLARE @Age int
     -- Add the T-SQL statements to compute the return value here
     SELECT @Age = DATEDIFF(Year, @BirthDate, GETDATE())
     -- Return the result of the function
     RETURN @Age
END
```

Call the function

```
--Input and Optional Parameters
SELECT dbo.GetEmployeeAge('5/26/1972', NULL)
--Default and Input Parameters
SELECT dbo.GetEmployeeAge(DEFAULT, '1/10/1972')
```

Calling scalar functions using the EXECUTE keyword

In the previous section, you learned how to call a scalar function inline. A scalar function can also be called using the EXECUTE keyword, which is discussed in more detail in Chapter 17, “Stored Procedures.” For now, it is sufficient to know that you can use this keyword to execute scalar functions. To obtain the output of a scalar function using the EXECUTE keyword, you must declare a variable that will hold the output:

```
USE AdventureWorks2012;
GO
DECLARE @Age int;
EXECUTE @Age = dbo.GetEmployeeAge @BirthDate = '5/26/1972'
SELECT @Age;
```

2. Table valued functions

Table-valued functions come in two types:

- Inline
- Multistatement

The inline function simply returns a result set, and the multistatement function offers the ability to include logic within the body of the function. Both return a complete result, similar to selecting from a table or view, but the multistatement function can perform logic and return data. Sample syntax for both is as follows:

```
--Inline Table-Valued Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
 [ = default ] [ READONLY ] }
 [ ,...n ]
 ]
)
RETURNS TABLE
 [ WITH <function_option> [ ,...n ] ]
 [ AS ]
 RETURN [ ( ] select_stmt [ ) ]
[ ; ]

--Multistatement Table-Valued Function Syntax
CREATE FUNCTION [ schema_name. ] function_name
( [ { @parameter_name [ AS ] [ type_schema_name. ] parameter_data_type
 [ = default ] [READONLY] }
 [ ,...n ]
 ]
)
RETURNS @return_variable TABLE <table_type_definition>
 [ WITH <function_option> [ ,...n ] ]
 [ AS ]
 BEGIN
 function_body
 RETURN
 END
[ ; ]
```

Right-click the folder labeled Functions and select New | Inline Table-valued Function.

A query window opens with a template you can use as a starting point for creating the scalar function. A few modifications have been added to the template.

```
 USE AdventureWorks2012;
 SET ANSI_NULLS ON
 GO
 SET QUOTED_IDENTIFIER ON
 GO
 -- =============================================
 -- Author: Patrick LeBlanc
 -- Create date: 6/8/2012
 -- Description: Returns the line items for a given orderid
 -- =============================================
CREATE FUNCTION dbo.GetOrderDetails
(
     @SalesOrderID int
)
RETURNS TABLE
AS
RETURN
(
     SELECT
          sod.SalesOrderID,
          sod.SalesOrderDetailID,
          sod.CarrierTrackingNumber,
          p.Name ProductName,
          so.Description
     FROM Sales.SalesOrderDetail sod
     INNER JOIN Production.Product p
          ON sod.ProductID = p.ProductID
     INNER JOIN Sales.SpecialOffer so
          ON sod.SpecialOfferID = so.SpecialOfferID
     WHERE
          sod.SalesOrderID = @SalesOrderID
)
GO
```

Open a new query window and paste in the following code:

```
USE AdventureWorks2012;
SELECT *
FROM dbo.GetOrderDetails(43659);
```

Limitations of functions

Just as with any object inside SQL Server, user-defined functions have limitations. One limitation is that you cannot use a TRY...CATCH block inside a function. Therefore, you have to create your own error-handling mechanisms to elegantly handle errors. A limitation specific to scalar functions is that they cannot return *text*, *ntext*, *image*, *cursor*, or *timestamp* data types. Finally, user-defined functions cannot be used to modify the database state. Using functions in a SELECT statement could adversely affect the performance of the query. This is because the function will be called once for every row returned. Therefore, be cautious using complex functions when returning large result sets.