T-SQL Programming – Part 1

Transact-SQL (T-SQL) is Microsoft's (and Sybase's – now owned by SAP) proprietary extension to SQL. SQL is a standardized computer language that was originally developed by IBM for querying, altering and defining relational databases, using declarative statements. SQL is the only way to retrieve data from a relational database – period end of discussion!

Startup

Perform the following startup tasks.

1. Download the *Northwind* ZIP file from the Labs/T-SQL item on Blackboard. Save the ZIP file to your USB drive or to *My Documents*. Extract the contents.
2. Attach the *Northwind* database using **SQL Server Management Studio**.

Basic Information

T-SQL expands on the SQL standard to include procedural programming, local variables, various support functions for string processing, date processing, mathematics, etc. and changes to the DELETE and UPDATE statements.

T-SQL scripts are stored as plain ASCII text files that you can edit with any text editor such as Notepad. SQL Server Management Studio (SSMS) also has a built-in text editor that can be used to edit T-SQL scripts. By convention, T-SQL scripts are named using a "dot SQL"(.SQL) extension. For example, *CreateDatabase.SQL*. Double-clicking a T-SQL script (that has the dot SQL extension) in the Windows File Explorer will open the script in the SSMS text editor. Opening a script in this way will not cause the script to execute.

Batch Processing

T-SQL scripts are composed of a series of "batches" or blocks of SQL commands. Any script can contain one or more batches. Batches are created by separating blocks of code with the T-SQL GO command. GO is not a Transact-SQL statement; it is a command recognized by the sqlcmd and osql utilities and SQL Server Management Studio Code editor.

SQL Server utilities interpret GO as a signal that they should send the current batch of Transact-SQL statements to an instance of SQL Server. The current batch of statements is composed of all statements entered since the last GO, or since the start of the ad hoc session or script if this is the first GO. All scripts are assumed to have an implicit GO command as the last command in a T-SQL script.

A Transact-SQL statement cannot occupy the same line as a GO command. However, the line can contain comments.

```
USE AdventureWorks2012;
GO
DECLARE @MyMsg VARCHAR(50)
SELECT @MyMsg = 'Hello, World.'
GO – The variable @MyMsg is not valid after this GO ends the batch.

-- Yields an error because @MyMsg not declared in this batch.
```

```
PRINT @MyMsg
GO
```

Note, other than to resolve ambiguity, T-SQL syntax does not require a semicolon to terminate a statement. Despite this, I recommend using a semicolon to terminate a T-SQL statement because it makes code cleaner, more readable, easier to maintain, and more portable.

Commenting T-SQL Scripts

T-SQL interpreters support two types of comments – single/inline line comments and block (or multi-line) comments. Single line comments begin with two (2) dash characters followed by the comment text which is assumed to extend to the end of the current line. The dashes can begin anywhere on the line and are often used on the same line as a T-SQL statement to comment that statement. For example, refer to the line containing the first **GO** command above.

Block comments can be inserted on a separate line or within a Transact-SQL statement. Multiple-line comments must be indicated by /* and */. A stylistic convention often used for multiple-line comments is to begin the first line with /*, subsequent lines with **, and end with */.

There is no maximum length for comments.

Nested comments are supported. If the /* character pattern occurs anywhere within an existing comment, it is treated as the start of a nested comment and, therefore, requires a closing */ comment mark. If the closing comment mark does not exist, an error is generated.

Here is are a couple of examples of a typical use of a block comment.

```
/*
 * text_of_comment
*/
```

```
USE AdventureWorks2012; /* This is a comment.  Unlike a single line comment,
                         * this comment can wrap to the next line.
                         */
```

Defining Variables

Local Variables

As with any programming language, T-SQL allows you to define and set variables. A variable holds a single piece of information, similar to a number or a character string. Variables can be used for a number of things. Here is a list of a few common variable uses:

- To pass parameters to stored procedures, or function
- To control the processing of a loop
- To test for a true or false condition in an IF statement
- To programmatically control conditions in a WHERE statement

In SQL Server a variable is typical known as a local variable, due the scope of the variable. The scope of a variable describes how visible or accessible the variable is with a particular T-SQL program. The scope of a local variable is restricted to the batch, stored procedure or code block in which it is defined. A local variable is defined using the T-SQL *DECLARE* statement.

T-SQL language syntax requires that the name of a local variable must begin with a @ (at sign). A local variable can be declared as any system or user defined SQL data type. Here is a typical declaration for an integer variable named @CNT.

```
DECLARE @CNT INT;
```

More than one variable can be defined with a single DECLARE statement. To define multiple variables, with a single DECLARE statement, you separate each variable definition with a comma.

```
DECLARE @CNT INT, @X INT, @Y INT, @Z CHAR(10);
```

Above I have defined 4 local variables with a single DECLARE statement. A local variable is initially assigned the NULL value. Remember in SQL, NULL is the absence of data. Which means newly declared variables are initialized.

A value can be assigned to a local variable by using the SET or SELECT statement. Yes, that's correct, a SELECT. Just like the SELECT you studied in your database class. On the SET command you specify the local variable and the value you wish to assign to the local variable. Here is an example of where I have defined the @CNT variable and then initialize the variable to 1.

```
DECLARE @CNT INT;
SET @CNT = 1;
```

Here is an example of how to use the SELECT statement to set the value of a local variable.

```
DECLARE @ROWCNT INT;
SELECT @ROWCNT=COUNT(*) FROM Northwind.dbo.Customers;
```

The above example sets the variable @ROWCNT to the number of rows in the *Customers* table which is stored in the *Northwind* database under the *dbo* schema.

One of the uses of a variable is to programmatically control the records returned from a SELECT statement. You do this by using a variable in the WHERE clause. Here is an example that returns all the Customers records in the Northwind database where the Customers Country column is equal to 'Germany'

```
DECLARE @City varchar(25);
SET @City = 'Boston';
SELECT City FROM Northwind.dbo.Customers WHERE City = @City;
```