T-SQL Programming – Part 3

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**.

Building a T-SQL Loop

A programming loop is a block of T-SQL code that is executed over and over again. In the loop, some logic is executed repeatedly until a condition is met that allows the code to break out of the loop. One example of where you might use a loop would be to process through a set of records one record at a time. Another example might be where you need to generate some test data and a loop would allow you to insert a record into your test data table with slightly different column values, each time the loop is executed. In this lab we will discuss the WHILE, BREAK, CONTINUE, and GOTO T-SQL statements.

WHILE Statement

In T-SQL, the WHILE statement is the most common way to execute a loop. Here is the basic syntax for a WHILE loop:

WHILE <Boolean expression> <code block>

Where a <Boolean expression> is any expression that equates to a true or false answer, and the <code block> is the desire code to be executed while the <Boolean expression> evaluates to true. Let's explore a simple example.

In this example, we will increment a counter from 1 to 10 and display the counter each time through the WHILE loop.

```
DECLARE @counter INT;
SET @counter = 0;
WHILE @counter < 10
BEGIN
	SET @counter = @counter + 1;
	PRINT 'The counter is ' + CAST(@counter as CHAR);
END;
```

Here, the T-SQL code between the BEGIN and END executes repeately as long as the @counter integer variable is strictly less than 10 (the Boolean expression of the WHILE loop). The @counter variable is explicitly set to zero at the outset and each time through the WHILE loop it is incremented by 1. The PRINT statement displays the value in the @counter variable each time through the WHILE loop. Copy this code to SSMS and try it.

As you can see, once the @counter variable reaches 10 the Boolean expression that is controlling the WHILE loop is no longer true. When that happens, the code within the WHILE loop is no longer

executed and the loop is exited. Execution of the script resumes with the statement immediately after the END statement.

Not only can you have a single WHILE loop, but you can also have WHILE loops inside WHILE loops which is commonly know as nested WHILE loops. There are lots of different tasks where nesting is valuable. For example, I commonly use nesting of WHILE loops to generate test data.

BREAK and CONTINUE Statements

Often you want to construct a loop that will iterate the loop's T-SQL code once for each value of your counter variable. Sometimes however, you may want to break out of the loop if some particular condition arises. Also, in addition to breaking out of the loop, you may not want to process all the code in the loop before going back to the top of the loop and starting through the next iteration of the loop. For these kinds of situations, T-SQL provides the BREAK and CONTINUE statements.

The BREAK statement exits out of the inner most WHILE loop, and proceeds to the statement following the END statement that is associated with the loop in which the BREAK statement was executed.

The CONTINUE statement, on the other hand, causes the execution of and T-SQL statements between the CONTINUE statement and the END statement to be skipped and causing the loop to return to the top of the loop and execute the logical test to determine if the loop should be re-entered.

For example, the T-SQL code below generates #Part table records. Whenever Category_ID is 2 and PART_ID is 1, we are going to BREAK out of the inner WHILE loop.

```
DECLARE @Part_Id int;
DECLARE @Category_Id int;
DECLARE @Desc varchar(50);

IF OBJECT_ID('tempdb..#Part') IS NOT NULL
BEGIN
       PRINT 'Deleting temporary table';
       DROP TABLE #Part;
END

CREATE TABLE #Part (Part_Id INT, Category_Id INT, Description VARCHAR(50));
SET @Part_Id = 0;
SET @Category_Id = 0;

WHILE @Part_Id < 2
BEGIN
       SET @Part_Id = @Part_Id + 1
       WHILE @Category_Id < 3
       BEGIN
              SET @Category_Id = @Category_Id + 1;
              IF @Category_ID = 2 and @Part_ID = 1 BREAK;
              SET @Desc = 'Part_Id is ' + cast(@Part_Id as char(1)) +
              ' Category_Id ' + cast(@Category_Id as char(1));

              INSERT INTO #Part VALUES(@Part_Id, @Category_Id, @Desc )
       END
       SET @Category_Id = 0
END
SELECT * FROM #Part;
```

Copy the above code into SSMS and try it.

From the output you can see that no records were inserted for Category_Id =2 or 3 when the Part_Id = 1. Where as there are records for Part_Id = 2 with all values for the Category_Id column. This is because the IF statement in the inner loop forced the BREAK statement to exit the inner loop. Since there were records generated for Part_Id = 2, this shows that the BREAK statement only exited the inner loop and not the outer loop.

Now, replace the BREAK statement with a CONTINUE statement and re-execute the code.

As you can see, when we use the CONTINUE statement only the record with Category_Id = 2 and Part_Id = 1 is missing. This is because the CONTINUE statement does not break out of the inner WHILE loop but only goes back to the top of the WHILE loop without inserting the record. This happens only when Category_Id is 2 and Part_Id is equal to 1. When Part_Id = 1 and Category_Id = 3 the INSERT statement is still executed.

> ***Sidebar – What is that weird # table name?***
>
> *Temporary tables are similar to permanent tables, except temporary tables are stored in a system database called tempdb and are deleted automatically when they are no longer used.*
>
> *There are two types of temporary tables: local and global. They differ from each other in their names, their visibility, and their availability. Local temporary tables, like the #Part table above, have a single number sign (#) as the first character of their names. These tables are visible only to the current connection for the user and they are deleted when the user disconnects from the instance of SQL Server.*
>
> *Global temporary tables have two number signs (##) as the first characters of their names. They are visible to any user after they are created and they are deleted only when all user's connections referencing the table disconnect from the instance of SQL Server.*
>
> *For example, if you create the table employees, the table can be used by any person who has the database security permissions to use it or until the table is deleted. If a database session creates the local temporary table #employees, only that session can work with the table and the table is deleted when the session disconnects.*
>
> *If you create the global temporary table ##employees, any user in the database can work with this table. If no other user works with this table after you create it, then the table is deleted when you disconnect. However, if another user works with the table after you create it, SQL Server deletes it only after all sessions that are using the table have been disconnected.*

GOTO Statement

The BREAK statement will only extract you from the currently processing WHILE loop. However, occasionally, no matter how deeply nested you might be in WHILE loops, you would like to have your code break out of all WHILE loops and escape back to free code. If this is what you want, then you will need to use the T-SQL GOTO statement.

Most programmers cringe at the thought of using the GOTO statement; however, there is a reason that nearly every programming language has one – sometimes, they are just damn useful!

The following example shows how to use GOTO as a branch mechanism to one of three “Branch” labels –

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

WHILE @Counter < 10
BEGIN
	PRINT 'Current Value of @Counter :' + CAST(@Counter AS VARCHAR);
	SET @Counter = @Counter + 1;
	IF @Counter = 4 GOTO Branch_One; --Jumps to the first branch.
	IF @Counter = 5 GOTO Branch_Two;  --This will never execute.
END

-- The following line is called a "label". You can only GOTO labels.
Branch_One:
	PRINT 'Jumped To Branch One.';
	GOTO Branch_Three; --This will prevent Branch_Two from executing.

-- By convention, labels appear alone on a line but as you can see they don't have to.
Branch_Two: PRINT 'Jumped To Branch Two.';

Branch_Three:
	PRINT'Jumped To Branch Three.';
```

Conclusion

Hopefully now you have a better idea of how to code a T-SQL WHILE loop. We've seen how to control the WHILE loop, break out of a loop by using the BREAK statement, use the CONTINUE statement to skip some of the code in the while loop, and/or break out of all WHILE loops using the GOTO statement. The techniques I've described should give you the basis for building all your WHILE statements from a single WHILE loop to a complex set of nested WHILE loops.