



Programming In T-SQL

Transact SQL

- T-SQL is SQL Server's own dialect of Structured Query Language (SQL)
- T-SQL
 - Stands for Transact SQL
 - Procedural Programming Language + SQL
- T-SQL extends SQL by adding constructs found in other procedural languages, such as
 - Variables, data types and operators
 - Flow Control Structures
 - Procedures and Functions (user-defined and built-in)

Scripts

- Scripts generally have a unified goal
 - All the statements within a script have one overall purpose
- SQL scripts are stored as text files
- Scripts are usually treated as a unit
- You normally execute the entire script or nothing at all
- SQL script
 - a collection of SQL commands
 - stored in a text file and
 - perform some operation or task

Scripts

- Scripts are usually used for repetitive Tasks
 - get executed over and over on a regular basis
- Advantages include:
 - Ease of Use - The scripts can be saved and loaded when needed.
 - Consistent Operation – Statements are tested and free from errors
 - Scheduled Operation
 - Scripts can be scheduled to run at a time when it is convenient
 - Can be automated



Batches

- A batch is a grouping of T-SQL statements into one logical unit
- Batches are used when something has to happen either before or separately from everything else in your script.
- All statements in a batch are compiled into one execution plan
- Batches are delimited by the GO statement
- The GO statement
 - The GO statement must be on its own line
 - Is not a T-SQL command
 - It is a command recognized by the SQL Server command utilities

Batches (cont.)

- All statements in the same batch

- ☐ Are parsed together
 - Must pass a validation of the syntax as a unit
 - The statements in the batch are compiled into a single execution plan
- ☐ Runtime errors may result in the partial execution of statements in a batch

- Summary

- ☐ If a statement fails at parse-time (syntax error), then nothing runs.
- ☐ If a statement fails at runtime, all statements until the error happened have already run.
- ☐ Each batch is processed independently
 - An error in one batch does not prevent another batch from running

Statements in a Batches

- Some commands must be in their own batch:
 - CREATE PROCEDURE
 - CREATE TRIGGER
 - CREATE VIEW
- You cannot create and object (eg. A table) and use it in the same batch
- If you want to combine any of these statements with other statements in a single script
 - You will need to break them up by using GO statements

Batches and Script - Example

```
USE MyDB
```

```
GO
```

```
    DECLARE @MyMsg VARCHAR(50)
```

```
    SELECT @MyMsg = 'Hello, World.'
```

```
GO                -- @MyMsg is not valid after this GO ends the batch.
```

```
                -- Error because @MyMsg not declared in this batch.
```

```
    PRINT @MyMsg
```

```
GO
```

```
    SELECT @@VERSION;
```

```
                -- Error: Must be EXEC sp_who if not first statement
```

```
    sp_who
```

```
GO
```

- sp_who - Provides information about current users, sessions, and processes

Basic Symbols

- End of statement [;]

- This is optional

- Comments

- Single line [--]

- Multi-line [/*...*/]

- String ['...']

Operators

- Arithmetic Operators [+ , - , * , / , %]
- Comparison Operators
 - >, <, >=, <=, =, <> (or !=, which is the standard)
 - BETWEEN ... AND ...
 - IN (...)
 - LIKE, IS NULL
- Logical Operators
 - AND
 - OR
 - NOT
 - EXISTS
- Concatenation Operator [+]

Data Types

- Data types used in scripts are the same as column data types
- Examples

VARCHAR(50)

INT

DECIMAL(6,2)

- Other Data Types
 - TABLE
 - CURSOR

LOCAL Variables

- The scope of a local variable is the current batch
- Variable declaration syntax:

```
DECLARE      @<variable name> <variable type> [= <value>]  
            [, @<variable name> <variable type> [= <value>] ]]
```

- Example

```
DECLARE @price decimal(4,1)
```

```
DECLARE @product varchar(50), @UnitPrice decimal(6,2)
```

- A variable cannot be of text, ntext, or image data type



GLOBAL Variables

- Global variables represent a special type of variable
- Also referred as SYSTEM-DEFINED Functions
 - The server always maintains the values of these variables
- Global variables represent
 - information specific to the server or
 - information on the current user session
- Global variable names begin with a @@ prefix
 - You do not need to declare them



GLOBAL Variables - Examples

- @@ERROR

- ☐ Returns the error number for the last Transact-SQL statement executed

- @@IDENTITY

- ☐ Returns the last-inserted identity value

- @@ROWCOUNT

- ☐ Returns the number of rows affected by the last statement

- @@SERVERNAME

- ☐ Returns the name of the local server that is running SQL Server

- @@VERSION

- ☐ Returns system and build information for the current installation of SQL Server

Variables Assignment

- The value of a variable will be NULL until it is initialized
- Two ways to set the value in a variable
 - Use the SELECT or SET statement
- Assignment Operator [=]
- Compound assignment operator [+=, *=, etc.]
- Example
 - Variable declaration and assignment
`DECLARE @price decimal(4,1) = 9.9`
 - Variable Assignment
`SET @price = 12.50`
- Multiple Values
`SET @price = 12.50 , @quantity = 12`

SELECT and SET

■ Use SET

- When you are performing a simple assignment of a variable
- Where the value is known (explicit value or from other variable)

■ Use SELECT

- When you are basing the assignment of your variable on a query
- A SELECT statement that assigns a value to a variable must not be combined with data-retrieval operations

■ Use Variables with Queries

```
DECLARE      @product varchar(10) = 'screw'
SELECT      *
FROM        Products
WHERE       ProductName LIKE '%' + @product + '%';
```

Example 1

```
DECLARE    @product varchar(50)
           , @UnitPrice decimal(6,2)
           , @QuantityPerUnit varchar(20)
```

```
DECLARE    @proID int      = 15;
SELECT     @product        = ProductName,
           @QuantityPerUnit = QuantityPerUnit,
           @UnitPrice      = UnitPrice
FROM       Products
WHERE      ProductID = @ proID
```

```
Print     @product + ' is ' + convert(varchar(20)
           , @UnitPrice) + ' BIRR per ' + @QuantityPerUnit
```

Example 2

- Using SET

```
SET @price          =          9.9
SET @TotalCost      =          @UnitCost * 1.1
```

```
DECLARE @MaxSalary money;
SET      @MaxSalary = ( SELECT MAX(Salary)
                        FROM Employee      )
```

SELECT or SET - Examples

```
DECLARE @cat int=7
```

```
SELECT      *  
FROM        Products  
WHERE       CategoryID = @cat
```

■ Assigning a value from a query

```
DECLARE      @rows INT  
SELECT       @rows = COUNT(*) FROM Products  
PRINT  @rows  
  
SET @rows    = (SELECT COUNT(*) FROM Products);  
PRINT  @rows;
```

Table Data Type

```
DECLARE @MyTableVar table
(
    FN varchar(50)
    , LN varchar(50)
)
```

```
INSERT      @MyTableVar (FN, LN)
SELECT      FirstName , FatherName
FROM        Student]
```

```
SELECT      FN, LN
FROM        @MyTableVar
```

Table Data Type 2

- We can UPDATE records in our table variable as well as DELETE records

```
UPDATE @ProductTotals  
SET Revenue = Revenue * 1.15  
WHERE ProductID = 62
```

- Update data in the table variable

```
DELETE FROM @ProductTotals  
WHERE ProductID = 60
```

Table Data Type 3

- Constraints can be used with table variables

```
DECLARE @MyTable TABLE
(
    ProductID int UNIQUE,
    Price money CHECK(Price < 10.0)
)
```

- You can also declare primary keys, identity columns, and default values

```
DECLARE @MyTable TABLE
(
    ProductID int IDENTITY(1,1) PRIMARY KEY,
    Name varchar(10) NOT NULL DEFAULT('Unknown')
)
```

Table Data Type 4

- The table definition of a table variable cannot change after the DECLARE statement
- If you are using a table variable in a join, you will need to alias the table in order to execute the query

```
SELECT    ProductName, Revenue
FROM      Products P JOIN @ProductTotals PT
          ON P.ProductID = PT.ProductID
```



Control-of-Flow Statements

Overview

- T-SQL has the following control of flow statement:
 - IF ... ELSE
 - WHILE
 - GOTO
 - RETURN
- CASE Expression (not a Control of Flow statement)



IF Statement

- The IF statement controls the conditional execution of one or more statements
- An IF statement will execute only the very next statement after it if a statement block is not used
- Statements blocks
 - Created by using BEGIN ... END
 - ALL or NONE of the statements in a block are executed
- Statements blocks
 - Created by using BEGIN ... END
 - ALL or NONE of the statements in a block are executed

Conditional Statements

- Syntax

IF <Boolean Expression>

<SQL statement> | BEGIN <code series> END

[ELSE

<SQL statement> | BEGIN <code series> END]

- The test condition can be any expression that return a Boolean value

- Basic structures

- if ...

- if ... else

- if ... else if ... else

Conditional Statements (cont.)

- Conditional execution of a statement

```
IF @myvar IS NULL
```

```
-- Do something
```

- Checking existence of a table

```
IF NOT EXISTS ( SELECT T.name
```

```
                FROM sys.tables T
```

```
                WHERE T.name = 'Employee' )
```

```
-- CREATE the table 'Employee'
```

```
ELSE
```

```
    PRINT 'The table already exists'
```

The Simple CASE Expression

CASE ... WHEN ... THEN ... END

- Evaluates several conditions and return a single value
- ELSE can be included as a default option
- A simple CASE expression
 - Needs a condition to be specified after the “CASE” keyword
 - The CASE block returns a value or expression
 - The return value can be used with other statements.

```
DECLARE @x int = 20
```

```
PRINT
```

```
CASE @x % 2
```

```
    WHEN 0 THEN 'EVEN number'
```

```
    WHEN 1 THEN 'ODD number'
```

```
END
```

The Search CASE

- Same as a simple CASE, except:
 - There is no input expression
 - The WHEN expression must evaluate to a Boolean value
- The ELSE can still be included as a default option
- Note
 - Even when two conditions evaluate to TRUE, only the first condition is used
 - You can use different conditions for each condition expressions.
 - Any expression that evaluates to a Boolean value can be used

The Search CASE (cont.)

- Example

```
DECLARE @x int = 20
```

```
CASE
```

```
    WHEN @x % 2 = 0 THEN 'EVEN number'
```

```
    WHEN @x % 2 = 1 THEN 'ODD number'
```

```
END
```

Uses of CASE Expr

- Replacing codes or abbreviation to more readable values
- Example

```
SELECT ProductName, Status =  
    CASE Discontinued  
    WHEN 0 THEN 'Discontinued'  
    WHEN 1 THEN 'Active'  
    END  
FROM Products  
ORDER BY Status
```

Uses of CASE Expr (cont.)

- Example - Categorizing data

```
SELECT ProductName ,  
       CASE  
         WHEN UnitPrice < 10 THEN 'Cheap'  
         WHEN UnitPrice BETWEEN 10 AND 20 THEN 'Normal'  
         ELSE 'Expensive'  
       FROM Products
```

-

The WHILE Statement

- The WHILE statement tests a condition and executes the statement as long as the test condition is TRUE
- The syntax:

WHILE *<Boolean expression>*

<sql statement> | [BEGIN

<statement block>

[BREAK]

<sql statement> | *<statement block>*

[CONTINUE]

END]

WHILE Statement - Example

```
DECLARE @counter int = 1
```

```
DECLARE @max int = 10
```

```
WHILE @counter <= @max
```

```
BEGIN
```

```
    PRINT @counter
```

```
    SET @counter = @counter + 1
```

```
END
```

The WHILE Statement (cont.)

- The BREAK statement is a way of exiting the loop without waiting for the bottom of the loop
- The CONTINUE statement tells the WHILE loop to go back to the beginning
 - regardless of where you are in the loop,
 - you immediately go back to the top and re-evaluate the expression (exiting if the expression is no longer TRUE)

WHILE Statement - Example

- What is the result of the following loop?

```
WHILE (SELECT SUM(Salary) FROM Employee) < 150000  
BEGIN
```

```
    UPDATE Employee  
    SET Salary += 100
```

```
    SELECT MIN(Salary) FROM Employee
```

```
    IF ( SELECT MIN(Salary) FROM Employee ) > 7000  
        BREAK
```

```
    ELSE  
        CONTINUE
```

```
END
```



The GOTO Statement

- Causes the flow of execution to a LABEL
- The LABEL is an identifier followed by a colon
- The statements that follow GOTO are skipped and processing continues at the label
- GOTO statements and labels can be used anywhere within a procedure, batch, or statement block
- GOTO and the Label must be in the same Batch

GOTO Statement - Example

```
DECLARE @Counter int;
```

```
SET @Counter = 1;
```

```
WHILE @Counter < 10
```

```
BEGIN
```

```
    SELECT @Counter
```

```
    SET @Counter = @Counter + 1
```

```
    IF @Counter = 4 GOTO BranchOne --Jumps to the first branch.
```

```
    IF @Counter = 5 GOTO BranchTwo --This will never execute.
```

```
END
```

```
BranchOne:
```

```
    SELECT 'Jumping To Branch One.'
```

```
    GOTO BranchThree; --This will prevent BranchTwo from executing.
```

```
BranchTwo:
```

```
    SELECT 'Jumping To Branch Two.'
```

```
BranchThree:
```

```
    SELECT 'Jumping To Branch Three.'
```



The RETURN Statement

- Exits unconditionally from a query or procedure
- RETURN can be used at any point to exit from a procedure, batch, or statement block
- Statements that follow RETURN are not executed